

*Poznaj język wykorzystywany przez
Google i YouTube!*

Wydanie IV

Wprowadzenie

Python



HELION

O'REILLY®

Mark Lutz

Tytuł oryginału: Learning Python, Fourth Edition by Mark Lutz

*Dla Very.
Jesteś moim życiem.*

Spis treści

Przedmowa	29
Część I Wprowadzenie	47
1. Pytania i odpowiedzi dotyczące Pythona	49
Dlaczego ludzie używają Pythona?	49
Jakość oprogramowania	50
Wydajność programistów	51
Czy Python jest językiem skryptowym?	51
Jakie są zatem wady Pythona?	53
Kto dzisiaj używa Pythona?	53
Co mogę zrobić za pomocą Pythona?	55
Programowanie systemowe	55
Graficzne interfejsy użytkownika	55
Skrypty internetowe	56
Integracja komponentów	56
Programowanie bazodanowe	57
Szybkie prototypowanie	57
Programowanie numeryczne i naukowe	57
Gry, grafika, porty szeregowe, XML, roboty i tym podobne	58
Jakie wsparcie techniczne ma Python?	58
Jakie są techniczne mocne strony Pythona?	59
Jest zorientowany obiektowo	59
Jest darmowy	59
Jest przenośny	60
Ma duże możliwości	61
Można go łączyć z innymi językami	62
Jest łatwy w użyciu	62
Jest łatwy do nauczenia się	62
Zawdzięcza swoją nazwę Monty Pythonowi	63
Jak Python wygląda na tle innych języków?	63
Podsumowanie rozdziału	64

Sprawdź swoją wiedzę — quiz	65
Sprawdź swoją wiedzę — odpowiedzi	65
2. Jak Python wykonuje programy?	69
Wprowadzenie do interpretera Pythona	69
Wykonywanie programu	71
Z punktu widzenia programisty	71
Z punktu widzenia Pythona	72
Warianty modeli wykonywania	74
Alternatywne implementacje Pythona	75
Narzędzia do optymalizacji wykonywania	76
Zamrożone pliki binarne	78
Inne opcje wykonywania	78
Przyszłe możliwości?	79
Podsumowanie rozdziału	80
Sprawdź swoją wiedzę — quiz	80
Sprawdź swoją wiedzę — odpowiedzi	80
3. Jak wykonuje się programy?	81
Interaktywny wiersz poleceń	81
Interaktywne wykonywanie kodu	82
Do czego służy sesja interaktywna?	83
Wykorzystywanie sesji interaktywnej	85
Systemowe wiersze poleceń i pliki	87
Pierwszy skrypt	87
Wykonywanie plików za pomocą wiersza poleceń	88
Wykorzystywanie wierszy poleceń i plików	90
Skrypty wykonywalne Uniksa (#!)	91
Kliknięcie ikony pliku	92
Kliknięcie ikony w systemie Windows	93
Sztuczka z funkcją input	94
Inne ograniczenia klikania ikon	95
Importowanie i przeładowywianie modułów	96
Więcej o modułach — atrybuty	98
Uwagi na temat używania instrukcji import i reload	100
Wykorzystywanie exec do wykonywania plików modułów	101
Interfejs użytkownika IDLE	102
Podstawy IDLE	103
Korzystanie z IDLE	105
Zaawansowane opcje IDLE	106
Inne IDE	107
Inne opcje wykonywania kodu	108
Osadzanie wywołań	108
Zamrożone binarne pliki wykonywalne	109
Uruchamianie kodu w edytorze tekstowym	110

Jeszcze inne możliwości uruchamiania	110
Przyszłe możliwości	110
Jaką opcję wybrać?	111
Podsumowanie rozdziału	112
Sprawdź swoją wiedzę — quiz	113
Sprawdź swoją wiedzę — odpowiedzi	113
Sprawdź swoją wiedzę — ćwiczenia do części pierwszej	114
Część II Typy i operacje	117
4. Wprowadzenie do typów obiektów Pythona	119
Po co korzysta się z typów wbudowanych?	120
Najważniejsze typy danych w Pythonie	121
Liczby	122
Łańcuchy znaków	124
Operacje na sekwencjach	124
Niezmienność	126
Metody specyficzne dla typu	126
Otrzymanie pomocy	127
Inne sposoby kodowania łańcuchów znaków	128
Dopasowywanie wzorców	129
Listy	130
Operacje na sekwencjach	130
Operacje specyficzne dla typu	130
Sprawdzanie granic	131
Zagnieżdżanie	131
Listy składane	132
Słowniki	133
Operacje na odwzorowaniach	134
Zagnieżdżanie raz jeszcze	134
Sortowanie kluczy — pętle for	136
Iteracja i optymalizacja	137
Brakujące klucze — testowanie za pomocą if	138
Krotki	139
Czemu służą krotki?	140
Pliki	140
Inne narzędzia podobne do plików	142
Inne typy podstawowe	142
Jak zepsuć elastyczność kodu	143
Klasy zdefiniowane przez użytkownika	144
I wszystko inne	145
Podsumowanie rozdziału	145
Sprawdź swoją wiedzę — quiz	146
Sprawdź swoją wiedzę — odpowiedzi	146

5. Typy liczbowe	149
Podstawy typów liczbowych Pythona	149
Literały liczbowe	150
Wbudowane narzędzia liczbowe	151
Operatory wyrażeń Pythona	152
Liczby w akcji	156
Zmienne i podstawowe wyrażenia	157
Formaty wyświetlania liczb	158
Porównania — zwykłe i łączone	160
Dzielenie — klasyczne, bez reszty i prawdziwe	161
Precyzyja liczb całkowitych	164
Liczby zespolone	165
Notacja szesnastkowa, ósemkowa i dwójkowa	165
Operacje poziomu bitowego	167
Inne wbudowane narzędzia liczbowe	168
Inne typy liczbowe	170
Typ liczby dziesiętnej	170
Typ liczby ułamkowej	172
Zbiory	176
Wartości Boolean	181
Dodatkowe rozszerzenia numeryczne	182
Podsumowanie rozdziału	183
Sprawdź swoją wiedzę — quiz	183
Sprawdź swoją wiedzę — odpowiedzi	184
6. Wprowadzenie do typów dynamicznych	185
Sprawa brakujących deklaracji typu	185
Zmienne, obiekty i referencje	186
Typy powiązane są z obiektami, a nie ze zmiennymi	187
Obiekty są uwalniane	188
Referencje współdzielone	190
Referencje współdzielone a modyfikacje w miejscu	191
Referencje współdzielone a równość	193
Typy dynamiczne są wszędzie	194
Podsumowanie rozdziału	194
Sprawdź swoją wiedzę — quiz	195
Sprawdź swoją wiedzę — odpowiedzi	195
7. Łańcuchy znaków	197
Literały łańcuchów znaków	199
Łańcuchy znaków w apostrofach i cudzysłowach są tym samym	200
Sekwencje ucieczki reprezentują bajty specjalne	200
Surowe łańcuchy znaków blokują sekwencje ucieczki	203
Potrójne cudzysłowy i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami	204

Łańcuchy znaków w akcji	205
Podstawowe operacje	206
Indeksowanie i wycinki	207
Narzędzia do konwersji łańcuchów znaków	210
Modyfikowanie łańcuchów znaków	213
Metody łańcuchów znaków	214
Przykłady metod łańcuchów znaków — modyfikowanie	215
Przykłady metod łańcuchów znaków — analiza składniowa tekstu	218
Inne znane metody łańcuchów znaków w akcji	219
Oryginalny moduł string (usunięty w 3.0)	220
Wyrażenia formatujące łańcuchy znaków	221
Zaawansowane wyrażenia formatujące	222
Wyrażenia formatujące z użyciem słownika	224
Metoda format	225
Podstawy	225
Użycie kluczy, atrybutów i przesunięć	226
Formatowanie specjalizowane	227
Porównanie z wyrażeniami formatującymi	229
Po co nam kolejny mechanizm formatujący?	232
Generalne kategorie typów	235
Typy z jednej kategorii wspólną zbiory operacji	235
Typy zmienne można modyfikować w miejscu	236
Podsumowanie rozdziału	236
Sprawdź swoją wiedzę — quiz	236
Sprawdź swoją wiedzę — odpowiedzi	237
8. Listy oraz słowniki	239
Listy	239
Listy w akcji	241
Podstawowe operacje na listach	241
Iteracje po listach i składanie list	242
Indeksowanie, wycinki i macierze	243
Modyfikacja list w miejscu	244
Słowniki	248
Słowniki w akcji	249
Podstawowe operacje na słownikach	250
Modyfikacja słowników w miejscu	251
Inne metody słowników	252
Przykład z tabelą języków programowania	253
Uwagi na temat korzystania ze słowników	254
Inne sposoby tworzenia słowników	257
Zmiany dotyczące słowników w 3.0	258
Podsumowanie rozdziału	264
Sprawdź swoją wiedzę — quiz	264
Sprawdź swoją wiedzę — odpowiedzi	264

9. Krotki, pliki i pozostałe	267
Krotki	267
Krotki w akcji	268
Dlaczego istnieją listy i krotki?	271
Pliki	271
Otwieranie plików	272
Wykorzystywanie plików	273
Pliki w akcji	274
Inne narzędzia powiązane z plikami	280
Raz jeszcze o kategoriach typów	281
Elastyczność obiektów	282
Referencje a kopie	283
Porównania, równość i prawda	285
Porównywanie słowników w Pythonie 3.0	287
Znaczenie True i False w Pythonie	288
Hierarchie typów Pythona	290
Obiekty typów	291
Inne typy w Pythonie	291
Pułapki typów wbudowanych	292
Przypisanie tworzy referencje, nie kopie	292
Powtórzenie dodaje jeden poziom zagłębienia	293
Uwaga na cykliczne struktury danych	293
Typów niezmiennych nie można modyfikować w miejscu	294
Podsumowanie rozdziału	294
Sprawdź swoją wiedzę — quiz	294
Sprawdź swoją wiedzę — odpowiedzi	295
Sprawdź swoją wiedzę — ćwiczenia do części drugiej	295

Część III Instrukcje i składnia 299

10. Wprowadzenie do instrukcji Pythona	301
Raz jeszcze o strukturze programu Pythona	301
Instrukcje Pythona	301
Historia dwóch if	303
Co dodaje Python	304
Co usuwa Python	304
Skąd bierze się składnia indentacji?	306
Kilką przypadków specjalnych	308
Szybki przykład — interaktywne pętle	310
Prosta pętla interaktywna	310
Wykonywanie obliczeń na danych użytkownika	311
Obsługa błędów za pomocą sprawdzania danych wejściowych	312
Obsługa błędów za pomocą instrukcji try	313
Kod zagnieżdżony na trzy poziomy głębokości	314

Podsumowanie rozdziału	315
Sprawdź swoją wiedzę — quiz	315
Sprawdź swoją wiedzę — odpowiedzi	315
11. Przypisania, wyrażenia i wyświetlanie	317
Instrukcje przypisania	317
Formy instrukcji przypisania	318
Przypisanie sekwencji	319
Rozszerzona składnia rozpakowania sekwencji w 3.0	322
Przypisanie z wieloma celami	325
Przypisania rozszerzone	326
Reguły dotyczące nazw zmiennych	329
Instrukcje wyrażeń	332
Instrukcje wyrażeń i modyfikacje w miejscu	333
Polecenia print	334
Funkcja print Pythona 3.0	334
Instrukcja print w Pythonie 2.6	337
Przekierowanie strumienia wyjściowego	338
Wyświetlanie niezależne od wersji	341
Podsumowanie rozdziału	343
Sprawdź swoją wiedzę — quiz	344
Sprawdź swoją wiedzę — odpowiedzi	344
12. Testy if i reguły składni	345
Instrukcje if	345
Ogólny format	345
Proste przykłady	346
Rozgałęzienia kodu	346
Reguły składni Pythona	348
Ograniczniki bloków — reguły indentacji	349
Ograniczniki instrukcji — wiersze i kontynuacje	351
Kilka przypadków specjalnych	352
Testy prawdziwości	353
Wyrażenie trójargumentowe if/else	355
Podsumowanie rozdziału	356
Sprawdź swoją wiedzę — quiz	357
Sprawdź swoją wiedzę — odpowiedzi	358
13. Pętle while i for	359
Pętle while	359
Ogólny format	360
Przykłady	360
Instrukcje break, continue, pass oraz else w pętli	361
Ogólny format pętli	361
Instrukcja pass	361

Instrukcja continue	363
Instrukcja break	363
Instrukcja else	364
Pętle for	365
Ogólny format	365
Przykłady	367
Techniki tworzenia pętli	372
Pętle liczników — while i range	373
Przechodzenie niewyczerpujące — range i wycinki	374
Modyfikacja list — range	375
Przechodzenie równolegle — zip oraz map	376
Generowanie wartości przesunięcia i elementów — enumerate	379
Podsumowanie rozdziału	380
Sprawdź swoją wiedzę — quiz	380
Sprawdź swoją wiedzę — odpowiedzi	380
14. Iteracje i składanie list — część 1.	383
Pierwsze spojrzenie na iteratory	383
Protokół iteracyjny, iteratory plików	384
Kontrola iteracji — iter i next	386
Inne iteratory typów wbudowanych	388
Listy składane — wprowadzenie	390
Podstawy list składanych	390
Wykorzystywanie list składanych w plikach	391
Rozszerzona składnia list składanych	392
Inne konteksty iteracyjne	393
Nowe obiekty iterowane w Pythonie 3.0	397
Iterator range()	397
Iteratory map(), zip() i filter()	398
Kilka iteratorów na tym samym obiekcie	399
Iteratory widoku słownika	400
Inne zagadnienia związane z iteratorami	402
Podsumowanie rozdziału	402
Sprawdź swoją wiedzę — quiz	402
Sprawdź swoją wiedzę — odpowiedzi	403
15. Wprowadzenie do dokumentacji	405
Źródła dokumentacji Pythona	405
Komentarze ze znakami #	406
Funkcja dir	406
Łańcuchy znaków dokumentacji — __doc__	407
PyDoc — funkcja help	410
PyDoc — raporty HTML	412
Zbiór standardowej dokumentacji	415
Zasoby internetowe	415
Publikowane książki	416

Często spotykane problemy programistyczne	417
Podsumowanie rozdziału	419
Sprawdź swoją wiedzę — quiz	419
Sprawdź swoją wiedzę — odpowiedzi	419
Ćwiczenia do części trzeciej	420
Część IV Funkcje	423
16. Podstawy funkcji	425
Po co używa się funkcji?	426
Tworzenie funkcji	426
Instrukcje def	428
Instrukcja def uruchamiana jest w czasie wykonania	428
Pierwszy przykład — definicje i wywoływanie	429
Definicja	429
Wywołanie	430
Polimorfizm w Pythonie	430
Drugi przykład — przecinające się sekwencje	431
Definicja	432
Wywołania	432
Raz jeszcze o polimorfizmie	433
Zmienne lokalne	433
Podsumowanie rozdziału	434
Sprawdź swoją wiedzę — quiz	434
Sprawdź swoją wiedzę — odpowiedzi	434
17. Zakresy	437
Podstawy zakresów w Pythonie	437
Reguły dotyczące zakresów	438
Rozwiązywanie konfliktów w zakresie nazw — reguła LEGB	440
Przykład zakresu	441
Zakres wbudowany	442
Instrukcja global	443
Minimalizowanie stosowania zmiennych globalnych	445
Minimalizacja modyfikacji dokonywanych pomiędzy plikami	446
Inne metody dostępu do zmiennych globalnych	447
Zakresy a funkcje zagnieżdżone	448
Szczegóły dotyczące zakresów zagnieżdżonych	449
Przykład zakresu zagnieżdzonego	449
Instrukcja nonlocal	455
Podstawy instrukcji nonlocal	455
Instrukcja nonlocal w akcji	456
Czemu służą zmienne nielokalne?	458
Podsumowanie rozdziału	462

Sprawdź swoją wiedzę — quiz	462
Sprawdź swoją wiedzę — odpowiedzi	463
18. Argumenty	465
Podstawy przekazywania argumentów	465
Argumenty a współdzielone referencje	466
Unikanie modyfikacji zmiennych argumentów	468
Symulowanie parametrów wyjścia	469
Specjalne tryby dopasowania argumentów	470
Podstawy	470
Składnia dopasowania	471
Dopasowywanie argumentów — szczegóły	472
Przykłady ze słowami kluczowymi i wartościami domyślnymi	473
Przykład dowolnych argumentów	475
Argumenty mogące być tylko słowami kluczowymi z Pythona 3.0	479
Przykład z funkcją obliczającą minimum	482
Pełne rozwiążanie	483
Dodatkowy bonus	484
Puente	485
Uogólnione funkcje działające na zbiorach	485
Emulacja funkcji print z Pythona 3.0	486
Wykorzystywanie argumentów mogących być tylko słowami kluczowymi	487
Podsumowanie rozdziału	488
Sprawdź swoją wiedzę — quiz	489
Sprawdź swoją wiedzę — odpowiedzi	490
19. Zaawansowane zagadnienia dotyczące funkcji	491
Koncepcje projektowania funkcji	491
Funkcje rekurencyjne	493
Sumowanie z użyciem rekurencji	493
Implementacje alternatywne	494
Pętle a rekurencja	495
Obsługa dowolnych struktur	496
Obiekty funkcji — atrybuty i adnotacje	497
Pośrednie wywołania funkcji	497
Introspekcja funkcji	498
Atrybuty funkcji	499
Adnotacje funkcji w Pythonie 3.0	499
Funkcje anonimowe — lambda	501
Wyrażenia lambda	501
Po co używa się wyrażenia lambda?	503
Jak łatwo zaciemnić kod napisany w Pythonie	504
Zagnieżdżone wyrażenia lambda a zakresy	505
Odwzorowywanie funkcji na sekwencje — map	507
Narzędzia programowania funkcyjnego — filter i reduce	508

Podsumowanie rozdziału	510
Sprawdź swoją wiedzę — quiz	510
Sprawdź swoją wiedzę — odpowiedzi	510
20. Iteracje i składanie list — część 2	513
Listy składane, podejście drugie — narzędzia funkcyjne	513
Listy składane kontra map	514
Dodajemy warunki i pętle zagnieżdżone — filter	515
Listy składane i macierze	517
Zrozumieć listy składane	518
Iteratorów ciąg dalszy — generatory	520
Funkcje generatorów — yield kontra return	520
Wyrażenia generatorów — iteratory spotykają złożenia	524
Funkcje generatorów kontra wyrażenia generatorów	525
Generatory są jednorazowymi iteratorami	526
Emulacja funkcji zip i map za pomocą narzędzi iteracyjnych	527
Generowanie wyników we wbudowanych typach i klasach	531
Podsumowanie obiektów składanych w 3.0	533
Zrozumieć zbiory i słowniki składane	534
Rozszerzona składnia zbiorów i słowników składanych	534
Pomiary wydajności implementacji iteratorów	535
Moduł mytimer	536
Skrypt mierzący wydajność	536
Pomiary czasu	537
Alternatywne moduły mierzące wydajność	539
Inne sugestie	543
Pułapki związane z funkcjami	544
Lokalne nazwy są wykrywane w sposób statyczny	544
Wartości domyślne i obiekty mutowalne	546
Funkcje niezwracające wyników	548
Funkcje zagnieżdżone a zmienne modyfikowane w pętli	548
Podsumowanie rozdziału	548
Sprawdź swoją wiedzę — quiz	549
Sprawdź swoją wiedzę — odpowiedzi	549
Sprawdź swoją wiedzę — ćwiczenia do części czwartej	550
Część V Moduły	553
21. Moduły — wprowadzenie	555
Po co używa się modułów?	555
Architektura programu w Pythonie	556
Struktura programu	556
Importowanie i atrybuty	557
Moduły biblioteki standardowej	558

Jak działa importowanie	559
1. Odnalezienie modułu	560
2. (Ewentualne) Kompilowanie	560
3. Wykonanie	561
Ścieżka wyszukiwania modułów	561
Konfiguracja ścieżki wyszukiwania	563
Wariacje ścieżki wyszukiwania modułów	564
Lista sys.path	564
Wybór pliku modułu	565
Zaawansowane zagadnienia związane z wyborem modułów	566
Podsumowanie rozdziału	566
Sprawdź swoją wiedzę — quiz	567
Sprawdź swoją wiedzę — odpowiedzi	568
22. Podstawy tworzenia modułów	569
Tworzenie modułów	569
Użycie modułów	570
Instrukcja import	570
Instrukcja from	571
Instrukcja from *	571
Operacja importowania odbywa się tylko raz	571
Instrukcje import oraz from są przypisaniami	572
Modyfikacja zmiennych pomiędzy plikami	573
Równoważność instrukcji import oraz from	573
Potencjalne pułapki związane z użyciem instrukcji from	574
Przestrzenie nazw modułów	575
Pliki generują przestrzenie nazw	576
Kwalifikowanie nazw atrybutów	577
Importowanie a zakresy	578
Zagnieżdżanie przestrzeni nazw	579
Przeładowywianie modułów	580
Podstawy przeładowywania modułów	581
Przykład przeładowywania z użyciem reload	581
Podsumowanie rozdziału	582
Sprawdź swoją wiedzę — quiz	583
Sprawdź swoją wiedzę — odpowiedzi	584
23. Pakiety modułów	585
Podstawy importowania pakietów	585
Pakiety a ustawienia ścieżki wyszukiwania	586
Pliki pakietów __init__.py	586
Przykład importowania pakietu	588
Instrukcja from a instrukcja import w importowaniu pakietów	589
Do czego służy importowanie pakietów?	590
Historia trzech systemów	590

Względne importowanie pakietów	593
Zmiany w Pythonie 3.0	593
Podstawy importów względnych	594
Do czego służą importy względne?	595
Zakres importów względnych	597
Podsumowanie reguł wyszukiwania modułów	598
Importy względne w działaniu	598
Podsumowanie rozdziału	603
Sprawdź swoją wiedzę — quiz	604
Sprawdź swoją wiedzę — odpowiedzi	604
24. Zaawansowane zagadnienia związane z modułami	607
Ukrywanie danych w modułach	607
Minimalizacja niebezpieczeństw użycia from * — _X oraz __all__	608
Włączanie opcji z przyszłych wersji Pythona	608
Mieszane tryby użycia __name__ oraz __main__	609
Testy jednostkowe z wykorzystaniem __name__	610
Użycie argumentów wiersza polecień z __name__	611
Modyfikacja ścieżki wyszukiwania modułów	613
Rozszerzenie as dla instrukcji import oraz from	614
Moduły są obiektami — metaprogramy	615
Importowanie modułów za pomocą łańcucha znaków nazwy	617
Przechodnie przeładowywane modułów	618
Projektowanie modułów	621
Pułapki związane z modułami	622
W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie	622
Instrukcja from kopiuje nazwy, jednak łączy już nie	623
Instrukcja from * może zaciemnić znaczenie zmiennych	624
Funkcja reload może nie mieć wpływu na obiekty importowane za pomocą from	624
Funkcja reload i instrukcja from a testowanie interaktywne	625
Rekurencyjne importowanie za pomocą from może nie działać	626
Podsumowanie rozdziału	627
Sprawdź swoją wiedzę — quiz	627
Sprawdź swoją wiedzę — odpowiedzi	628
Sprawdź swoją wiedzę — ćwiczenia do części piątej	628
Część VI Klasy i programowanie zorientowane obiektowo	631
25. Programowanie zorientowane obiektowo	633
Po co używa się klas?	634
Programowanie zorientowane obiektowo z dystansu	635
Wyszukiwanie dziedziczenia atrybutów	635
Klasy a instancje	637

Wywołania metod klasy	638
Tworzenie drzew klas	638
Programowanie zorientowane obiektowo oparte jest na ponownym wykorzystaniu kodu	641
Podsumowanie rozdziału	643
Sprawdź swoją wiedzę — quiz	644
Sprawdź swoją wiedzę — odpowiedzi	644
26. Podstawy tworzenia klas	647
Klasy generują większą liczbę obiektów instancji	647
Obiekty klas udostępniają zachowania domyślne	648
Obiekty instancji są rzeczywistymi elementami	648
Pierwszy przykład	649
Klasy dostosowuje się do własnych potrzeb przez dziedziczenie	651
Drugi przykład	652
Klasy są atrybutami w modułach	653
Klasy mogą przechwytywać operatory Pythona	654
Trzeci przykład	655
Po co przeciąża się operatory?	657
Najprostsza klasa Pythona na świecie	658
Klasy a słowniki	660
Podsumowanie rozdziału	662
Sprawdź swoją wiedzę — quiz	662
Sprawdź swoją wiedzę — odpowiedzi	663
27. Bardziej realistyczny przykład	665
Krok 1. — tworzenie instancji	666
Tworzenie konstruktorów	666
Testowanie w miarę pracy	667
Wykorzystywanie kodu na dwa sposoby	668
Krok 2. — dodawanie metod	669
Tworzenie kodu metod	671
Krok 3. — przeciążanie operatorów	673
Udostępnienie wyświetlania	674
Krok 4. — dostosowanie zachowania do własnych potrzeb za pomocą klas podrzędnych	675
Tworzenie klas podrzędnych	675
Rozszerzanie metod — niewłaściwy sposób	676
Rozszerzanie metod — właściwy sposób	676
Polimorfizm w akcji	678
Dziedziczenie, dostosowanie do własnych potrzeb i rozszerzenie	679
Programowanie zorientowane obiektowo — idea	680
Krok 5. — dostosowanie do własnych potrzeb także konstruktorów	680
Programowanie zorientowane obiektowo jest prostsze, niż się wydaje	682
Inne sposoby łączenia klas	683

Krok 6. — wykorzystywanie narzędzi do introspekcji	684
Specjalne atrybuty klas	686
Uniwersalne narzędzie do wyświetlania	687
Atrybuty instancji a atrybuty klas	688
Rozważania na temat nazw w klasach narzędzi	689
Ostateczna postać naszych klas	690
Krok 7. i ostatni — przechowanie obiektów w bazie danych	691
Obiekty pickle i shelve	691
Przechowywanie obiektów w bazie danych za pomocą shelve	692
Interaktywne badanie obiektów shelve	694
Aktualnianie obiektów w pliku shelve	695
Przyszłe kierunki rozwoju	697
Podsumowanie rozdziału	699
Sprawdź swoją wiedzę — quiz	699
Sprawdź swoją wiedzę — odpowiedzi	700
28. Szczegóły kodu klas	703
Instrukcja class	703
Ogólna forma	703
Przykład	704
Metody	706
Przykład metody	707
Wywoływanie konstruktorów klas nadrzędnych	708
Inne możliwości wywoływania metod	708
Dziedziczenie	708
Tworzenie drzewa atrybutów	709
Specjalizacja odziedziczonych metod	710
Techniki interfejsów klas	711
Abstrakcyjne klasy nadrzędne	712
Abstrakcyjne klasy nadrzędne z Pythona 2.6 oraz 3.0	713
Przestrzenie nazw — cała historia	714
Pojedyncze nazwy — globalne, o ile nie przypisane	715
Nazwy atrybutów — przestrzenie nazw obiektów	715
Zen przestrzeni nazw Pythona — przypisania klasyfikują zmienne	715
Słowniki przestrzeni nazw	718
Łącza przestrzeni nazw	720
Raz jeszcze o łańcuchach znaków dokumentacji	722
Klasy a moduły	723
Podsumowanie rozdziału	724
Sprawdź swoją wiedzę — quiz	724
Sprawdź swoją wiedzę — odpowiedzi	724

29. Przeciążanie operatorów .	727
Podstawy	727
Konstruktory i wyrażenia — <code>_init_</code> i <code>_sub_</code>	728
Często spotykane metody przeciążania operatorów	728
Indeksowanie i wycinanie — <code>_getitem_</code> i <code>_setitem_</code>	730
Wycinki	730
Iteracja po indeksie — <code>_getitem_</code>	731
Obiekty iteratorów — <code>_iter_</code> i <code>_next_</code>	733
Iteratory zdefiniowane przez użytkownika	734
Wiele iteracji po jednym obiekcie	735
Test przynależności — <code>_contains_</code> , <code>_iter_</code> i <code>_getitem_</code>	737
Metody <code>_getattr_</code> oraz <code>_setattr_</code> przechwytyują referencje do atrybutów	740
Inne narzędzia do zarządzania atrybutami	741
Emulowanie prywatności w atrybutach instancji	741
Metody <code>_repr_</code> oraz <code>_str_</code> zwracają reprezentacje łańcuchów znaków	742
Metoda <code>_radd_</code> obsługuje dodawanie prawostronne i modyfikację w miejscu	745
Dodawanie w miejscu	746
Metoda <code>_call_</code> przechwytuje wywołania	747
Interfejsy funkcji i kod oparty na wywołaniach zwrotnych	748
Porównania — <code>_lt_</code> , <code>_gt_</code> i inne	750
Metoda <code>_cmp_</code> w 2.6 (usunięta w 3.0)	750
Testy logiczne — <code>_bool_</code> i <code>_len_</code>	751
Destrukcja obiektu — <code>_del_</code>	752
Podsumowanie rozdziału	754
Sprawdź swoją wiedzę — quiz	755
Sprawdź swoją wiedzę — odpowiedzi	755
30. Projektowanie z użyciem klas .	757
Python a programowanie zorientowane obiektowo	757
Przeciążanie za pomocą sygnatur wywołań (lub bez nich)	758
Programowanie zorientowane obiektowo i dziedziczenie — związek „jest”	759
Programowanie zorientowane obiektowo i kompozycja — związki typu „ma”	760
Raz jeszcze procesor strumienia danych	762
Programowanie zorientowane obiektowo a delegacja — obiekty „opakowujące”	765
Pseudoprivatne atrybuty klas	767
Przegląd zniekształcania nazw zmiennych	767
Po co używa się atrybutów pseudoprivatnych?	768
Metody są obiektami — z wiązaniem i bez wiązania	770
Metody niezwiązane w 3.0	772
Metody związane i inne obiekty wywoływane	773
Dziedziczenie wielokrotne — klasy mieszane	775
Tworzenie klas mieszanych	776
Klasy są obiektami — uniwersalne fabryki obiektów	785
Do czego służą fabryki?	787

Inne zagadnienia związane z projektowaniem	788
Podsumowanie rozdziału	788
Sprawdź swoją wiedzę — quiz	789
Sprawdź swoją wiedzę — odpowiedzi	789
31. Zaawansowane zagadnienia związane z klasami	791
Rozszerzanie typów wbudowanych	791
Rozszerzanie typów za pomocą osadzania	792
Rozszerzanie typów za pomocą klas podrzędnych	793
Klasy w nowym stylu	795
Nowości w klasach w nowym stylu	796
Zmiany w modelu typów	797
Zmiany w dziedziczeniu diamentowym	801
Nowości w klasach w nowym stylu	805
Sloty	805
Właściwości klas	809
Przeciążanie nazw — <code>__getattribute__</code> i deskryptory	811
Metaklasy	811
Metody statyczne oraz metody klasy	811
Do czego potrzebujemy metod specjalnych?	812
Metody statyczne w 2.6 i 3.0	812
Alternatywy dla metod statycznych	814
Używanie metod statycznych i metod klas	815
Zliczanie instancji z użyciem metod statycznych	817
Zliczanie instancji z metodami klas	818
Dekoratory i metaklasy — część 1.	820
Podstawowe informacje o dekoratorach funkcji	820
Przykład dekoratora	821
Dekoratory klas i metaklasy	822
Dalsza lektura	823
Pułapki związane z klasami	824
Modyfikacja atrybutów klas może mieć efekty uboczne	824
Modyfikowanie mutowalnych atrybutów klas również może mieć efekty uboczne	825
Dziedziczenie wielokrotne — kolejność ma znaczenie	826
Metody, klasy oraz zakresy zagnieżdżone	827
Klasy wykorzystujące delegację w 3.0 — <code>__getattr__</code> i funkcje wbudowane	829
Przesadne opakowywanie	829
Podsumowanie rozdziału	830
Sprawdź swoją wiedzę — quiz	830
Sprawdź swoją wiedzę — odpowiedzi	830
Sprawdź swoją wiedzę — ćwiczenia do części szóstej	831

Część VII Wyjątki oraz narzędzia	839
32. Podstawy wyjątków	841
Po co używa się wyjątków?	841
Role wyjątków	842
Wyjątki w skrócie	843
Domyślny program obsługi wyjątków	843
Przechwytywanie wyjątków	844
Zgłaszanie wyjątków	845
Wyjątki zdefiniowane przez użytkownika	845
Działania końcowe	846
Podsumowanie rozdziału	847
Sprawdź swoją wiedzę — quiz	849
Sprawdź swoją wiedzę — odpowiedzi	849
33. Szczegółowe informacje dotyczące wyjątków	851
Instrukcja try/except/else	851
Części instrukcji try	853
Część try/else	855
Przykład — zachowanie domyślne	856
Przykład — przechwytywanie wbudowanych wyjątków	857
Instrukcja try/finally	857
Przykład — działania kończące kod z użyciem try/finally	858
Połączona instrukcja try/except/finally	859
Składnia połączonej instrukcji try	860
Łączenie finally oraz except za pomocą zagnieżdżania	861
Przykład połączonego try	862
Instrukcja raise	863
Przekazywanie wyjątków za pomocą raise	864
Łańcuchy wyjątków w Pythonie 3.0 — raise from	865
Instrukcja assert	865
Przykład — wyłapywanie ograniczeń (ale nie błędów!)	866
Menedżery kontekstu with/as	867
Podstawowe zastosowanie	867
Protokół zarządzania kontekstem	868
Podsumowanie rozdziału	870
Sprawdź swoją wiedzę — quiz	871
Sprawdź swoją wiedzę — odpowiedzi	871
34. Obiekty wyjątków	873
Wyjątki — powrót do przeszłości	874
Wyjątki oparte na łańcuchach znaków znikają	874
Wyjątki oparte na klasach	875
Tworzenie klas wyjątków	875

Do czego służą hierarchie wyjątków?	877
Wbudowane klasy wyjątków	880
Kategorie wbudowanych wyjątków	881
Domyślne wyświetlanie oraz stan	882
Własne sposoby wyświetlania	883
Własne dane oraz zachowania	884
Udostępnianie szczegółów wyjątku	884
Udostępnianie metod wyjątków	885
Podsumowanie rozdziału	886
Sprawdź swoją wiedzę — quiz	886
Sprawdź swoją wiedzę — odpowiedzi	886
35. Projektowanie z wykorzystaniem wyjątków	889
Zagnieżdżanie programów obsługi wyjątków	889
Przykład — zagnieżdżanie przebiegu sterowania	891
Przykład — zagnieżdżanie składniowe	891
Zastosowanie wyjątków	893
Wyjątki nie zawsze są błędami	893
Funkcje mogą sygnalizować warunki za pomocą raise	893
Zamykanie plików oraz połączeń z serwerem	894
Debugowanie z wykorzystaniem zewnętrznych instrukcji try	895
Testowanie kodu wewnętrz tego samego procesu	895
Więcej informacji na temat funkcji sys.exc_info	896
Wskazówki i pułapki dotyczące projektowania wyjątków	897
Co powinniśmy opakować w try	897
Jak nie przechwytywać zbyt wiele — unikanie pustych except i wyjątków	898
Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach	900
Podsumowanie jądra języka Python	901
Zbiór narzędzi Pythona	901
Narzędzia programistyczne przeznaczone do większych projektów	902
Podsumowanie rozdziału	906
Sprawdź swoją wiedzę — quiz	906
Sprawdź swoją wiedzę — odpowiedzi	906
Sprawdź swoją wiedzę — ćwiczenia do części siódmej	907
Część VIII Zagadnienia zaawansowane	909
36. Łańcuchy znaków Unicode oraz łańcuchy bajtowe	911
Zmiany w łańcuchach znaków w Pythonie 3.0	912
Podstawy łańcuchów znaków	913
Kodowanie znaków	913
Typy łańcuchów znaków Pythona	915
Pliki binarne i tekstowe	916

Łańcuchy znaków Pythona 3.0 w akcji	918
Literały i podstawowe właściwości	918
Konwersje	919
Kod łańcuchów znaków Unicode	920
Kod tekstu z zakresu ASCII	921
Kod tekstu spoza zakresu ASCII	921
Kodowanie i dekodowanie tekstu spoza zakresu ASCII	922
Inne techniki kodowania łańcuchów Unicode	923
Konwersja kodowania	925
Łańcuchy znaków Unicode w Pythonie 2.6	925
Deklaracje typu kodowania znaków pliku źródłowego	928
Wykorzystywanie obiektów bytes z Pythona 3.0	929
Wywołania metod	929
Operacje na sekwencjach	930
Inne sposoby tworzenia obiektów bytes	931
Mieszanie typów łańcuchów znaków	931
Wykorzystywanie obiektów bytearray z Pythona 3.0 (i 2.6)	932
Wykorzystywanie plików tekstowych i binarnych	935
Podstawy plików tekstowych	935
Tryby tekstowy i binarny w Pythonie 3.0	936
Brak dopasowania typu i zawartości	938
Wykorzystywanie plików Unicode	939
Odczyt i zapis Unicode w Pythonie 3.0	939
Obsługa BOM w Pythonie 3.0	941
Pliki Unicode w Pythonie 2.6	943
Inne zmiany narzędzi łańcuchów znaków w Pythonie 3.0	944
Moduł dopasowywania wzorców re	944
Moduł danych binarnych struct	945
Moduł serializacji obiektów pickle	947
Narzędzia do analizy składniowej XML	948
Podsumowanie rozdziału	951
Sprawdź swoją wiedzę — quiz	952
Sprawdź swoją wiedzę — odpowiedzi	952

37. Zarządzane atrybuty	955
Po co zarządza się atrybutami?	955
Wstawianie kodu wykonywanego w momencie dostępu do atrybutów	956
Właściwości	957
Podstawy	957
Pierwszy przykład	958
Obliczanie atrybutów	959
Zapisywanie właściwości w kodzie za pomocą dekoratorów	960
Deskryptory	961
Podstawy	962
Pierwszy przykład	964
Obliczone atrybuty	966

Wykorzystywanie informacji o stanie w deskryptorach	967
Powiązania pomiędzy właściwościami a deskryptorami	968
Metody <code>__getattr__</code> oraz <code>__getattribute__</code>	970
Podstawy	971
Pierwszy przykład	973
Obliczanie atrybutów	974
Porównanie metod <code>__getattr__</code> oraz <code>__getattribute__</code>	975
Porównanie technik zarządzania atrybutami	976
Przechwytywanie atrybutów wbudowanych operacji	979
Powrót do menedżerów opartych na delegacji	983
Przykład — sprawdzanie poprawności atrybutów	986
Wykorzystywanie właściwości do sprawdzania poprawności	986
Wykorzystywanie deskryptorów do sprawdzania poprawności	988
Wykorzystywanie metody <code>__getattribute__</code> do sprawdzania poprawności	990
Wykorzystywanie metody <code>__getattribute__</code> do sprawdzania poprawności	991
Podsumowanie rozdziału	992
Sprawdź swoją wiedzę — quiz	992
Sprawdź swoją wiedzę — odpowiedzi	993
38. Dekoratory	995
Czym jest dekorator?	995
Zarządzanie wywołaniami oraz instancjami	996
Zarządzanie funkcjami oraz klasami	996
Wykorzystywanie i definiowanie dekoratorów	997
Do czego służą dekoratory?	997
Podstawy	998
Dekoratory funkcji	998
Dekoratory klas	1002
Zagnieżdżanie dekoratorów	1004
Argumenty dekoratorów	1006
Dekoratory zarządzają także funkcjami oraz klasami	1006
Kod dekoratorów funkcji	1007
Śledzenie wywołań	1007
Możliwości w zakresie zachowania informacji o stanie	1009
Uwagi na temat klas I — dekorowanie metod klas	1012
Mierzenie czasu wywołania	1017
Dodawanie argumentów dekoratora	1019
Kod dekoratorów klas	1021
Klasy singletona	1021
Śledzenie interfejsów obiektów	1023
Uwagi na temat klas II — zachowanie większej liczby instancji	1027
Dekoratory a funkcje zarządzające	1028
Do czego służą dekoratory? (raz jeszcze)	1029
Bezpośrednie zarządzanie funkcjami oraz klasami	1031
Przykład — atrybuty „prywatne” i „publiczne”	1033
Implementacja atrybutów prywatnych	1033

Szczegóły implementacji I	1035
Uogólnienie kodu pod kątem deklaracji atrybutów jako publicznych	1036
Szczegóły implementacji II	1038
Znane problemy	1039
W Pythonie nie chodzi o kontrolę	1043
Przykład: Sprawdzanie poprawności argumentów funkcji	1044
Cel	1044
Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych	1045
Uogólnienie kodu pod kątem słów kluczowych i wartości domyślnych	1047
Szczegóły implementacji	1050
Znane problemy	1052
Argumenty dekoratora a anotacje funkcji	1053
Inne zastosowania — sprawdzanie typów (skoro nalegamy!)	1054
Podsumowanie rozdziału	1055
Sprawdź swoją wiedzę — quiz	1056
Sprawdź swoją wiedzę — odpowiedzi	1056
39. Metaklasy	1061
Tworzyć metaklasy czy tego nie robić?	1061
Zwiększaające się poziomy magii	1062
Wady funkcji pomocniczych	1064
Metaklasy a dekoratory klas — runda 1.	1066
Model metaklasy	1068
Klasy są instancjami obiektu type	1068
Metaklasy są klasami podlegającymi klasy type	1070
Protokół instrukcji class	1071
Deklarowanie metaklas	1071
Tworzenie metaklas	1073
Prosta metaklasa	1073
Dostosowywanie tworzenia do własnych potrzeb oraz inicjalizacja	1074
Pozostałe sposoby tworzenia metaklas	1074
Instancje a dziedziczenie	1077
Przykład — dodawanie metod do klas	1078
Ręczne rozszerzanie	1078
Rozszerzanie oparte na metaklasie	1080
Metaklasy a dekoratory klas — runda 2.	1081
Przykład — zastosowanie dekoratorów do metod	1084
Ręczne śledzenie za pomocą dekoracji	1084
Śledzenie metaklas oraz dekoratorów	1085
Zastosowanie dowolnego dekoratora do metod	1086
Metaklasy a dekoratory klas — runda 3.	1088
Podsumowanie rozdziału	1091
Sprawdź swoją wiedzę — quiz	1092
Sprawdź swoją wiedzę — odpowiedzi	1092

Dodatki	1093
Dodatek A Instalacja i konfiguracja	1095
Instalowanie interpretera Pythona	1095
Czy Python jest już zainstalowany?	1095
Skąd pobrać Pythona	1096
Instalacja Pythona	1097
Konfiguracja Pythona	1098
Zmienne środowiskowe Pythona	1098
Jak ustawić opcje konfiguracyjne?	1100
Opcje wiersza poleceń Pythona	1103
Uzyskanie pomocy	1104
Dodatek B Rozwiązania ćwiczeń podsumowujących poszczególne części książki	1105
Część I Wprowadzenie	1105
Część II Typy i operacje	1107
Część III Instrukcja i składnia	1112
Część IV Funkcje	1114
Część V Moduły	1121
Część VI Klasy i programowanie zorientowane obiektowo	1125
Część VII Wyjątki oraz narzędzia	1132
Skorowidz	1139

Przedmowa

Niniejsza książka stanowi wprowadzenie do języka Python. *Python* to popularny język programowania z dziedziny open source, wykorzystywany w wielu różnych zastosowaniach zarówno w samodzielnnych programach, jak i skryptach. Jest darmowy, łatwo przenośny, ma duże możliwości i jest niesamowicie prosty w użyciu. Programiści z każdego zakątku branży tworzenia oprogramowania uznają, że koncentracja Pythona na wydajności programisty i jakości oprogramowania daje strategiczną przewagę w dużych i małych projektach.

Bez względu na to, czy jest się doświadczonym programistą, czy też osobą poczynającą, celem niniejszej książki jest omówienie podstaw tego języka programowania. Po lekturze książki każdy powinien być w stanie wykorzystywać Pythona w wielu różnych zastosowaniach.

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

- *Learning Python* (w Polsce: *Python. Wprowadzenie*) — niniejsza książka — naucza o samym Pythonie.
- *Programming Python* (i wiele innych) — pokazuje, co można zrobić z Pythonem po nauczaniu się go.

Oznacza to, że książki poświęcone dziedzinie aplikacji, takie jak *Programming Python*, kontynuują to, na czym kończy się niniejsza książka, omawiając rolę Pythona w popularnych zastosowaniach, takich jak Internet, graficzne interfejsy użytkownika (GUI) oraz bazy danych. Dodatkowo książka *Python. Leksykon kieszonkowy* udostępnia materiały nieujęte tutaj i zaprojektowana jest w taki sposób, by uzupełniać niniejszą pozycję.

Z powodu skupienia się w tej książce na podstawach możemy zaprezentować jądro języka Python z głębią przekraczającą to, co wielu programistów widzi przy pierwszym podejściu do nauki języka programowania. A ponieważ książka oparta jest na trzydniowym kursie szkoleniowym z wieloma quizami i ćwiczeniami, pozwala ona na samodzielne zapoznanie się z Pythonem we własnym tempie.

O niniejszym, czwartym wydaniu książki

Czwarte wydanie książki zmieniło się na trzy różne sposoby. Wydanie to:

- Omawia zarówno Pythona 3.0, jak i Pythona 2.6 — nacisk położony jest na wersję 3.0, ale podkreślone zostały różnice w wersji 2.6.
- Zawiera kilka nowych rozdziałów poświęconych przede wszystkim zaawansowanym zagadnieniom związanym z jądrem tego języka programowania.
- Dokonuje reorganizacji części istniejącego materiału i dla większej jasności rozszerza go za pomocą nowych przykładów.

Kiedy piszę to wydanie książki, w roku 2009, Python ma dwie odmiany — wersja 3.0 jest nowym i niezgodnym z poprzednimi wydaniem języka, natomiast wersja 2.6 zachowuje zgodność z większością kodu napisanego w tym języku. Choć Python 3 uznawany jest za przyszłość, Python 2 nadal jest w szerokim użyciu i będzie w najbliższych latach obsługiwany równolegle z wersją 3. Pomimo że Python 3.0 jest w dużej mierze tym samym językiem, nie jest w stanie wykonać prawie żadnego kodu przeznaczonego dla wcześniejszych wersji (sama zmiana `print` z instrukcji na funkcję, choć estetycznie bardzo rozsądna, sprawia, że prawie żaden napisany wcześniej w Pythonie program nie będzie działał).

Ten podział stanowi dylemat zarówno dla programistów, jak i autorów książek. Choć łatwiej byłoby udawać, że Python 2 nigdy nie istniał, i omówić tylko wersję 3, takie podejście nie sprostałoby potrzebom ogromnej liczby dzisiejszych użytkowników Pythona. Imponująca ilość istniejącego kodu napisana została w Pythonie 2 i nie zniknie zbyt szybko. I choć osoby poczynające mogą się skupić na Pythonie 3, każdy, kto ma do czynienia z kodem napisanym w przeszłości, musi jedną nogą stać w świecie Pythona 2. Ponieważ przeniesienie wszystkich zewnętrznych bibliotek i rozszerzeń na nową wersję może zajść lata, taki podział na dwa nurty może nie być całkiem tymczasowy.

Omówienie zarówno Pythona 3.0, jak i 2.6

By odpowiedzieć na tę dydaktyczną i sprostać potrzebom wszystkich potencjalnych użytkowników, niniejsze wydanie książki zostało uaktualnione w taki sposób, by omawiać zarówno Pythona 3.0, jak i Pythona 2.6 (a także późniejsze wydania z gałęzi 3.X oraz 2.X). Książka przeznaczona jest dla programistów wykorzystujących Pythona 2, programistów używających Pythona 3 i programistów, którzy utknęli gdzieś pomiędzy.

Oznacza to, że można wykorzystać niniejszą pozycję do opanowania obu gałęzi Pythona. Choć skupiamy się przede wszystkim na wersji 3.0, różnice w stosunku do Pythona 2.6 i narzędzia z tej wersji są po drodze opisywane z przeznaczeniem dla programistów korzystających ze starszego kodu. Choć obie wersje są w dużej mierze podobne, różnią się na kilka istotnych sposobów, co będę podkreślał w miarę omawiania jądra języka.

Przykładowo w większości kodów będę wykorzystywał wywołanie `print` z Pythona 3.0, jednak opiszę także instrukcję `print` z wersji 2.6, tak by łatwo było zrozumieć starszy kod. Będę także swobodnie wprowadzał nowe opcje tego języka programowania, takie jak instrukcja `nonlocal` z wersji 3.0 czy metoda łańcuchów znaków `format` z 2.6 i 3.0, a także wskazywał, kiedy rozszerzenia te są nieobecne w jeszcze wcześniejszych wersjach.

Osoby uczące się Pythona od podstaw, które nie muszą korzystać ze starszego kodu, zachęcam do rozpoczęcia nauki od razu od Pythona 3.0. Wersja ta usuwa sporo cech, które można określić jako wrzody nagromadzone przez lata w tym języku, zachowując jednak wszystkie oryginalne idee leżące u jego podstaw i dodając sporo ciekawych nowych narzędzi.

Wiele z popularnych bibliotek oraz narzędzi Pythona będzie raczej w momencie czytania tej książki współdziałało z Pythonem 3.0, zwłaszcza biorąc pod uwagę poprawę wydajności wejścia-wyjścia plików oczekiwana w wydaniu 3.1. Osoby korzystające z systemu opartego na Pythonie 2.X przekonają się jednak, że niniejsza książka odpowiada także ich potrzebom i ułatwi przejście w przyszłości na wersję 3.0.

W sposób pośredni niniejsze wydanie książki opisuje także inne wersje Pythona 2 oraz 3, choć część starszego kodu z wydań 2.X może nie być w stanie wykonać wszystkich zaprezentowanych tutaj przykładów. Przykładowo choć dekoratory klas dostępne są w Pythonie 2.6 oraz 3.0, nie można z nich korzystać w starszych wydaniach Pythona 2.X, które nie obsługują tej opcji. W tabelach P.1 oraz P.2 w dalszej części rozdziału znajduje się podsumowanie zmian z Pythona 2.6 oraz 3.0.



Tuż przed oddaniem do druku niniejsza książka została uzupełniona o wzmianki dotyczące ważniejszych rozszerzeń nowej wersji Pythona o numerze 3.1 — separatorów w formie przecinków i automatycznego numerowania pól w wywołaniach metody `format`, składni większej liczby menedżerów kontekstu z instrukcjami `with` czy nowych metod obiektów liczb. Ponieważ Python 3.1 ma na celu przede wszystkim optymalizację, niniejsza książka w sposób bezpośredni odnosi się również do tej wersji. Tak naprawdę, ponieważ Python 3.1 jest następcą Pythona 3.0, a największy Python jest zazwyczaj jednocześnie najlepszym, w niniejszej książce nazwa „Python 3.0” odnosi się do odmiany tego języka wprowadzonej z wersją 3.0 i kontynuowanej w całej gałęzi 3.X.

Nowe rozdziały

Choć głównym celem niniejszego wydania książki jest uaktualnienie przykładów oraz materiału z poprzedniego wydania, tak by działały one w wersjach 3.0 oraz 2.6, dodałem również pięć nowych rozdziałów w celu omówienia nowych zagadnień i rozszerzenia kontekstu:

- Rozdział 27. to nowy materiał dotyczący klas, wykorzystujący bardziej realistyczny przykład w celu zbadania podstaw programowania zorientowanego obiektowo w Pythonie.
- Rozdział 36. przedstawia szczegółowe informacje na temat Unicode iłańcuchów bajtowych, a także zarysowuje najważniejsze różnice między modelami łańcuchów znaków oraz plików w wersjach 3.0 i 2.6.
- Rozdział 37. gromadzi narzędzia do zarządzania atrybutami, takie jak właściwości, i udostępnia nowe omówienie deskryptorów.
- Rozdział 38. prezentuje dekoratory funkcji i klas, a także omawia wyczerpujące przykłady.
- Rozdział 39. przedstawia metaklasy, a także porównuje je i kontrastuje z dekoratorami.

Pierwszy z powyższych rozdziałów udostępnia stopniowe, rozwijane krok po kroku przykłady dotyczące wykorzystywania klas oraz programowania zorientowanego obiektowo w Pythonie. Oparty jest na demonstracji wykorzystywanej przeze mnie na kursach, na których wykładałem, przystosowanej jednak do użycia w książce. Rozdział ten ma zaprezentować programowanie

zorientowane obiektowo w kontekście bardziej realistycznym niż wcześniejsze przykłady, a także ilustrować połączenie różnych koncepcji związanych z klasami w większych, działających programach. Mam nadzieję, że w książce będzie działał równie dobrze jak w prowadzonych na żywo kursach.

Ostatnie cztery z nowych rozdziałów ujęte zostały w nową, ostatnią część książki, zatytułowaną „Zagadnienia zaawansowane”. Choć są to zagadnienia składające się na jądro języka, nie każdy programista Pythona musi zagłębiać się w szczegóły tekstu Unicode czy metaklas. Z tego powodu te cztery rozdziały zostały wydzielone do nowej części i oficjalnie są *lekturą opcjonalną*. Szczegółowe informacje na temat łańcuchów Unicode i danych binarnych zostały na przykład przeniesione do tej ostatniej części książki, ponieważ duża część programistów korzysta z prostych łańcuchów znaków ASCII i nie musi się na nich znać. W podobny sposób dekoratory i metaklasy są zagadnieniami specjalistycznymi, które zazwyczaj są bardziej interesujące dla twórców API niż programistów aplikacji.

Jeśli jednak korzystamy z tych narzędzi lub kodu, który je wykorzystuje, nowe rozdziały poświęcone zagadnieniom zaawansowanym powinny nam pomóc opanować podstawy. Dodatkowo przykłady z tych rozdziałów zawierają studia przypadków, które łączą w sobie różne koncepcje związane z jądrem języka Python; są one też bardziej rozbudowane od przykładów z pozostałą częścią książki. Ponieważ nowa część jest lekturą opcjonalną, zawiera quizy podsugujące rozdziały, jednak nie ma tam ćwiczeń końcowych.

Zmiany w istniejącym materiale

Dodatkowo część materiału z poprzedniego wydania została *przeorganizowana* lub rozszerzona o *nowe przykłady*. W rozdziale 30. dziedziczenie wielokrotne otrzymało na przykład nowe studium przypadku wymieniające drzewa klas. W rozdziale 20. znalazły się nowe przykłady generatorów w ręczny sposób implementujących funkcje `map` i `zip`. Nowy kod ilustruje także metody statyczne i klas z rozdziału 31., a względne importowanie pakietów pokazane zostało na przykładzie w rozdziale 23. Metody przeciążania operatorów `__contains__`, `__bool__` oraz `__index__` są teraz w rozdziale 29. opatrzone przykładem, podobnie jak nowe protokoły przeciążania dla wycinków i porównań.

W niniejszym wydaniu przeprowadzono także pewną reorganizację tekstu dla zwiększenia jego jasności. Przykładowo w celu dołączenia nowego materiału i zagadnień, a także uniknięcia przeładowania rozdziałów — pięć poprzednich rozdziałów zostało każdorazowo podzielonych na dwa. W wyniku takiego działania otrzymaliśmy nowe, samodzielne rozdziały poświęcone przeciążaniu operatorów, zakresom i argumentom, szczegółom instrukcji wyjątków, a także złożeniom i iteracji. Niektóre istniejące rozdziały zostały poddane reorganizacji w celu poprawienia ich czytelności.

Dzięki temu w niniejszym wydaniu staramy się także minimalizować odwołania do przeszłych rozdziałów, choć zmiany z Pythona 3.0 sprawiają, że w niektórych przypadkach jest to niemożliwe. By zrozumieć operacje wyświetlania oraz metodę łańcuchów znaków `format`, trzeba na przykład wiedzieć coś o argumentach ze słowami kluczowymi w funkcjach. By opanować listy kluczy słowników oraz testy kluczy, trzeba wiedzieć coś o iteracji. Z kolei by wykorzystać funkcję `exec` do wykonania kodu, niezbędna jest umiejętność używania obiektów plików. Czytanie liniowe, od początku do końca, nadal wydaje się najbardziej rozsądną opcję, jednak niektóre zagadnienia mogą wymagać nielinowego przeskakiwania tam i z powrotem w tekście książki, a także zaglądania do odwołań.

W sumie w tym wydaniu książki znajdują się setki zmian. Tabele zamieszczone poniżej dokumentują 27 dodatkowych elementów i 57 zmian w Pythonie. Tak naprawdę można bezpiecznie powiedzieć, że niniejsze wydanie książki jest nieco bardziej zaawansowane, ponieważ sam Python taki się stał. Jeśli jednak chodzi o wersję 3.0, lepiej będzie chyba przeczytać o wszystkich tych zmianach samemu, w tekście książki, zamiast pobieżnie zapoznawać się z nimi w przedmowie do niej.

Niektóre zmiany języka Python w wersjach 2.6 oraz 3.0

Generalnie Python 3.0 jest językiem *czystszym*, jednak w pewien sposób także bardziej *wyszukanym*. Tak naprawdę niektóre z wprowadzonych zmian zdają się zakładać, że by nauczyć się Pythona, trzeba już znać Pythona. W poprzednim podrozdziale zarysowaliśmy kilka bardziej znaczących cyklicznych zależności, jeśli chodzi o wiedzę na temat Pythona 3.0. Pierwszym z brzegu przykładem będzie uzasadnienie opakowywania widoków słownika w wywołanie `list`, które jest dość subtelne i wymaga znacznej wiedzy kontekstowej. Poza uczeniem podstaw Pythona niniejsza książka służy do połączenia ze sobą tych odrębnych elementów.

W tabeli P.1 wymieniono najważniejsze nowe opcje języka omówione w niniejszym wydaniu książki, wraz z odwołaniem do najważniejszych rozdziałów, w których są one przedstawione.

Niektóre elementy języka usunięte w Pythonie 3.0

Poza dodatkowymi opcjami w wersji 3.0 usunięto także sporą liczbę narzędzi w celu oczyszczenia jego projektu. W tabeli P.2 podsumowano zmiany, które wpłynęły na niniejsze wydanie książki, omówione w poszczególnych jej rozdziałach. Wiele z elementów usuniętych i wymienionych w tabeli P.2 ma bezpośrednie następcy; część z nich dostępna jest również w Pythonie 2.6 w celu obsłużenia przyszłej migracji do wersji 3.0.

W Pythonie 3.0 znajdują się dodatkowe zmiany, które nie zostały ujęte w tabeli tylko dlatego, że nie wpłynęły na niniejszą książkę. Modyfikacje w bibliotece standardowej mogą na przykład mieć większy wpływ na książki poświęcone dziedzinie aplikacji, takie jak *Programming Python*, niż na tę; choć większość funkcjonalności biblioteki standardowej nadal jest obecna, Python 3.0 swobodnie zmienia nazwy modułów czy grupuje je w pakiety. Bardziej wyczerpującą listę zmian można znaleźć w dokumencie „What’s New in Python 3.0” w zbiorze dokumentacji biblioteki standardowej.

Przy migracji z Pythona 2.X na wersję 3.X należy pamiętać o zapoznaniu się ze skryptem do automatycznej konwersji kodu `2to3`, dostępnym w wersji 3.0. Nie jest on w stanie przełożyć na nową wersję Pythona wszystkiego, ale nieźle sobie radzi z przekształceniem większości kodu Pythona 2.X w taki sposób, by mógł on być wykonywany w wersji 3.X. W czasie gdy piszę te słowa, trwają również prace nad nowym projektem `3to2` służącym do przekładania kodu w Pythonie 3.X na kod środowiska 2.X. Oba narzędzia mogą się przydać, jeśli musimy utrzymywać kod dla obu gałęzi Pythona; więcej informacji na ich temat można znaleźć w Internecie.

Ponieważ wydanie czwarte niniejszej książki jest w dużej mierze uaktualnieniem do Pythona w wersji 3.0 z kilkoma nowymi rozdziałami, a dodatkowo wydanie poprzednie zostało opublikowane zaledwie dwa lata przed tym, dalszy ciąg niniejszej przedmowy pochodzi z poprzedniego wydania książki, z jedynie niewielkimi zmianami.

Tabela P.1. Zmiany języka Python w wersjach 2.6 oraz 3.0

Zmiana	Omówiona w rozdziale (rozdziałach)
Funkcja <code>print</code> w wersji 3.0	11.
Instrukcja <code>nonlocal x, y</code> w wersji 3.0	17.
Metoda <code>str.format</code> w wersji 2.6 oraz 3.0	7.
Typy łańcuchów znaków w wersji 3.0: <code>str</code> dla tekstu Unicode, <code>bytes</code> dla danych binarnych	7., 36.
Rozróżnienie plików tekstowych i binarnych w wersji 3.0	9., 36.
Dekoratory klas w wersji 2.6 oraz 3.0: <code>@private('age')</code>	31., 38.
Nowe iteratory w wersji 3.0: <code>range</code> , <code>map</code> i <code>zip</code>	14., 20.
Widoki słowników w wersji 3.0: <code>D.keys</code> , <code>D.values</code> , <code>D.items</code>	8., 14.
Operatory dzielenia w wersji 3.0: <code>reszta</code> , <code>/</code> oraz <code>//</code>	5.
Literaly zbiorów w wersji 3.0: <code>{a, b, c}</code>	5.
Zbiory składane w wersji 3.0: <code>{x**2 for x in seq}</code>	4., 5., 14., 20.
Słowniki składane w wersji 3.0: <code>{x: x**2 for x in seq}</code>	4., 8., 14., 20.
Obsługa łańcuchów cyfr binarnych w wersji 2.6 oraz 3.0: <code>Ob0101, bin(I)</code>	5.
Typ liczby ułamkowej w wersji 2.6 oraz 3.0: <code>Fraction(1, 3)</code>	5.
Anotacje funkcji w wersji 3.0: <code>def f(a:99, b:str)->int</code>	19.
Argumenty mogące być tylko słowami kluczowymi w wersji 3.0: <code>f(a, *b, c, **d)</code>	18., 20.
Rozszerzone rozpakowywanie sekwencji w wersji 3.0: <code>a, *b = seq</code>	11., 13.
Składnia importowania względnego dostępna dla pakietów w wersji 3.0: <code>from .</code>	23.
Menedżery kontekstu dostępne w wersji 2.6 oraz 3.0: <code>with/as</code>	33., 35.
Zmiany w składni wyjątków w wersji 3.0: <code>raise, except/as, klasy nadzędne</code>	33., 34.
Łączenie wyjątków w łańcuchy w wersji 3.0: <code>raise e2 from e1</code>	33.
Zmiany w słowach zarezerwowanych w wersjach 2.6 oraz 3.0	11.
Przelączanie na klasy w nowym stylu w wersji 3.0	31.
Dekoratory właściwości w wersji 2.6 oraz 3.0: <code>@właściwość</code>	37.
Użycie deskryptatorów w wersji 2.6 oraz 3.0	31., 38.
Użycie metaklas w wersji 2.6 oraz 3.0	31., 39.
Obsługa klas o abstrakcyjnej podstawie w wersji 2.6 oraz 3.0	28.

O trzecim wydaniu książki

W ciągu czterech lat dzielących publikację drugiego i trzeciego wydania książki w samym języku zaszły znaczące zmiany; podobnie zmieniły się zagadnienia prezentowane na szkoleniach z Pythona, które organizuję i prowadzę. Trzecie wydanie książki odzwierciedlało te zmiany. Przy okazji poprawiono również samą strukturę tej publikacji.

Zmiany w języku Python w trzecim wydaniu książki

Z punktu widzenia samego języka Python trzecie wydanie książki zostało uaktualnione do wersji Pythona zanumerowanej jako 2.5; wprowadzono również inne zmiany, które pojawiły się po wydaniu drugiej edycji książki z 2003 roku. Drugie wydanie książki bazowało z kolei na

Tabela P.2. Elementy usunięte z Pythona 3.0, które wpłynęły na niniejszą książkę

Usunięcie	Zastępnik	Omówione w rozdziale (rozdziałach)
reload(M)	imp.reload(M) (lub exec)	3., 22.
apply(f, ps, ks)	f(*ps, **ks)	18.
`X`	repr(X)	5.
X <> Y	X != Y	5.
long	int	5.
9999L	9999	5.
D.has_key(K)	K in D (lub D.get(key) != None)	8.
raw_input	input	3., 10.
old input	eval(input())	3.
xrange	range	14.
file	open (oraz klasy modułu io)	9.
X.next	X.__next__, wywoływane za pomocą next(X)	14., 20., 29.
X.__getslice__	X.__getitem__ po przekazaniu obiektu wycinka	7., 29.
X.__setslice__	X.__setitem__ po przekazaniu obiektu wycinka	7., 29.
reduce	functools.reduce (lub kod pętli)	14., 19.
execfile(nazwa_pliku)	exec(open(nazwa_pliku).read())	3.
exec open(nazwa_pliku)	exec(open(nazwa_pliku).read())	3.
0777	0o777	5.
print x, y	print(x, y)	11.
print >> F, x, y	print(x, y, file=F)	11.
print x, y,	print(x, y, end=' ')	11.
u'ccc'	'ccc'	7., 36.
'bbb' dlałańcuchów bajtowych	b'bbb'	7., 9., 36.
raise E, V	raise E(V)	32., 33., 34.
except E, X:	except E as X:	32., 33., 34.
def f((a, b)):	def f(x): (a, b) = x	11., 18., 20.
file.xreadlines	for line in file: (lub X=iter(file))	13., 14.
D.keys() itd. jako listy	list(D.keys()) (widoki słowników)	8., 14.
map(), range() itd. jako listy	list(map()), list(range()) (funkcje wbudowane)	14.
map(None, ...)	zip (lub ręczny kod dopełniający wyniki)	13., 20.
X=D.keys(); X.sort()	sorted(D) (lub list(D.keys()))	4., 8., 14.
cmp(x, y)	(x > y) - (x < y)	29.
X.__cmp__(y)	__lt__, __gt__, __eq__ itd.	29.
X.__nonzero__	X.__bool__	29.
X.__hex__, X.__oct__	X.__index__	29.
Sortujące funkcje porównujące	Należy użyć key=transform lub reverse=True	8.
Operacje <, >, <=, >= dla słowników	Porównanie sorted(D.items()) (lub kod pętli)	8., 9.
types.ListType	list (types przeznaczony jest jedynie dla nazw niewbudowanych)	9.

Tabela P.2. Elementy usunięte z Pythona 3.0, które wpłynęły na niniejszą książkę — ciąg dalszy

Usunięcie	Zastępnik	Omówione w rozdziale (rozdziałach)
<code>__metaclass__ = M</code>	<code>class C(metaclass=M):</code>	28., 31., 39.
<code>__builtin__</code>	<code>builtins (zmiana nazwy)</code>	17.
<code>Tkinter</code>	<code>tkinter (zmiana nazwy)</code>	18., 19., 24., 29., 30.
<code>sys.exc_type, exc_value</code>	<code>sys.exc_info()[0], [1]</code>	34., 35.
<code>function.func_code</code>	<code>function.__code__</code>	19., 38.
<code>__getattr__</code> wykonywane przez obiekty wbudowane	Ponowne zdefiniowane metod <code>__X__</code> w klasach opakowujących	30., 37., 38.
Opcje wiersza poleceń <code>-t, -tt</code>	Niespójne użycie tabulatorów i spacji zawsze jest błędem	10., 12.
<code>from ... * wewnętrz funkcji</code>	Może się pojawiać jedynie na najwyższym poziomie pliku	22.
<code>import mod w tym samym pakietie</code>	<code>from . import mod</code> , forma względem pakietu	23.
<code>class MyException:</code>	<code>class MyException(Exception):</code>	34.
Moduł exceptions	Zakres wbudowany, dokumentacja biblioteki	34.
Moduły threadQueue	<code>_thread, queue (zmiany nazw)</code>	17.
Moduł anydbm	<code>dbm (zmiana nazwy)</code>	27.
Moduł cPickle	<code>_pickle (zmiana nazwy, wykorzystywany automatycznie)</code>	9.
<code>os.popen2/3/4</code>	<code>subprocess.Popen</code> (zachowano <code>os.popen</code>)	14.
Wyjątki oparte na łańcuchach znaków	Wyjątki oparte na klasach (wymagane także w wersji 2.6)	32., 33., 34.
Funkcje modułu łańcuchów znaków	Metody obiektu łańcuchów znaków	7.
Metody bez wiązania	Funkcje (<code>staticmethod</code> wywoływana za pośrednictwem instancji)	30., 31.
Porównywanie i sortowanie obiektów o typach mieszanych	Porównania mieszanych typów nieliczbowych są błędem	5., 9.

wersji 2.2; na końcu projektu dodano również niektóre elementy z Pythona 2.3. W odpowiednich miejscach trzeciego wydania wstawione zostały również uwagi dotyczące zbliżającej się wersji 3.0. Poniżej znajduje się lista najważniejszych zagadnień, które zostały wprowadzone bądź uaktualnione w tej wersji książki (numery rozdziałów zostały uaktualnione i mają zastosowanie do wydania czwartego):

- wyrażenie warunkowe `B if A else C` (rozdział 19.),
- menedżery kontekstu — `with/as` (rozdział 33.),
- ujednolicenie `try/except/finally` (rozdział 33.),
- składnia importowania względnego (rozdział 23.),
- wyrażenia generatora (rozdział 20.),
- nowe możliwości funkcji generatora (rozdział 20.),
- dekoratory funkcji (rozdział 31.),
- typ obiektu zbioru (rozdział 5.),
- nowe funkcje wbudowane — `sorted, sum, any, all, enumerate` (rozdziały 13. oraz 14.),
- typ obiektu liczby dziesiętnej o stałej precyzji (rozdział 5.),
- pliki, listy składane oraz iteratory (rozdziały 14. oraz 20.),

- nowe narzędzia programistyczne — między innymi Eclipse, distutils, unittest oraz doctest, ulepszenia IDLE, Shedskin (rozdziały 2. oraz 35.).

Mniejsze zmiany w języku Python (na przykład rozszerzone użycie True oraz False, nowa funkcja sys.exc_info służąca do pobierania szczegółowych informacji o wyjątkach, zniknięcie wyjątków opartych na łańcuchach znaków, metod działających na łańcuchach znaków, wbudowanych funkcji apply oraz reduce) omawiane były w całej książce. Niektóre zagadnienia, które były nowością w wydaniu drugim, w wydaniu trzecim zostały omówione bardziej szczegółowo — w tym wycinki o trzech granicach oraz składnia wywołania o dowolnej liczbie argumentów, która zastąpiła funkcję apply.

Zmiany w szkoleniach z Pythona w trzecim wydaniu książki

Poza zmianami w samym języku trzecie wydanie książki wzbogacono o nowe zagadnienia oraz przykłady wykorzystywane dotychczas w szkoleniach z tego języka, jakie prowadziłem w ostatnich latach. Związane z tym zmiany objęły między innymi (numery rozdziałów zostały uaktualnione i mają zastosowanie do wydania czwartego):

- nowy rozdział będący wprowadzeniem do typów wbudowanych (rozdział 4.),
- nowy rozdział będący wprowadzeniem do składni instrukcji (rozdział 10.),
- nowy pełny rozdział na temat typów dynamicznych wraz z rozszerzonym omówieniem tego zagadnienia (rozdział 6.),
- nowe, rozbudowane wprowadzenie do programowania zorientowanego obiektowo (rozdział 25.),
- nowe przykłady, między innymi dla zagadnień związanych z plikami, zakresami, zagnieżdżaniem instrukcji, klasami oraz wyjątkami.

Wiele z tych zmian zostało wprowadzonych z myślą o początkujących użytkownikach tego języka. Niektóre tematy przeniesione zostały w miejsca, w których w czasie szkoleń wydawały się najłatwiejsze do zrozumienia. Listy składane oraz iteratory pojawiają się teraz na początku w połączeniu z instrukcją pętli for, a nie później, przy okazji omawiania narzędzi funkcyjnych.

W trzecim wydaniu książki rozbudowano również omówienia wielu podstawowych zagadnień dotyczących Pythona; dodano do nich także nowe przykłady. Ponieważ tekst książki stał się tak naprawdę standardowym podręcznikiem do nauki języka Python, prezentacja wszystkich zagadnień stała się bardziej kompletna i wzbogacona o nowe przykłady.

Co więcej, w książce zawarto również nowy zbiór sztuczek i porad, będących efektem ostatnich dziesięciu lat przeprowadzania szkoleń i piętnastu prawdziwej pracy w języku Python. Ćwiczenia zostały uaktualnione i rozszerzone w taki sposób, by obejmowały aktualne zalecenia i najlepsze praktyki w dziedzinie programowania w Pythonie, a także nowe opcje dostępne w tym języku i kwestie, które sprawiają początkującym użytkownikom największe problemy. Generalnie omówienie podstaw tego języka zostało rozszerzone w stosunku do poprzednich wydań.

Zmiany w strukturze wydania trzeciego

Ponieważ treść książki stała się bardziej kompletna, materiał został podzielony na części. Podzieliłem książkę na większą liczbę części składających się z kilku rozdziałów, tak by materiał ten był łatwiejszy do przyswojenia. Typy oraz instrukcje stały się na przykład dwoma częściami

wyższego rzędu, a każdemu ważniejszemu typowi i każdej instrukcji poświęcono osobny rozdział. Ćwiczenia oraz części opisujące najczęściej popełniane błędy zostały przeniesione z końców rozdziałów do końców części — pojawiają się zatem na końcu ostatniego rozdziału danej części.

W trzecim wydaniu książki wzbogaciłem również ćwiczenia końcowe o streszczenia podsumowujące oraz quizy pozwalające na podsumowanie poszczególnych rozdziałów. Każdy rozdział kończy się zestawem pytań, które pozwalają na sprawdzenie swojego zrozumienia przedstawionego w nim materiału. W przeciwnieństwie do ćwiczeń końcowych, których rozwiązania znajdują się w dodatku B, rozwiązania quizów znajdują się od razu po pytaniach. Polecam zaglądanie do nich nawet wtedy, gdy jest się pewnym, że na pytania odpowiedziało się poprawnie — odpowiedzi stanowią najczęściej podsumowanie danej kwestii.

Pomimo wprowadzenia wielu nowych zagadnień książka nadal przeznaczona jest dla użytkowników początkujących. Zaprojektowana jest także tak, by być pierwszym tekstem o Pythonie przeznaczonym dla szeroko pojętych programistów. Ponieważ książka oparta jest na prawdziwych, wypробowanych szkoleniach, może ona służyć jako kurs wprowadzający do Pythona dla samouków.

Zmiany w zakresie trzeciego wydania książki

Trzecie wydanie książki miało być przewodnikiem po podstawach języka Python, niczym innym. Skupiało się na dogłębny poznaniu tego języka jeszcze przed zastosowaniem go w programowaniu na poziomie aplikacji. Wszystko omawiane było od podstaw, co pozwalało na całościowe spojrzenie na Pythona, w oddzieleniu od ról pełnionych przez ten język w jego różnych zastosowaniach.

Dla niektórych osób „wprowadzenie do Pythona” oznacza poświęcenie godziny czy dwóch na przeczytanie jakiegoś przewodnika w Internecie. Takie podejście sprawdza się w przypadku bardziej zaawansowanych programistów — ale tylko do pewnego momentu. Python jest w końcu dość prosty w porównaniu z innymi językami. Takie podejście ma jednak zasadniczą wadę: przedżej czy później każdy trafia na jakiś dziwny przypadek, z którego nie potrafi wybrnąć — zmienne, których wartości zmieniają się w dziwny sposób, czy zmienne argumenty domyślne zachowujące się w sposób, który trudno zrozumieć. Celem niniejszej książki jest danie Czytelnikowi solidnych podstawa, tak by nawet te nietypowe przypadki stały się dla niego zrozumiałe, kiedy już się pojawią.

Taki zakres książki został wybrany celowo. Ograniczając się do podstaw tego języka, jesteśmy je w stanie omówić w dogłębny sposób. Inne, opisane dalej pozycje uzupełniają tekst niniejszej książki i pozwalają na pełniejsze omówienie dodatkowych zagadnień związanych ze stosowaniem Pythona. Celem niniejszej książki jest nauczenie podstaw Pythona, tak by każdy mógł go swobodnie stosować w dowolnej dziedzinie.

O książce

Niniejszy podrozdział jest dobrym miejscem na umieszczenie kilku ważnych informacji dotyczących samej książki, bez względu na numer wydania. Każda książka ma swoich odbiorców docelowych, dlatego warto wiedzieć z góry, jakie są cele niniejszej pozycji.

Wymagania w stosunku do Czytelników

Tak naprawdę nie ma żadnych wymagań. Z książkami z powodzeniem mogą korzystać zarówno osoby początkujące, jak i weterani programowania. Dla osób chętnych do nauki Pythona niniejsza pozycja powinna być dobrym punktem wyjścia. W praktyce zauważylem jednak, że jakieś kolwiek doświadczenie w dziedzinie programowania czy tworzenia skryptów jest często pomocne, choć nie jest wymagane.

Książka została zaprojektowana jako wprowadzenie do Pythona dla programistów¹. Może nie być idealnym tekstem dla osób, które nigdy nie miały kontaktu z komputerem (ponieważ nie będziemy tracić czasu na objaśnianie sposobu działania komputera). Nie zakładam jednak z góry, że Czytelnik zna jakieś języki programowania czy ma jakiekolwiek wykształcenie w dziedzinie informatyki.

Z drugiej strony, nie zamierzam obrażać Czytelników, nadając książce łatkę „dla opornych” — cokolwiek to nie znaczy. Robienie przydatnych rzeczy za pomocą tego języka programowania jest naprawdę proste, co pokaże niniejsza książka. W niektórych miejscach zestawiono ze sobą możliwości Pythona z innymi językami programowania, jak C, C++, Java czy Pascal, ale osoby, które nigdy nie korzystały z tych języków, mogą takie uwagi spokojnie zignorować.

Zakres książki w porównaniu z innymi pozycjami

Choć w niniejszej książce omówiono wszystkie najważniejsze elementy języka Python, celowo zawdzięlem jej zakres i rozmiar. By utrzymać jej charakter, skupiamy się na podstawach, wykorzystujemy krótkie, samodzielne przykłady ilustrujące poszczególne kwestie i czasami pomijamy detale, o których można z łatwością poczytać w dokumentacji Pythona. Z tej przyczyny niniejszą książkę najlepiej opisuje słowo „wprowadzenie” — jest ona wprowadzeniem do samego języka oraz bardziej skomplikowanych i zaawansowanych tekstów.

Nie będziemy na przykład mówili za wiele na temat integracji języków Python oraz C — to skomplikowane zagadnienie, które ma duże znaczenie w wielu systemach opartych na Pythonie. Nie będziemy również omawiać zbyt szczegółowo historii Pythona czy procesu rozwoju tego języka. O popularnych aplikacjach, takich jak graficzne interfejsy użytkownika, narzędzia systemowe, a także skrypty sieciowe, jedynie wspomnimy. Taki zakres nieco ogranicza ogólną perspektywę.

Python podnosi standardy w świecie języków skryptowych. Niektóre z idei tego języka wymagają poznania nieco szerszego kontekstu niż przedstawiony w niniejszej książce, dlatego po jej przeczytaniu zalecam zapoznanie się z kolejnymi pozycjami na temat tego języka. Mam nadzieję, że większość Czytelników niniejszej książki nie ograniczy się tylko do niej i pozną programowanie na poziomie aplikacji z innych tekstów.

Ze względu na wybór określonej grupy odbiorców docelowych — osób początkujących — książkę *Python. Wprowadzenie. Wydanie IV* w naturalny sposób uzupełniają inne pozycje wydawnictwa O'Reilly poświęcone temu językowi programowania. *Programming Python*, kolejna

¹ Mówiąc „programista”, mam na myśli każdego, kto w swoim życiu napisał choć jeden wiersz kodu w dowolnym języku programowania lub języku skryptowym. Osoby, do których nie odnosi się ten opis, powinny i tak skorzystać na lekturze niniejszej książki, choć powinny być świadome, że więcej czasu poświęca się w niej na naukę Pythona niż podstaw programowania.

książka mojego autorstwa, zawiera bardziej rozbudowane i skomplikowane przykłady, a także omawia programowanie aplikacji; została zaprojektowana jako bezpośredni ciąg dalszy niniejszej pozycji. Aktualne wydania książek *Python. Wprowadzenie. Wydanie IV* (w oryginale *Learning Python*) oraz *Programming Python* stanowią dwie części materiałów szkoleniowych ich autora — poświęcone podstawom tego języka oraz tworzeniu w nim aplikacji. Dodatkowo *Python. Leksykon kieszonkowy* (Helion; w oryginale *Python Pocket Reference* wydawnictwa O'Reilly) jest podręcznym leksykonem, który może się przydać do szybkiego odszukania pominiętych tutaj szczegółów.

Inne książki będące rozwinięciem tej udostępniają dodatkowe przykłady czy szczegóły dotyczące wykorzystywania Pythona w określonych, węższych dziedzinach, takich jak Internet czy graficzne interfejsy użytkownika. Przykładowo *Python in a Nutshell* (O'Reilly) oraz *Python Essential Reference* (Sams) to podręczne leksykon, a *Python. Receptury* (Helion; w oryginale *Python Cookbook* wydawnictwa O'Reilly) udostępnia bibliotekę gotowych przykładów umożliwiających zapoznanie się z różnymi technikami programistycznymi. Ponieważ wybór lektury jest sprawą subiektywną, polecam samodzielny dobór bardziej zaawansowanych tekstów. Bez względu na wybór należy pamiętać, że dogłębne poznanie Pythona wymaga zapoznania się z przykładami bardziej realistycznymi od tych zawartych w niniejszej książce.

Generalnie sądzę, że niniejsza pozycja jest dobrą pierwszą lekturą poświęconą temu językowi programowania — pomimo jej ograniczonego zakresu (a może właśnie dzięki temu). Dzięki tej książce każdy nauczy się wszystkiego, co jest potrzebne do pisania samodzielnych programów oraz skryptów w Pythonie. Umożliwia ona nie tylko opanowanie samego języka, ale również stosowanie go w codziennych zadaniach. Przygotowuje także Czytelnika na zajęcie się bardziej zaawansowanymi kwestiami oraz przykładami w przyszłości.

Styl oraz struktura książki

Niniejsza książka oparta jest na materiałach szkoleniowych przygotowanych na trzydniowy kurs języka Python. Pod koniec każdego rozdziału zamieszczono quizy; na końcu ostatniego rozdziału każdej części znajdują się ćwiczenia sprawdzające. Rozwiązania quizów znajdują się w tekście poszczególnych rozdziałów, natomiast rozwiązania ćwiczeń — w dodatku B. Quizy mają na celu powtórzenie materiału, natomiast ćwiczenia pomagają zacząć samodzielnie programować. Zazwyczaj są to te same ćwiczenia, które wykorzystuję na moich kursach.

Polecam regularne wykonywanie quizów oraz ćwiczeń w miarę studiowania kolejnych rozdziałów — nie tylko żeby poznać programowanie w Pythonie, ale również dlatego, że w niektórych ćwiczeniach pojawiają się zagadnienia, które nie są omawiane w innych miejscach. Rozwiązania quizów i ćwiczeń z rozdziałów i dodatku B pomogą poradzić sobie w sytuacji, kiedy nie wiadomo, co dalej zrobić (i polecam korzystanie z nich tak często i w takim stopniu, jaki dla każdego będzie odpowiedni).

Struktura książki również oparta jest na moim kursie. Ponieważ książka ma na celu szybkie zapoznanie Czytelnika z podstawami Pythona, ich prezentacja została podzielona zgodnie z najważniejszymi częściami samego języka, a nie przykładami. Zaczniemy od najmniejszych części języka i stopniowo będziemy przechodzić do bardziej ogólnych koncepcji — od wbudowanych typów, poprzez instrukcje, aż do części programów. Każdy rozdział jest niezależną całością, choć późniejsze rozdziały oparte są na zagadnieniach wprowadzonych w rozdziałach początkowych (czyli na przykład kiedy dojdziemy do klas, zakładam, że każdy rozumie sposób działania omówionych wcześniej funkcji). Czytanie książki po kolej dla większości osób będzie najlepszym rozwiązaniem.

Jak wspomniałem wcześniej, książka ta prezentuje Pythona od podstawowych zagadnień po te bardziej złożone. Każda jej część poświęcona jest jakiemuś szerszemu zagadnieniu — typom, funkcjom i tak dalej. Większość przykładów jest niewielka i samodzielna (niektóre osoby mogą je nawet uznać za sztuczne, ale przykłady służą jedynie do zilustrowania omawianego zagadnienia). W książce można znaleźć następujące części:

Część I, „Wprowadzenie”

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

Część II, „Typy i operacje”

Następnie rozpoczynamy naszą wycieczkę po języku Python, skupiając się na początku na najważniejszych wbudowanych typach danych — liczbach, listach czy słownikach. W Pythonie wiele można osiągnąć z użyciem samych tych narzędzi. To najważniejsza część książki, która stanowi podstawę dla kolejnych rozdziałów. Przyjrzymy się również typom dynamicznym oraz ich interfejsom — kolejnym kluczowym zagadnieniom w Pythonie.

Część III, „Instrukcje i składnia”

W kolejnej części omówione zostaną *instrukcje*, czyli kod, który pisze się w celu tworzenia i przetwarzania obiektów w Pythonie. Zaprezentujemy również ogólny model składni Pythona. Choć w tej części skupimy się na składni, omówimy również kilka powiązanych narzędzi, takich jak system PyDoc, a także alternatywne możliwości tworzenia kodu.

Część IV, „Funkcje”

Ta część jest początkiem omawiania struktur programistycznych wyższego poziomu. Funkcje to proste sposoby pakowania kodu, który nadaje się do ponownego użycia. W tej części omówimy reguły związane z zakresem, techniki przekazywania argumentów i podobne zagadnienia.

Część V, „Moduły”

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

Część VI, „Klasy i programowanie zorientowane obiektywem”

W tej części omówimy element z dziedziny programowania zorientowanego obiektywem — klasy. *Klasa* to opcjonalny i bardzo wydajny sposób strukturyzowania kodu w celu dopasowania go do własnych potrzeb i późniejszego ponownego wykorzystania. Jak się zresztą okaże, klasy wykorzystują koncepcje omówione we wcześniejszych rozdziałach książki, a programowanie zorientowane obiektywem w Pythonie polega na wyszukiwaniu zmiennych w połączonych obiektach. Jak zresztą zostanie to pokazane, programowanie zorientowane obiektywem jest w Pythonie opcjonalne, choć może znacząco zmniejszyć czas potrzebny na tworzenie programów, w szczególności w przypadku długofalowych projektów.

Część VII, „Wyjątki oraz narzędzia”

Tekst głównej części książki zamyka omówienie modelu obsługi wyjątków w Pythonie oraz wykorzystywanych w tym modelu instrukcji. Na koniec krótko przyjrzymy się narzędziom programistycznym, które staną się bardziej przydatne, kiedy zaczniemy pisać dłuższe programy (na przykład narzędziom do testowania i debugowania). Choć wyjątki są narzędziem stosunkowo prostym w użyciu, ta część pojawia się po omówieniu klas, ponieważ wszystkie wyjątki powinny teraz w Pythonie być klasami.

Część VIII, „Zagadnienia zaawansowane” (nowość w wydaniu czwartym)

W ostatniej części omawiamy niektóre bardziej zaawansowane zagadnienia. Zapoznamy się z łańcuchami Unicode oraz bajtowymi, narzędziami do zarządzania atrybutami, takimi jak właściwości i deskryptory, dekoratorami funkcji oraz klas, a także metaklasami. Rozdziały te są lekturą opcjonalną, ponieważ nie wszyscy programiści muszą poznać zagadnienia, o których one traktują. Z drugiej strony osoby, które muszą przetwarzanie zinternacjonalizowany tekst lub dane binarne bądź odpowiedzialne są za tworzenie API wykorzystywanych przez innych programistów, powinny w tej części znaleźć dla siebie coś ciekawego.

Dodatki

Książkę kończy kilka dodatków zawierających wskazówki na temat wykorzystywania Pythona na różnych platformach (dodatek A) oraz rozwiązań ćwiczeń podsumowujących kolejne części książki (dodatek B). Rozwiązania quizów znajdujących się na końcu poszczególnych rozdziałów znajdują się w tych rozdziałach.

Warto zauważyć, że wyszukiwanie w książce można sobie ułatwić, korzystając z indeksu oraz spisu treści, natomiast nie ma w niej osobnego dodatku stanowiącego spis instrukcji czy obiektów tego języka (książka jest podręcznikiem, nie leksykonem). Takie informacje uzyskać można w pozycji *Python. Leksykon kieszonkowy* (Helion), a także innych książkach oraz w darmowej dokumentacji dostępnej na stronie <http://www.python.org>. Tam można zapoznać się ze szczegółami dotyczącymi poszczególnych elementów składni czy komponentów wbudowanych.

Uaktualnienia książki

Czasami zdarzają się poprawki (podobnie jak czasami zdarzają się błędy w tekście). Uaktualnienia, suplementy oraz korekta dla oryginalnego, angielszczyznego wydania książki będą się znajdować pod jednym z poniższych adresów:

<http://www.oreilly.com/catalog/9780596158064/> (strona tej książki w witrynie wydawnictwa O'Reilly),

<http://www.rmi.net/~lutz> (strona autora książki),

<http://www.rmi.net/~lutz/about-lp.html> (strona autora poświęcona książce),

Trzeci z adresów kieruje do strony tego wydania książki, na której będę publikował uaktualnienia, jeśli jednak adres ten przestanie działać, warto poszukać tej strony za pomocą wyszukiwarki. Gdybym mógł przewidzieć wszystko, co stanie się w przyszłości, pewnie bym tak zrobił, jednak Internet zmienia się szybciej od druku.

Stronę polskiego wydania książki wraz z ewentualną erratą można znaleźć pod adresem <http://helion.pl/ksiazki/pythw4.htm>.

O programach zawartych w książce

Czwarte wydanie książki (wraz ze wszystkimi przykładowymi programami) oparte jest na Pythonie w wersji 3.0. Dodatkowo większość przykładów działa także w Pythonie 2.6, zgodnie z opisem w tekście, natomiast uwagi dla użytkowników wersji 2.6 można znaleźć w całej książce.

Ponieważ tekst książki skupia się na podstawach Pythona, można zakładać, że większość przedstawionych tu zagadnień nie zmieni się znacząco w kolejnych wersjach tego języka. Większość tekstu ma zastosowanie również do wersji wcześniejszych, choć nie zawsze. Oczywiście jest, że jeśli używa się rozszerzeń dodanych po wydaniu jakiejś starszej wersji, nie będą one działały.

Ogólna zasada jest taka: najnowszy Python to najlepszy Python. Ponieważ książka skupia się na podstawach tego języka, większość zagadnień ma zastosowanie również do Jythona — implementacji Pythona w Javie — a także innych implementacji opisanych w rozdziale 2.

Kod źródłowy przykładów wraz z rozwiązaniami ćwiczeń można pobrać z serwera wydawnictwa Helion (<ftp://ftp.helion.pl/przyklady/pythw4.zip>); odnośnik do materiałów znajduje się również na stronie polskiego wydania książki (<http://helion.pl/ksiazki/pythw4.htm>). W jaki sposób uruchamia się te przykłady? Kwestie te omówione zostaną w rozdziale 3., dlatego proszę o chwilę cierpliwości.

Wykorzystywanie fragmentów kodu

Książka ta ma na celu pomóc w wykonaniu pracy. Ogólnie rzecz biorąc, można wykorzystywać kod z książki w swoich programach i dokumentacji. Nie ma potrzeby kontaktowania się z nami w sprawie uzyskania na to zgody, chyba że kopiuje się znaczącą część kodu. Przykładowo napisanie programu, który używa kilku fragmentów kodu z tej książki, nie wymaga uzyskania zgody. Sprzedaż lub dystrybucja przykładów z książek wydawnictw O'Reilly i Helion na nośniku CD-ROM wymaga zgody. Udzielenie odpowiedzi poprzez zacytowanie książki i fragmentu kodu nie wymaga akceptacji. Włączenie znaczącej części przykładowego kodu z tej książki do dokumentacji własnego produktu wymaga jej.

Doceniamy cytowanie z podaniem źródła, choć go nie wymagamy. Podanie źródła zazwyczaj obejmuje tytuł, autora, wydawcę i numer ISBN. Na przykład: *Python. Wprowadzenie. Wydanie IV*, Mark Lutz, Helion 2010, ISBN 978-83-246-1648-0.

W razie przypuszczeń, że zamiar wykorzystania przykładów z książki wykracza poza dozwolony zakres podany tutaj, prosimy o kontakt pod adresem e-mail helion@helion.pl.

Konwencje wykorzystywane w książce

W książce wykorzystywane są następujące konwencje typograficzne:

Kursywa

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

Czionka o stałej szerokości

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

Pogrubiona czcionka o sta ej szeroko ci

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

Pochy a czcionka o sta ej szeroko ci

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

<Czcionka o sta ej szeroko ci>

Oznacza element, który powinien być zastąpiony prawdziwym kodem.



Ikona ta oznacza wskazówkę, sugestię lub ogólną uwagę odnoszącą się do tekstu.



Ikona ta oznacza ostrzeżenie lub uwagę.



Uwaga dotycząca tej ksi zki: W przykładach znak % na początku systemowego wiersza poleceń zastępuje znak zach ty systemu, czymkolwiek by on nie był na danym komputerze (na przykład w oknie konsoli DOS może to być C:\Python30>). Nie należy wpisywać znaku % (czy innego znaku, który on zastępuje) samodzielnie.

W podobny sposób w listingach z sesji interaktywnej interpretera nie należy wpisywać znaków >>> oraz ... pokazanych na początku wierszy — są to znaki zach ty wyświetlane przez Pythona. Wystarczy wpisać jedynie tekst znajdujący się po nich. W celu ułatwienia zapami tania tego rozróżnienia tekst wpisywany przez użytkownika jest w ksi zce zapisany czcionką pogrubioną.

Zazwyczaj nie trzeba również wpisywać tekstu rozpoczęjącego się w kodzie od znaku # — jak zostanie niedługo wyjaśnione, tekst ten jest komentarzem, a nie kodem wykonywalnym.

Podziękowania

Kiedy piszę czwarte wydanie tej ksi zki, w roku 2009, nie mogę się powstrzymać od poczucia zadowolenia związanego z dobrze wykonanym zadaniem. Od siedemnastu lat używam i promuję używanie Pythona. Od dwunastu lat prowadzę szkolenia z tego języka. Pomimo upływu czasu nadal jestem pod wrażeniem tego, jak wielki sukces osiągnął Python w tym okresie. Język ten rozwinał się w sposób, o którym większość nas w 1992 roku nawet nie śniła. I choć może to brzmieć jak egoistyczny wywód pochłoniętego samym sobą autora, Czytelnicy będą mi musieli wybaczyć par  słów wspomnień, gratulacji i podziękowa .

To była przysłowiowa d uga i kr ta droga. Spogl daj c na to dzisiaj, widz ,  e kiedy w 1992 roku odkryłem Pythona, nie mia em pojęcia, jak du y wpływ ten język programowania będzie miał na kolejnych siedemnascie lat mojego życia. Dwa lata po napisaniu pierwszego wydania ksi zki *Programming Python* (O'Reilly 1995) zacz lem podróżować po całym kraju i świecie, prowadząc szkolenia z Pythona dla osób pocz kuj cych i ekspertów. Od pierwszego wydania ksi zki *Python. Wprowadzenie* (*Learning Python*, O'Reilly 1999) sta em się niezale nym trenerem

i pisarzem zajmującym się tylko Pythonem — co było w dużej mierze zasługą błyskawicznie rosnącej popularności tego języka.

Pisząc te słowa w połowie 2009 roku, mam już na koncie dwanaście książek na temat Pythona (po cztery wydania trzech tytułów). Uczę Pythona od ponad dekady i dotychczas odbyłem około dwustu dwudziestu pięciu sesji szkoleniowych dla studentów ze Stanów Zjednoczonych, Europy, Kanady oraz Meksyku, których w sumie było ponad trzy tysiące. Poza zbieraniem mil w programach dla klientów linii lotniczych wszystkie szkolenia pomogły mi ulepszyć tekst tej książki, a także pozostałych pozycji. Przez te wszystkie lata szkolenia pomagały mi udoskonalać książki i odwrotnie. Tak naprawdę niniejsza książka jest prawie w całości oparta na moich materiałach szkoleniowych.

Z tego powodu chciałbym podziękować wszystkim studentom, którzy wzięli udział w moich kursach w ciągu ostatnich dwunastu lat. Oprócz zmian w samym Pythonie to Wasze reakcje i informacje zwrotne bardzo pomogły w ukształtowaniu tego tekstu (nie ma nic bardziej konstruktywnego od zobaczenia, jak trzy tysiące osób powtarza te same błędy!). Zmiany wprowadzone w tym wydaniu książki są przede wszystkim zasługą kursów odbytych po 2003 roku, choć każde szkolenie — od tych pierwszych, z 1997 roku — w jakiś sposób pomogło w ulepszeniu tej pozycji. Chciałbym szczególnie wyróżnić klientów, którzy organizowali kursy w Dublinie, Meksyku, Barcelonie, Londynie, Edmonton oraz Portoryko — trudno byłoby sobie wyobrazić lepsze kursy.

Chciałbym również wyrazić swoją wdzięczność każdemu, kto odegrał jakąś rolę w stworzeniu niniejszej książki. Dziękuję redaktorom, którzy pracowali nad tym projektem: Julie Steele (aktualne wydanie), Tatianie Apandi (poprzednie wydanie) oraz wielu osobom pracującym nad wcześniejszymi wydaniami. Dougowi Hellmannowi oraz Jessemu Nollerowi dziękuję za wzięcie udziału w korekcie merytorycznej książki. Wydawnictwu O'Reilly dziękuję za umożliwienie mi pracy nad dwunastoma projektami książek — to naprawdę świetna zabawa (nawet jeśli czasami czuję się jak w filmie *Dzień świstaka*).

Chcę podziękować współautorowi dwóch pierwszych wydań książki — Davidowi Ascherowi, za jego wkład w jej rozwój. David napisał część zatytułowaną „Warstwy zewnętrzne”, którą jednak w trzecim wydaniu byliśmy zmuszeni wyciąć, by zrobić miejsce dla nowych materiałów dotyczących Pythona. By to zrekompensować, na końcu tego wydania dodałem kilka bardziej zaawansowanych programów, które można samodzielnie przestudiować, natomiast w wydaniu czwartym dodane zostały zarówno nowe, bardziej zaawansowane przykłady, jak i zupełnie nowa część omawiająca zagadnienia zaawansowane. Wcześniej w przedmowie wymienione zostały również poświęcone dziedzinie aplikacji publikacje dodatkowe, do których można zajrzeć po opanowaniu podstaw języka Python w tej książce.

Jestem również winny podziękowania Guido van Rossumowi oraz całej społeczności skupionej wokół Pythona — za stworzenie tak przyjemnego w użyciu i przydatnego języka. Tak jak większość projektów z dziedziny open source, Python jest wytworem wielu heroicznych wysiłków. Po siedemnastu latach programowania w tym języku nadal uważam go za świetny. Wyróżnieniem było dla mnie obserwowanie, jak Python rozwija się od nowego języka skryptowego do szeroko wykorzystywanego narzędzia, używanego w jakimś stopniu przez prawie każdą organizację tworzącą oprogramowanie. Uczestniczenie w jego rozwoju było wyjątkowym przeżyciem, za co całej społeczności Pythona serdecznie dziękuję, wyrażając jednocześnie swój podziw i gratulując dobrej roboty.

Chciałbym również podziękować mojemu pierwszemu redaktorowi w wydawnictwie O'Reilly, nieodóżałowanemu Frankowi Willisonowi. Niniejsza książka była w dużej mierze pomysłem Franka i odzwierciedla wizję, którą od niego przejąłem. Spoglądając w przeszłość, widzę, jak wielki wpływ Frank miał nie tylko na moją ścieżkę zawodową, ale również na samą społeczność Pythona. Nie będzie żadną przesadą powiedzieć, że Frank jest odpowiedzialny za wiele radości i sukcesów związanych z Pythonem w początkach jego istnienia. Nadal nam go bardzo brakuje.

Na koniec kilka osobistych uwag i podziękowań. Dziękuję OQO za najlepsze zabawki, jakie dotychczas widziałem (póki istniały). Świętej pamięci Carlowi Saganowi dziękuję za zainspirowanie osiemnastolatka z Wisconsin. Mojej mamie dziękuję za jej odwagę. Wszystkim dużym korporacjom, z którymi miałem kontakt przez wszystkie te lata, dziękuję za przypomnienie mi, jak bardzo jestem szczęśliwy z powodu tego, że od ponad dekady jestem swoim własnym szefem.

Do moich dzieci — Mike'a, Sammy oraz Roxy — bez względu na to, jaką drogę obierzecie w przyszłości: kiedy zaczynałem pracę z Pythonem, byliście jeszcze mali. Wygląda na to, że w tym czasie sporo podrośliście i jestem z Was bardzo dumny. Życie może zmusić nas wszystkich do wyboru różnych dróg, ale jedna droga zawsze pozostanie otwarta — droga do domu.

Przede wszystkim jednak dziękuję Verze — mojej najlepszej przyjaciółce, dziewczynie i żonie. Najlepszym dniem mojego życia był ten, kiedy Cię w końcu znalazłem. Nie wiem, co przyniesie mi następne pięćdziesiąt lat, ale wiem, że chcę je spędzić z Tobą.

— Mark Lutz
Sarasota, Floryda
lipiec 2009

CZĘŚĆ I

Wprowadzenie

Pytania i odpowiedzi dotyczące Pythona

Osoby, które kupiły niniejszą książkę, wiedzą zapewne, czym jest Python i dlaczego warto się go nauczyć. Jeśli tak nie jest, być może nie dowiedzą się tego, dopóki nie opanują tego języka, czytając resztę książki i wykonując parę pierwszych projektów. Zanim jednak przejdziemy do szczegółów, pierwszych kilka stron książki poświęcimy krótkiemu wprowadzeniu do elementów składających się na popularność Pythona. Żeby zacząć tworzyć definicję tego języka, najpierw przyjrzyjmy się pytaniom często zadawanym przez osoby początkujące, a także odpowiedziom na nie.

Dlaczego ludzie używają Pythona?

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

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

Jakość oprogramowania

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

Wydajność programistów

Python wielokrotnie zwiększa wydajność i produktywność programistów w porównaniu z językami kompilowanymi czy o statycznych typach, takimi jak C, C++ czy Java. Kod w Pythonie stanowi średnio jedną trzecią do jednej piątej rozmiaru jego odpowiednika

w C++ czy Javie. Oznacza to mniejszą liczbę znaków do wpisania, a także mniejszą ilość kodu do sprawdzenia i utrzymania w przyszłości. Programy napisane w Pythonie działają natychmiast, bez konieczności długiej komplikacji i korzystania z narzędzi zewnętrznych, co jeszcze bardziej zwiększa szybkość tworzenia kodu.

Przenośność programów

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

Obsługa bibliotek

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

Integracja komponentów

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

Przyjemność

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

Z tych czynników pierwsze dwa (jakość oraz wydajność) są najprawdopodobniej najważniejszymi korzyściami dla większości użytkowników Pythona.

Jakość oprogramowania

W Pythonie celowo zaimplementowano prostą i czytelną składnię, a także bardzo spójny model programowania. Jak podkreśla slogan jednej z ostatnich konferencji na temat Pythona, rezultat takiego działania jest taki, że Python zdaje się „pasować do naszego sposobu myślenia” — co oznacza, że możliwości tego języka pozostają ze sobą spójne i są naturalną konsekwencją niewielkiej liczby podstawowych koncepcji. To sprawia, że język ten łatwo jest zrozumieć, łatwo też nauczyć się go i zapamiętać. W praktyce programiści Pythona, kiedy czytają lub tworzą

kod, nie muszą się ciągle odwoływać do podręczników i dokumentacji. Python to spójnie zaprojektowany system, który — w opinii wielu osób — powoduje tworzenie kodu wyglądającego zaskakująco regularnie.

Filozofia Pythona zakłada minimalizm. Oznacza to, że choć istnieje wiele sposobów wykonania jednego zadania, zazwyczaj istnieje jedna oczywista droga, kilka mniej oczywistych alternatyw i niewielki zbiór spójnych interakcji w każdym obszarze tego języka. Co więcej, Python nie podejmuje za nas (jedynych słusznych) decyzji — kiedy jakąś interakcję jest niejasna, preferowane są jawne interwencje. W filozofii Pythona jawne jest lepsze od niejawnego, a proste — od skomplikowanego.¹

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

Wydajność programistów

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

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

Czy Python jest językiem skryptowym?

Python jest językiem programowania ogólnego przeznaczenia, który często wykorzystywany jest do tworzenia skryptów. Często nazywa się go *zorientowanym obiektowo skryptowym językiem programowania* — w tej definicji łączy się obsługę programowania zorientowanego obiektowo z przeznaczeniem do tworzenia skryptów. Wiele osób często używa słowa „skrypt” w miejscu „program”, kiedy opisuje plik kodu Pythona. W tej książce oba te terminy używane są wymiennie;

¹ Bardziej rozbudowane spojrzenie na filozofię Pythona można uzyskać, wpisując polecenie `import this` w wierszu poleceń Pythona (jak to zrobić — można przeczytać w rozdziale 2.). Polecenie to wywołuje ukrytą w tym języku pewną niespodziankę — zbiór zasad projektowych będących podstawą Pythona. Akronim EIBTI jest aktualnie modałnym żargonem oznaczającym „jawne jest lepsze od niejawnego” (ang. *explicit is better than implicit*).

słowo „skrypt” jest preferowane w kontekście prostych plików najwyższego poziomu, natomiast „program” — w odniesieniu do bardziej skomplikowanych aplikacji składających się z wielu plików.

Ponieważ pojęcie „język skryptowy” dla każdego znaczy co innego, niektóre osoby wołałyby, żeby w ogóle nie stosować go w kontekście Pythona. Tak naprawdę ludzie mają trzy różne skojarzenia, kiedy słyszą, że Python jest językiem skryptowym:

Narzędzia powłoki

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

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

Język zarządzania

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

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

Łatwość użycia

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

Nie należy jednak dać się zwieść — Python nie służy tylko do prostych zadań. Lepiej będzie powiedzieć, że to sam Python upraszcza wiele zadań dzięki swojej elastyczności i łatwości użycia. Python ma prosty zbiór możliwości, który jednak pozwala na skalowanie programów do tak rozbudowanych, jak jest to potrzebne. Z tego powodu wykorzystywany jest zarówno w krótkich, taktycznych zadaniach, jak i w długofalowych, strategicznych celach programistycznych.

Czy zatem Python jest językiem skryptowym? Zależy, kogo o to zapytać. Generalnie określenia „skryptowy” najlepiej będzie używać w odniesieniu do szybkiego, elastycznego trybu programowania, który Python w pełni obsługuje, a nie do pewnej dziedziny zastosowania języka.

Jakie są zatem wady Pythona?

Używając Pythona od siedemnastu lat i nauczając go od dwunastu, zauważylem, że jedyną wadą w jego obecnej implementacji jest to, iż prędkość wykonywania może nie zawsze być porównywalna z prędkością języków kompilowanych, takich jak C czy C++.

O implementacji pomówimy w kolejnych rozdziałach książki. W skrócie mówiąc, dzisiejsze standardowe implementacje Pythona kompilują (przekładają) instrukcje z kodu źródłowego na format pośredni nazywany *kodem bajtowym* (ang. *byte code*), a następnie interpretują kod bajtowy. Kod bajtowy zapewnia przenośność aplikacji, ponieważ jest to format niezależny od platformy. Ponieważ jednak kod w Pythonie nie jest kompilowany do poziomu binarnego kodu maszynowego (na przykład instrukcji dla chipa firmy Intel), niektóre programy będą działały wolniej w Pythonie w porównaniu z aplikacjami napisanymi w języku w pełni kompilowanym, takim jak C.

To, czy różnica w szybkości wykonywania będzie miała *jakieś znaczenie*, zależy od typu programów, jakie będziemy pisać. Python wiele razy był optymalizowany, a kod napisany w tym języku sam z siebie działa wystarczająco szybko dla większości zastosowań. Co więcej, w przypadku większości „prawdziwych” zadań w skrypcie napisanym w Pythonie, takich jak przetwarzanie plików czy konstruowanie graficznego interfejsu użytkownika (GUI), program i tak działa z prędkością programu w języku C, ponieważ takie zadania są wewnętrz interpretera Pythona natychmiast wykonywane jako kod w C. Co jednak ważniejsze, zyski w szybkości pisania kodu w Pythonie są o wiele istotniejsze od ewentualnego obniżenia szybkości wykonywania, w szczególności na współczesnych komputerach.

Jednak nawet przy dzisiejszych możliwościach procesorów istnieją dziedziny, w których optymalna szybkość wykonywania jest istotna. W programowaniu numerycznym czy animacjach często wymagane jest, by przynajmniej najważniejsze komponenty przetwarzające liczby działały z prędkością programów w języku C (lub lepszą). Kiedy pracuje się w takiej dziedzinie, nadal można wykorzystywać Pythona — wystarczy oddzielić części aplikacji wymagające optymalnej szybkości jako *kompilowane rozszerzenia* i połączyć je z całym systemem za pomocą skryptów napisanych w Pythonie.

W niniejszej książce nie będziemy zbyt szeroko omawiać rozszerzeń, jednak jest to jeden z przypadków stosowania Pythona jako języka zarządzania. Doskonałym przykładem tego typu strategii jest rozszerzenie do programowania numerycznego *NumPy*. Dzięki połączeniu skompilowanych i zoptymalizowanych bibliotek numerycznych z językiem Python, NumPy sprawia, że Python staje się wydajnym i łatwym w użyciu narzędziem do programowania numerycznego. Może się okazać, że tworzenie takich rozszerzeń w naszej pracy nie będzie konieczne, jednak dobrze jest wiedzieć, że jeśli będzie nam to potrzebne, w Pythonie dostępne są doskonałe mechanizmy optymalizacyjne.

Kto dzisiaj używa Pythona?

W chwili pisania niniejszej książki, można szacować, że na całym świecie jest obecnie około miliona użytkowników Pythona. Ta liczba wynika z kilku statystyk, na przykład liczby pobrań na stronie Pythona czy ankiet przeprowadzanych wśród programistów. Ponieważ Python jest produktem open source (o otwartym kodzie źródłowym), trudno jest dokonać dokładniejszych

obliczeń, ponieważ nie można zliczać liczby wykupionych licencji. Co więcej, Python jest częścią dystrybucji Linuksa, systemu operacyjnego komputerów Macintosh, a także innych produktów czy urządzeń, co jeszcze utrudnia dokładniejsze statystyki użytkowników.

Ogólnie rzecz biorąc, Python cieszy się sporą bazą użytkowników; wokół tego języka skupiona jest też bardzo aktywna społeczność programistów. Ponieważ język ten istnieje od około dwudziestu lat i jest w szerokim użyciu, jest stabilny i ma duże możliwości. Oprócz użycia przez indywidualne osoby stosuje się go również w wielu produktach generujących przychody rozmaitym firmom. Na przykład:

- Firma Google intensywnie wykorzystuje Pythona w swojej wyszukiwarce, a także zatrudnia twórcę tego języka.
- Serwis służący do dzielenia się filmami wideo YouTube jest w większości napisany w Pythonie.
- Popularny system dzielenia się plikami w systemie p2p, BitTorrent, jest programem napisanym w Pythonie.
- Popularna platforma programowania aplikacji webowych firmy Google o nazwie App Engine wykorzystuje Pythona w roli języka aplikacji.
- Python jest intensywnie wykorzystywany w EVE Online, grze typu MMOG (Massively Multiplayer Online Game).
- API do tworzenia skryptów oparte na Pythonie można znaleźć w programie Maya, zintegrowanym systemie modelowania 3D i animacji o ogromnych możliwościach.
- Firmy Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm oraz IBM wykorzystują Pythona w testowaniu urządzeń.
- Firmy Industrial Light & Magic, Pixar i inne wykorzystują Pythona w tworzeniu filmów animowanych.
- Firmy JPMorgan Chase, UBS, Getco i Citadel stosują Pythona w prognozach finansowych.
- Instytucje, takie jak NASA, Los Alamos, Fermilab, JPL i inne, wykorzystują Pythona do zadań programistycznych w różnych dziedzinach nauki.
- Firma iRobot wykorzystuje Pythona do rozwijania komercyjnych robotów.
- ESRI wykorzystuje Pythona jako narzędzie służące do dostosowania ich popularnych produktów z mapami GIS do potrzeb użytkownika.
- Agencja NSA wykorzystuje Pythona w kryptografii oraz analizach wywiadowczych.
- Serwer poczty elektronicznej IronPort wykorzystuje ponad milion wierszy kodu w Pythonie do swojej codziennej pracy.
- Projekt One Laptop Per Child (OLPC) buduje interfejs użytkownika oraz model aktywności w Pythonie.

Lista ta nie jest skończona. Chyba jedynym elementem łączącym różne zastosowania Pythona jest to, że język ten wykorzystywany jest na całym świecie i we wszystkich dziedzinach. Python jest językiem ogólnego zastosowania, dzięki czemu można go wykorzystywać w prawie wszystkich dziedzinach. Można właściwie powiedzieć, że Python wykorzystywany jest przez prawie każdą znaczącą organizację tworzącą oprogramowanie, obojętnie czy pod postacią krótkich zadań taktycznych (testowanie, administracja), czy w rozwoju strategicznych produktów. Język ten sprawdza się w obu sytuacjach.

Więcej informacji na temat firm używających obecnie Pythona można znaleźć na stronie <http://www.python.org>.

Co mogę zrobić za pomocą Pythona?

Python jest nie tylko dobrze zaprojektowanym językiem programowania — to także język przydatny w wykonywaniu prawdziwych zadań, takich, z jakimi programiści spotykają się na co dzień. Jest używany w wielu różnych dziedzinach jako narzędzie do tworzenia skryptów dla innych komponentów, a także służące do implementowania samodzielnych programów. Jako język ogólnego przeznaczenia Python może być wykorzystywany właściwie wszędzie — można go zastosować w dowolnej dziedzinie, od tworzenia witryn internetowych po programowanie gier, a także robotykę i kontrolowanie statków kosmicznych.

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

Programowanie systemowe

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

Standardowa biblioteka Pythona zawiera wiązania POSIX i obsługę wszystkich najczęściej spotykanych elementów systemu operacyjnego — zmiennych środowiskowych, plików, gniazd, potoków, procesów, wielu wątków, dopasowywania wzorców wyrażeń regularnych, argumentów wiersza poleceń, standardowych interfejsów strumieni, programów uruchamianych z wiersza poleceń, rozszerzania nazw plików. Dodatkowo większość interfejsów systemowych Pythona zaprojektowano pod kątem przenośności. Przykładowo skrypt kopiący drzewa katalogów zazwyczaj działa tak samo na wszystkich najważniejszych platformach. System *Stackless Python* wykorzystywany w EVE Online oferuje zaawansowane rozwiązania na potrzeby wieloprzetwarzania (ang. *multiprocessing*).

Graficzne interfejsy użytkownika

Prostota oraz szybkość programowania w Pythonie sprawiają, że język ten często wykorzystywany jest w programowaniu graficznych interfejsów użytkownika (GUI, od ang. *graphical user interface*). W Pythonie znajduje się standardowy, zorientowany obiektowo interfejs do Tk GUI API o nazwie *tkinter* (w wersji 2.6 *Tkinter*), pozwalający na implementowanie przenośnych GUI o wyglądzie danego systemu operacyjnego w programach napisanych w tym języku. Graficzne interfejsy użytkownika oparte na Pythonie i tkinter działają bez większych zmian w systemach Microsoft Windows, X Windows (Unix oraz Linux), a także Mac OS (zarówno Classic, jak

i OS X). Darmowy pakiet rozszerzeń, *PMW*, dodaje do interfejsu tkinter zaawansowane widgety. Kolejne GUI API, *wxPython*, oparte na bibliotece C++, oferuje alternatywny zbiór narzędzi służących do tworzenia przenośnych graficznych interfejsów użytkownika w Pythonie.

Zbiory narzędzi wyższego poziomu, takie jak *PythonCard* czy *Dabo*, zbudowane są na bazie API, takich jak *wxPython* oraz *tkinter*. Za pomocą odpowiedniej biblioteki w Pythonie można również wykorzystać inne GUI, takie jak *Qt* (z wykorzystaniem PyQT), *GTK* (za pomocą PyGTK), *MFC* (dzięki PyWin32), *.NET* (IronPython) czy *Swing* (Jython — opisana w rozdziale 2. wersja Pythona oparta na Javie — lub JPype). Dla aplikacji działających w przeglądarkach lub mających małe wymagania w zakresie interfejsu zarówno Jython, jak i opisane poniżej platformy webowe w Pythonie oraz skrypty CGI po stronie serwera udostępniają dalsze opcje.

Skrypty internetowe

Python zawiera standardowe moduły internetowe, które pozwalają programom napisanym w tym języku na wykonywanie różnorodnych zadań sieciowych zarówno w trybie klienta, jak i serwera. Skrypty mogą się komunikować za pośrednictwem gniazd, mogą pobierać informacje z formularzy przesyłanych do skryptów CGI po stronie serwera, dokonywać transmisji za pomocą protokołu FTP, analizować składniowo, generować i przetwarzanie pliki XML, wysyłać, otrzymywać, tworzyć i przetwarzanie wiadomości e-mail, pobierać całe strony internetowe za pomocą ich adresów URL, przetwarzanie kod XHTML oraz XML pobranych stron, komunikować się za pośrednictwem XML-RPC, SOAP czy Telnetu. Biblioteki Pythona sprawiają, że wszystkie te zadania są zaskakująco proste.

Dodatkowo w Internecie dostępnych jest wiele zbiorów narzędzi, dzięki którym programowanie webowe w Pythonie staje się jeszcze łatwiejsze. System *HTMLGen* generuje na przykład pliki HTML z opisów Pythona opartych na klasach, a pakiet *mod_python* uruchamia Pythona na serwerze Apache i obsługuje szablony po stronie serwera za pomocą Python Server Pages. System Jython umożliwia bezproblemową integrację Pythona i Javy, a także obsługuje kodowanie appletów po stronie serwera, które działają na kliencie.

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

Integracja komponentów

Integrację komponentów omówiliśmy już wcześniej, kiedy opisywaliśmy Pythona jako język zarządzania. Możliwość rozszerzania Pythona za pomocą języków C i C++, a także osadzania go w kodzie w tych językach sprawia, że jest on niezwykle przydatnym spokiem służącym do tworzenia skryptów kontrolujących zachowanie innych systemów i komponentów. Integracja biblioteki języka C z Pythonem pozwala na przykład Pythonowi na testowanie i uruchamianie komponentów tej biblioteki. Osadzenie Pythona w produkcie pozwala na wprowadzanie zmian na miejscu, bez konieczności ponownej komplikacji całego produktu (czy przesyłania jego kodu źródłowego).

Narzędzia takie, jak generatory kodu *SWIG* czy *SIP*, mogą pomóc zautomatyzować wiele z czynności niezbędnych do połączenia skompilowanych komponentów w Pythonie dla celów skryptów, natomiast system *Cython* pozwala programistom na łączenie kodu Pythona i przypominającego języka C. Większe platformy, takie jak obsługa COM w systemie Windows, implementacja Jython oparta na Javie, oparta na platformie .NET implementacja IronPython czy różne zestawy narzędzi CORBA dla Pythona, udostępniają alternatywne sposoby tworzenia skryptów dla komponentów. W systemie Windows skrypty napisane w Pythonie można na przykład wykorzystać w połączeniu z programami Microsoft Word czy Excel.

Programowanie bazodanowe

Dla Pythona istnieją interfejsy do wszystkich najpopularniejszych relacyjnych baz danych — jak Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite. Zdefiniowano również *przenośne API dla baz danych* służące do dostępu do systemów baz danych SQL ze skryptów napisanych w Pythonie; wygląda ono tak samo dla wielu różnych systemów baz danych. Ponieważ interfejsy różnych producentów implementują przenośne API, skrypt napisany dla darmowej bazy danych MySQL będzie w zasadzie działał bez większych zmian w innych systemach (na przykład Oracle). Wystarczy tylko zastąpić interfejs producenta.

Standardowy moduł Pythona *pickle* udostępnia prosty system *trwałości obiektu* (ang. *object persistence*) pozwalający na łatwe zapisywanie i przywracanie całych obiektów Pythona do i z plików, a także obiektów podobnych do plików. W Internecie można również znaleźć system open source o nazwie *ZODB*, który udostępnia pełny system bazy danych zorientowanej obiektywnie dla skryptów Pythona, a także podobne systemy (jak *SQLObject* czy *SQLAlchemy*) odwzorowujące tabele relacyjne na model klas Pythona. Co więcej, od wersji 2.5 Pythona osadzony silnik bazy danych SQL (*SQLite*) jest standardową częścią samego języka.

Szybkie prototypowanie

Dla programów w Pythonie komponenty napisane w Pythonie i C wyglądają tak samo. Z tego powodu możliwe jest początkowe prototypowanie systemów w Pythonie i późniejsze przenoszenie wybranych komponentów do języka kompliowanego, takiego jak C czy C++. W przeciwieństwie do niektórych narzędzi do prototypowania Python nie wymaga całkowitego przepisania komponentu po jego ustabilizowaniu. Części systemu, które nie potrzebują wydajności języka takiego, jak C++, mogą pozostać zapisane w Pythonie ze względu na łatwość utrzymywania i użycia.

Programowanie numeryczne i naukowe

Wspomniane wcześniej rozszerzenie do programowania numerycznego *NumPy* zawiera zaawansowane narzędzia, takie jak obiekt tablicy czy interfejsy do standardowych bibliotek matematycznych. Integrując Pythona z procedurami numerycznymi zakodowanymi w języku kompliowanym ze względu na szybkość, NumPy sprawia, że Python staje się zaawansowanym, a przy tym łatwym w użyciu narzędziem do programowania numerycznego, które często jest w stanie zastąpić istniejący kod napisany w tradycyjnych językach kompliowanych, takich jak FORTRAN czy C++. Dodatkowe narzędzia numeryczne dla Pythona obsługują między

innymi animacje, wizualizacje trójwymiarowe i przetwarzanie równolegle. Popularne rozszerzenia *SciPy* oraz *ScientificPython* udostępniają na przykład dodatkowe biblioteki narzędzi służących do programowania naukowego i korzystają z kodu NumPy.

Gry, grafika, porty szeregowe, XML, roboty i tym podobne

Python jest często stosowany w innych dziedzinach niż omówione wyżej. Możliwe są na przykład następujące zastosowania:

- Programowanie gier oraz multimediorów dzięki systemowi *pygame*.
- Komunikacja przez port szeregowy w systemach Windows, Linux i innych dzięki rozszerzeniu *PySerial*.
- Przetwarzanie grafiki wykonywane za pomocą narzędzi *PIL*, *PyOpenGL*, *Blender* lub *Maya*, a także innych.
- Kontrolowanie robotów za pomocą zestawu narzędzi *PyRo*.
- Przetwarzanie dokumentów XML za pomocą pakietu biblioteki *xml*, modułu *xmlrpclib* oraz rozszerzeń.
- Programowanie sztucznej inteligencji (AI) za pomocą symulatorów sieci neuronowej i zaawansowanych powłok systemowych.
- Analiza języka naturalnego dzięki pakietowi *NLTK*.

Za pomocą programu *PySol* można nawet układać pasjansa! Obsługę wielu podobnych dziedzin można znaleźć na stronie internetowej PyPI, a także za pomocą wyszukiwarki (odnośniki można znaleźć w wyszukiwarce Google lub na stronie <http://www.python.org>).

Najczęściej te określone dziedziny zastosowania są w dużej mierze po prostu przypadkami wykorzystania Pythona do integracji komponentów. Dodawanie Pythona jako frontendu do bibliotek komponentów napisanych w językach kompilowanych (takich, jak C) sprawia, że Python przydaje się do pisania skryptów przydatnych w wielu zastosowaniach. Python jest szeroko stosowany jako język ogólnego przeznaczenia obsługujący integrację.

Jakie wsparcie techniczne ma Python?

Jako popularny system z dziedziny open source Python zgromadził dużą i aktywną społeczność programistów reagujących na zgłoszane problemy i proponujących ulepszenia z szybkością, jaka może zrobić wrażenie na wielu twórcach programów komercyjnych (by nie powiedzieć, że może być dla nich szokującą). Programiści Pythona koordynują pracę za pomocą systemu kontroli wersji. Wszelkie modyfikacje muszą być zgodne z formalnym protokołem *PEP* (Python Enhancement Proposal — „Propozycja ulepszenia Pythona”) i muszą zawierać rozszerzenia dla obszernego systemu testów regresyjnych. Tak naprawdę proces wprowadzania jakichkolwiek zmian do języka jest podobnie złożony jak zmiany w programach komercyjnych i nie ma wiele wspólnego z początkami Pythona, gdy wystarczyło wysłać e-maila do programisty. Biorąc pod uwagę ogromną liczbę użytkowników, jest to zmiana na lepsze.

Fundacja *PSF* (Python Software Foundation), formalna organizacja non-profit, organizuje konferencje i zajmuje się kwestiami własnością intelektualnej. Na całym świecie odbywają się liczne

konferencje na temat Pythona — największymi z nich są OSCON organizowana przez O'Reilly oraz organizowana przez PSF PyCon. Pierwsza z nich poświęcona jest większej liczbie projektów open source, natomiast druga zajmuje się wyłącznie Pythonem, a liczba jej uczestników w ostatnich latach znacznie wzrosła. Konferencja PyCon 2008 prawie podwoiła liczbę uczestników w stosunku do poprzedniego roku — z 586 w 2007 roku do ponad 1000 w roku 2008. Rok wcześniej liczba uczestników wzrosła o 40% — w stosunku do 410 osób obecnych w 2006 roku. Na konferencji PyCon 2009 odnotowano 943 uczestników — niewielki spadek w porównaniu z poprzednim rokiem, jednak nadal była to ogromna liczba, biorąc pod uwagę globalną recesję.

Jakie są techniczne mocne strony Pythona?

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

Jest zorientowany obiektowo

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

Choć służy jako doskonałe narzędzie do strukturyzacji oraz ponownego wykorzystania kodu, zorientowanie obiektowe w Pythonie idealnie sprawdza się również w tworzeniu skryptów dla innych języków zorientowanych obiektowo, takich jak C++ czy Java. Za pomocą odpowiedniego kodu spajającego programy napisane w Pythonie mogą służyć do tworzenia podklas dla klas zaimplementowanych w językach C++, Java czy C#.

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

Jest darmowy

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

Nie należy jednak tego źle zrozumieć: „darmowy” nie znaczy, że Python nie ma żadnego wsparcia technicznego. Wręcz przeciwnie — społeczność programistów Pythona odpowiada na zapytania użytkowników z prędkością, jakiej może im pozazdrościć niejeden producent i sprzedawca oprogramowania komercyjnego. A ponieważ Python dostępny jest wraz z pełnym kodem źródłowym, daje to szerokie możliwości programistom, przez co powstaje ogromna grupa specjalistów od implementacji tego języka. Choć studiowanie czy próba zmiany implementacji nie jest ulubioną rozrywką dla każdego, bezpiecznie jest wiedzieć, że w razie problemów kod źródłowy dostępny jest jako ostatnia deska ratunku i źródło dokumentacji. Nie jesteśmy uzależnieni od kaprysów komercyjnego producenta czy sprzedawcy.

Jak wspomniano wcześniej, Python rozwijany jest przez społeczność, która w dużej mierze koordynuje swoje wysiłki za pomocą Internetu. Składa się ona z twórcy Pythona — *Guido van Rossum*, oficjalnie noszącego tytuł Benevolent Dictator for Life (BDFL — po polsku „dobrowolny, dożywotni dyktator”) — oraz tysiące innych osób wspierających. Wszelkie zmiany w języku muszą przejść odpowiednią procedurę; muszą również być sprawdzone przez van Rossuma oraz innych programistów. Taka procedura zatwierdzania zmian sprawia, że Python jest na szczęście bardziej konserwatywny w tej materii niż niektóre języki programowania.

Jest przenośny

Standardowa implementacja Pythona napisana jest w przenośnym ANSI C; może być kompilowana i działa praktycznie na każdej ważniejszej platformie będącej obecnie w użyciu. Programy napisane w Pythonie działają obecnie zarówno na PDA, jak i na superkomputerach. Python jest dostępny między innymi na platformach takich, jak:

- Linux oraz Unix,
- Microsoft Windows oraz DOS (we wszystkich współczesnych odmianach),
- Mac OS (zarówno OS X, jak i Classic),
- BeOS, OS/2, VMS oraz QNX,
- systemy czasu rzeczywistego, jak VxWorks,
- superkomputery Cray oraz duże systemy typu mainframe, jak IBM,
- PDA z systemami operacyjnymi Palm OS, PocketPC i Linux,
- telefony komórkowe z systemami operacyjnymi Symbian oraz Windows Mobile,
- konsole do gier i iPodów.

Podobnie do samego interpretera języka, również standardowe moduły biblioteki Pythona zaimplementowane są w taki sposób, by były maksymalnie przenośne. Co więcej, programy w Pythonie są automatycznie komplikowane do przenośnego kodu bajtowego, który działa tak samo na dowolnej platformie z zainstalowaną zgodną wersją Pythona (więcej na ten temat w kolejnym rozdziale).

Oznacza to, że programy w Pythonie wykorzystujące fundamenty tego języka oraz standardowe biblioteki będą działały w ten sam sposób na systemach Linux, Windows i większości innych platform zawierających interpreter Pythona. Większość wersji Pythona zawiera również rozszerzenia dla określonej platformy (na przykład obsługę COM w systemie Windows), jednak samo jądro Pythona wraz z bibliotekami standardowymi działają wszędzie tak samo. Jak wspomniano wcześniej, Python zawiera również interfejs do zestawu narzędzi Tk GUI o nazwie

tkinter (w Pythonie 2.6 — Tkinter), który pozwala na implementowanie w programach napisanych w Pythonie pełnych graficznych interfejsów użytkownika, które działają na wszystkich najważniejszych platformach bez konieczności wprowadzania zmian do samego programu.

Ma duże możliwości

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

Typy dynamiczne

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

Automatyczne zarządzanie pamięcią

Python automatycznie alokuje obiekty i odzyskuje je (czyści pamięć), kiedy nie są już używane. Większość obiektów może rosnąć i kurczyć się na żądanie. Python śledzi wszystkie szczegóły związane z niskim poziomem pamięci, tak byśmy nie musieli tego robić sami.

Obsługa programowania dużych systemów

Python zawiera wiele narzędzi, takich jak moduły, klasy oraz wyjątki, które przydadzą się przy budowaniu większych systemów. Narzędzia te pozwalają na organizowanie systemów w komponenty, wykorzystywanie programowania zorientowanego obiektowo do ponownego wykorzystywania kodu i dostosowywania go do własnych potrzeb, a także radzenie sobie ze zdarzeniami i błędami.

Wbudowane typy obiektów

Python udostępnia najczęściej wykorzystywane struktury danych, takie jak listy, słowniki oraz łańcuchy znaków — są one nieodłączną częścią samego języka. Jak zostanie to nie-długo pokazane, są one zarówno elastyczne, jak i łatwe w użyciu. Wbudowane obiekty mogą na przykład rosnąć i kurczyć się na żądanie; mogą także być dowolnie zagnieźdzane, tak by reprezentować skomplikowane informacje.

Wbudowane narzędzia

Do przetwarzania tych wszystkich typów obiektów potrzebne są standardowe operacje, takie jak konkatenacja (łączenie kolekcji), wycinanie (ekstrakcja części), sortowanie, odwzorowywanie i inne. Są one częścią Pythona.

Wbudowane biblioteki narzędzi

Dla bardziej szczegółowych zadań Python zawiera również spory zbiór narzędzi, które obsługują wszystko — od dopasowywania wyrażeń regularnych po zagadnienia sieciowe. Po opanowaniu samego języka duża część użycania Pythona na poziomie aplikacji odbywa się z wykorzystaniem narzędzi z różnych bibliotek.

Narzędzia udostępniane przez programistów

Ponieważ Python należy do oprogramowania open source, programiści zachęcani są do udostępniania gotowych narzędzi, które obsługują zadania nieobejmowane przez narzędzia wbudowane. W Internecie można znaleźć darmowe narzędzia do obsługi technologii COM, CORBA, ORB, XML, grafiki czy dostępu do baz danych.

Pomimo tak dużej liczby możliwości oraz narzędzi Python zachowuje swoją nieskomplikowaną składnię i projekt. Dzięki temu jest narzędziem do programowania o dużych możliwościach, zachowującym użyteczność i prostotę języków skryptowych.

Można go łączyć z innymi językami

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

Łączenie Pythona z bibliotekami napisanymi w językach takich, jak C czy C++ sprawia, że Python staje się łatwym w użyciu językiem frontendu służącym do dostosowywania programów do własnych potrzeb. Jak wspomniano wcześniej, dzięki temu Python świetnie nadaje się do szybkiego tworzenia prototypów. Systemy można najpierw zaimplementować w Pythonie, by skorzystać z szybkości programowania w tym języku, a następnie przenieść je do języka C, fragment po fragmencie, zgodnie z wymaganiami w zakresie wydajności.

Jest łatwy w użyciu

By uruchomić program napisany w Pythonie, wystarczy go napisać i wykonać. Nie ma tutaj żadnych przejściowych etapów komplikacji czy łączenia, które występują w językach takich, jak C czy C++. Python natychmiast wykonuje programy, co sprawia, że programowanie w tym języku jest bardzo interaktywne, a wszelkie zmiany wprowadzane są bardzo szybko — w wielu przypadkach można być świadkiem efektu zmiany programu, która odbywa się na naszych oczach, w miarę pisania kodu.

Oczywiście długość cyklu programowania to tylko jeden z aspektów łatwości użycia Pythona. Język ten celowo zawiera prostą składnię i wbudowane narzędzia o dużych możliwościach. Niektórzy nazywają nawet Pythona „wykonwalnym pseudokodem”. Ponieważ Python eliminuje wiele ze stopnia skomplikowania innych narzędzi, programy napisane w tym języku są prostsze, mniejsze i bardziej elastyczne od swoich odpowiedników w językach takich, jak C, C++ czy Java.

Jest łatwy do nauczenia się

Powyższe hasło prowadzi nas bezpośrednio do kluczowego celu niniejszej książki — w porównaniu z innymi językami programowania Python jest łatwy do opanowania. Tak naprawdę można się spodziewać, że pierwsze programy w Pythonie zacznie się pisać już w kilka dni po rozpoczęciu nauki (a być może parę godzin, w przypadku doświadczonych programistów). To dobra wiadomość dla zawodowych programistów chcących się nauczyć języka, który będą

wykorzystywać w swojej pracy, a także dla użytkowników systemów, które udostępniają napisaną w Pythonie warstwę służącą do dostosowania systemu do własnych celów czy kontroliowania go.

Już dziś wiele systemów polega na fakcie, że użytkownicy mogą szybko nauczyć się wystarczająco dużo Pythona, by być w stanie na miejscu dostosować istniejące programy do własnych potrzeb z niewielkim wsparciem ze strony producenta (lub przy jego braku). Choć Python zawiera zaawansowane narzędzia programistyczne, samo jądro tego języka jest proste zarówno dla osób początkujących, jak i guru programowania.

Zawdzięcza swoją nazwę Monty Pythonowi

Jasne, trudno to nazwać techniczną mocną stroną, ale kwestia ta wydaje się zaskakująco dobrze ukrywanym sekretem, który chciałbym ujawnić. Nie można dać się zwieść mnóstwu różnych ikon z gadami, które pojawiają się w świecie Pythona. Tak naprawdę twórca Pythona, Guido van Rossum, nazwał język od serialu komediowego telewizji BBC zatytułowanego *Latający Cyrk Monty Pythona* (*Monty Python's Flying Circus*). On sam jest wielkim fanem Monty Pythona, podobnie jak wielu innych programistów (tak naprawdę wydaje się, że pomiędzy tymi dwoma polami istnieje swoista symetria).

Takie dziedzictwo bez wątpienia dodaje do przykładów napisanych w Pythonie nutkę humoru. Tradycyjne „foo” i „bar” jako nazwy zmiennych w Pythonie są na przykład zastępowane przez „spam” („mionanka”) i „eggs” („jajka”). Okazjonalnie pojawiają się też inne perelki, jak „Brian” czy „ni”. Wpływ widoczne są również w samej społeczności Pythona — odczyty na konferencjach poświęconych temu językowi często noszą miano „The Spanish Inquisition” („Hiszpańska inkwizycja”).²

Wszystko to jest oczywiście zabawne tylko wtedy, gdy zna się sam serial. Do zrozumienia przykładów czerpiących z Monty Pythona (w tym tych z niniejszej książki) nie jest jednak potrzebna znajomość tej serii. Dzięki powyższemu wyjaśnieniu każdy będzie jednak przynajmniej wiedział, skąd te odniesienia pochodzą.

Jak Python wygląda na tle innych języków?

Wreszcie, by umieścić Pythona w znany kontekście, wiele osób często porównuje go z innymi językami, takimi jak Perl, Tcl czy Java. Wydajność omówiliśmy już wcześniej, teraz skupmy się na funkcjonalności. Choć inne języki są przydatnymi narzędziami, wiele osób uważa, że Python:

- Ma większe możliwości od Tcl. Obsługa programowania dla dużych systemów sprawia, że nadaje się również do tego celu.
- Ma łatwiejszą składnię i prostszy projekt od Perla, co sprawia, że jest bardziej czytelny i łatwiejszy w utrzymywaniu, a tym samym mniej podatny na błędy.
- Jest prostszy i łatwiejszy w użyciu od Javy. Python jest językiem skryptowym, a Java dziedziczy wiele ze stopnia skomplikowania oraz składni języków systemowych, takich jak C++.

² W polskim wydaniu książki wykorzystano przede wszystkim tłumaczenia autorstwa Tomasza Beksińskiego, dostępne w serwisie Monty Python's Modrzew (<http://www.modrzew.stopklatka.pl>) — przyp. tłum.

- Jest prostszy i łatwiejszy w użyciu od C++, jednak często nie konkuje z tym językiem bezpośrednio. Jako język skryptowy Python zazwyczaj służy do innych celów.
- Ma o wiele większe możliwości i jest dostępny na większą liczbę platform niż Visual Basic. Jest też produktem open source, co oznacza, że nie jest kontrolowany przez jedną firmę.
- Jest bardziej czytelny i ogólny od PHP. Python jest czasami wykorzystywany do tworzenia stron internetowych, jednak równie powszechnie stosowany jest w prawie każdej innej dziedzinie informatyki, od robotyki po animację filmów.
- Jest bardziej dojrzały i ma bardziej czytelną składnię od języka Ruby. W przeciwnieństwie do Javy i Ruby'ego programowanie zorientowane obiektowo jest w Pythonie opcją — Python nie narzuca go użytkownikom czy projektom, w których nie ma ono zastosowania.
- Ma dynamiczny charakter języków takich, jak SmallTalk czy Lisp, ale do tego ma też prostą, tradycyjną składnię dostępną dla programistów oraz użytkowników systemów dostosowanych do własnych potrzeb.

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

Oczywiście autor niniejszej książki jest znanym promotorem Pythona od 1992 roku, dlatego te komentarze można potraktować w dowolny sposób. Odzwierciedlają one jednak doświadczenia współzielone przez wielu programistów, którzy poświęcili swój czas na poznanie tego, co oferuje Python.

Podsumowanie rozdziału

Na tym kończymy marketingową część niniejszej książki. W tym rozdziale omówiliśmy niektóre powody wybierania Pythona do zadań programistycznych. Dowiedzieliśmy się również, w jaki sposób się go stosuje, i zapoznaliśmy się z reprezentatywną próbką osób, które z niego dzisiaj korzystają. Moim celem jest jednak uczenie Pythona, a nie sprzedawanie go. Najlepszym sposobem oceny języka jest zobaczenie go w działaniu, dlatego pozostała część książki skupia się w całości na elementach języka, o których tutaj jedynie wspomnieliśmy.

Na początek dwa kolejne rozdziały stanowią techniczne wprowadzenie do Pythona. Dowiemy się z nich, jak uruchamia się programy napisane w Pythonie, przyjrzymy się modelowi wykonywania kodu bajtowego Pythona, a także wprowadzimy podstawy plików modułów, w których zapisuje się kod. Celem będzie dostarczenie każdemu wystarczającej ilości informacji, które pozwolą na uruchamianie przykładów i ćwiczeń z dalszej części książki. Samo programowanie zaczniemy dopiero w rozdziale 4. — najpierw należy opanować podstawy.

Sprawdź swoją wiedzę — quiz

W tym wydaniu książki każdy rozdział kończy się krótkim quizem dotyczącym zaprezentowanego materiału, co ma pomóc w powtórzeniu kluczowych koncepcji. Odpowiedzi na pytania z quizów znajdują się bezpośrednio pod nimi; zachęcam do zapoznania się z nimi po udzieleniu odpowiedzi na pytania. Poza quizami kończącymi rozdziały na końcu każdej części książki znajdują się ćwiczenia zaprojektowane tak, by każdy mógł zacząć samodzielne programowanie w Pythonie. A oto pierwszy test — powodzenia!

1. Należy podać sześć powodów, dla których ludzie wybierają Pythona?
2. Należy wymienić cztery znaczące firmy czy organizacje, które wykorzystują Pythona.
3. Dlaczego można *nie chcieć* używać Pythona w aplikacji?
4. Do czego można wykorzystać Pythona?
5. Na czym polega znaczenie instrukcji `import this` w Pythonie?
6. Dlaczego słowo „spam” („mielonka”) pojawia się w tak wielu przykładach kodu w Pythonie w książkach oraz Internecie?
7. Należy podać swój ulubiony kolor.

Sprawdź swoją wiedzę — odpowiedzi

I jak poszło? Poniżej znajdują się odpowiedzi, których udzieliłbym ja sam, choć w przypadku niektórych pytań z quizów poprawnych odpowiedzi może być więcej. Ponownie zachęcam do zapoznania się z odpowiedziami — nawet osoby, które są pewne, że odpowiedziały poprawnie. W rozwiązaniach często podaję dodatkowy kontekst. Jeśli któryś z nich jest niezrozumiałe, polecam wrócić do odpowiedniego fragmentu tekstu rozdziału.

1. Jakość oprogramowania, wydajność programistów, przenośność programu, dodatkowe biblioteki, integracja komponentów oraz po prostu zadowolenie. Z tych wszystkich opcji najczęstszymi powodami, dla których różne osoby wybierają Pythona, są jakość oraz wydajność.
2. Google, Industrial Light & Magic, EVE Online, Jet Propulsion Labs, Maya, ESRI i wiele innych. Prawie każda organizacja zajmująca się programowaniem wykorzystuje w jakimś stopniu Pythona — albo w strategicznym, długofalowym rozwijaniu produktu, albo w zadaniach krótkoterminowych, takich jak testowanie czy administrowanie systemami.
3. Słabą stroną Pythona jest jego wydajność — nie będzie on działał tak szybko, jak języki w pełni komplikowane, takie jak C czy C++. Z drugiej strony, Python jest wystarczająco szybki dla wielu zastosowań, a typowy kod w Pythonie działa z prędkością zbliżoną do języka C, ponieważ zawiera połączony kod w tym języku w swoim interpreterze. Jeśli jednak szybkość jest kwestią kluczową, dla najbardziej wymagających części aplikacji dostępne są skompilowane rozszerzenia.
4. Pythona można używać prawie do wszystkiego, co wykonuje się za pomocą komputera — od tworzenia stron internetowych i gier po robotykę i kontrolę statków kosmicznych.

5. Instrukcja `import this` wywołuje w Pythonie sztuczkę wyświetlającą pewne elementy filozofii będącej fundamentem projektu tego języka. W kolejnym rozdziale pokażemy, jak można wykonać tę instrukcję.
6. „Spam” („mielonka”) to odniesienie do sławnego skeczu *Latającego Cyrku Monty Pythona*, w którym ludzie próbujący zamówić w restauracji coś do jedzenia są zagłuszani przez chór Wikingów śpiewających o mielonce. Przy okazji jest to również popularna nazwa zmiennej stosowanej w skryptach napisanych w Pythonie...
7. Niebieski. Nie, żółty!

Python to inżynieria, nie sztuka

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

Krótko mówiąc: *w Pythonie można zrobić wszystko to, co w Perlu, a dodatkowo jeszcze po skończeniu da się przeczytać gotowy kod.* I tyle. Dziedziny zastosowania tych języków w dużej mierze się pokrywają, jednak Python jest bardziej zorientowany na tworzenie czytelnego kodu. Dla wielu osób zwiększała czytelność kodu Pythona przekłada się na możliwość ponownego wykorzystania tego kodu w przyszłości i łatwe jego utrzymywanie. Z tej przyczyny Python staje się lepszym wyborem dla programów, które nie są pisane raz, a potem zapomniane. Kod w Perlu łatwo jest napisać, jednak trudniej odczytać. Pamiętając o tym, że większość programów ma dużo dłuższy żywot, niż to na początku planujemy, wiele osób uważa Pythona za wydajniejsze narzędzie.

By nieco bardziej rozwinąć powyższe stwierdzenia, warto dodać, że oba języki odzwierciedlają doświadczenia i wykształcenie swoich projektantów, co pokazuje również, jakie są przyczyny wybierania Pythona przez niektóre osoby. Twórca Pythona jest matematykiem. Z tego powodu stworzył język z wysokim stopniem jednorodności — jego składnia i zestaw narzędzi są bardzo spójne. Tak jak matematyka — projekt tego języka jest ortogonalny, większa część języka pochodzi od niewielkiego zbioru podstawowych koncepcji. Przykładowo kiedy zrozumie się polimorfizm w Pythonie, cała reszta to tylko szczegóły.

Twórca Perla jest za to jazykoznawcą, a sam język znakomicie to odzwierciedla. W Perlu te same zadania można wykonać na wiele sposobów, konstrukcje tego języka wchodzą ze sobą w interakcje w sposób kontekstowy i czasami dość subtelny — podobnie do języków naturalnych. Dobrze znane motto Perla mówi: „Istnieje więcej niż jedna droga do zrobienia czegoś”. Dzięki takiemu projektowi zarówno sam język, jak i społeczność jego użytkowników od zawsze zachęcali do wolności wyrażania się w czasie tworzenia kodu. Kod napisany w Perlu przez jedną osobę może być radykalnie inny od kodu napisanego przez kogoś innego. Tak naprawdę pisanie unikalnego, podchwytliwego kodu jest często wśród użytkowników Perla powodem do dumy.

Jak przyzna każdy, kto miał do czynienia z koniecznością utrzymywania większych ilości kodu, wolność wyrażania się jest świetna w przypadku sztuki, jednak niekoniecznie w przypadku inżynierii. W inżynierii potrzebny jest nam minimalny zestaw narzędzi i przewidywalność.

W inżynierii wolność wyrażania się może prowadzić do koszmaru przy utrzymywaniu kodu. Jak zdradził mi niejeden użytkownik Perla, rezultatem zbyt dużej wolności jest często to, że kod o wiele łatwiej jest napisać od nowa, niż próbować modyfikować.

Pomyślmy o tym w taki sposób: kiedy ludzie tworzą obraz czy rzeźbę, robią to dla siebie samych, dla czysto estetycznych celów. Możliwość, że ktoś inny będzie kiedyś musiał zmieniać nasze dzieło, mało komu przychodzi do głowy. To właśnie kluczowa różnica między sztuką a inżynierią. Kiedy ludzie piszą oprogramowanie, nie robią tego dla siebie samych. Tak naprawdę nie robią tego nawet dla komputera. Dobrzy programiści wiedzą, że kod pisany jest dla kolejnych osób, które będą go musiały odczytywać, by z niego korzystać i go utrzymywać. Jeśli taka osoba nie będzie w stanie zrozumieć kodu, w rzeczywistych warunkach stanie się on bezużyteczny.

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

Na dłuższą metę sam nacisk na jakość kodu w Pythonie zwiększa wydajność programistów, a także ich zadowolenie. Oczywiście programiści tworzący kod w tym języku także mogą być kreatywni, a jak zobaczymy niebawem, Python również oferuje kilka rozwiązań niektórych problemów. Sam fundament Pythona zachęca jednak do dobrej pracy inżynierskiej, czego inne języki skryptowe często nie robią.

Powyższe argumenty to kwestie, z jakimi zgadza się wiele osób, które zdecydowały się na używanie Pythona. Każda osoba powinna jednak oczywiście wyciągnąć własne wnioski na podstawie tego, co może jej zaoferować Python. W celu zapoznania się z tą ofertą wystarczy przejść do kolejnego rozdziału książki.

Jak Python wykonuje programy?

Niniejszy rozdział wraz z kolejnym omawiają wykonywanie programów — sposób uruchamiania kodu oraz wykonywania go przez Pythona. W tym rozdziale skupimy się na interpreterze Pythona. W rozdziale 3. będzie można zobaczyć, jak można uruchamiać własne programy.

Początki są zależne od platformy, więc część materiału z tych dwóch rozdziałów może nie mieć zastosowania do platformy, na jakiej w rzeczywistości będziemy wykonywać swoją pracę, dlatego każdy może spokojnie ominąć części, które go nie dotyczą. Podobnie osoby bardziej zaawansowane, które w przeszłości korzystały już z podobnych narzędzi i wolą od razu przejść do opisu samego języka, mogą pominąć ten rozdział. Pozostałym osobom rozdział ten powinien pokazać, w jaki sposób wykonuje się kod programu.

Wprowadzenie do interpretera Pythona

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

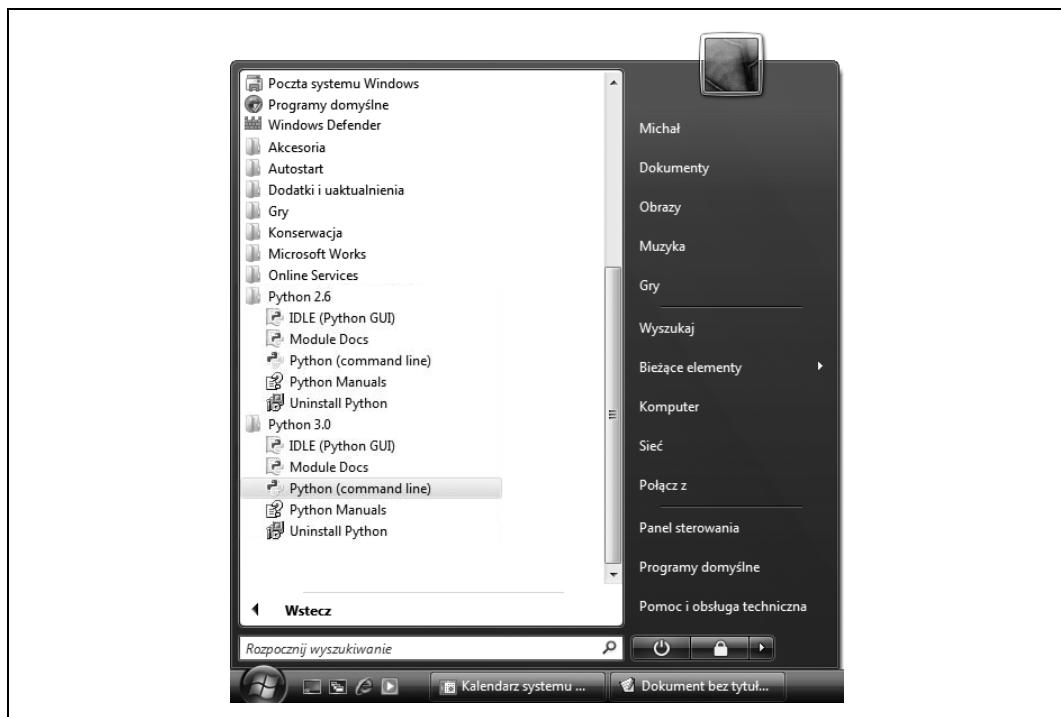
Kiedy na komputerze instalowany jest pakiet Pythona, generuje on pewną liczbę komponentów — przynajmniej interpreter oraz bibliotekę pomocniczą. W zależności od sposobu wykorzystania interpreter Pythona może przybrać postać programu wykonywalnego lub zbioru bibliotek połączonych z innym programem. W zależności od rodzaju wykorzystywanej implementacji Pythona sam interpreter może być programem w języku C, zbiorem klas Javy czy jeszcze czymś innym. Bez względu na formę kod w Pythonie musi zawsze być wykonywany przez interpreter. A żeby było to możliwe, konieczne jest zainstalowanie interpretera Pythona na własnym komputerze.

Szczegóły instalacji Pythona będą różne dla różnych platform i bardziej szczegółowo omówione są w dodatku A. W skrócie:

- Użytkownicy systemu Windows pobierają i uruchamiają instalacyjny plik wykonywalny umieszczający Pythona na ich komputerach. Wystarczy dwukrotnie kliknąć ten plik, a później na wszystkie pytania odpowiadać *Yes (Tak)* lub *Next (Dalej)*.

- Użytkownicy systemów Linux oraz Mac OS X najprawdopodobniej mają już Pythona zainstalowanego na swoich komputerach — jest on obecnie standardowym komponentem tych platform.
- Niektórzy użytkownicy Linuksa i Mac OS X (oraz większość użytkowników Unixa) samodzielnie komplilują Pythona z pakietu dystrybucyjnego z pełnym kodem źródłowym.
- Użytkownicy Linuksa mogą także znaleźć pliki RPM, natomiast dla użytkowników systemu Mac OS X dostępne są różne pakiety instalacyjne specyficzne dla Maca.
- Dla innych platform istnieją techniki instalacyjne odnoszące się tylko do nich. Python jest na przykład dostępny na telefony komórkowe, konsole do gier i iPody, jednak szczegółы instalacji mogą być bardzo różne.

Samego Pythona można pobrać z odpowiedniej podstrony witryny internetowej tego języka — <http://www.python.org>. Można go również uzyskać za pomocą różnych innych kanałów dystrybucji. Należy pamiętać, że przed instalacją zawsze należy sprawdzić, czy pakiet ten nie jest już zainstalowany na naszym komputerze. Osoby pracujące w systemie Windows zazwyczaj znajdują Pythona w menu *Start*, jak na rysunku 2.1 (poszczególne opcje menu omówione są w kolejnym rozdziale). W systemach Unix oraz Linux zazwyczaj można znaleźć Pythona w drzewie katalogu */usr*.



Rysunek 2.1. Po zainstalowaniu w systemie Windows Python widoczny jest w menu Start. W różnych wydaniach Pythona może to wyglądać nieco inaczej, jednak zazwyczaj IDLE służy do uruchamiania graficznego środowiska programistycznego, a opcja Python powoduje rozpoczęcie prostej sesji interaktywnej. W menu tym znajduje się również standardowa dokumentacja (Python Manuals), a także silnik dokumentacji Pydoc (Module Docs).

Ponieważ szczegóły implementacyjne tak bardzo różnią się między poszczególnymi platformami, nie będziemy się nad tym bardziej rozwodzić. Więcej informacji na temat procesu instalacji można znaleźć w dodatku A. Na potrzeby tego i kolejnego rozdziału zakładam, że każdy ma już zainstalowanego Pythona.

Wykonywanie programu

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

Z punktu widzenia programisty

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

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

Plik ten składa się z dwóch instrukcji Pythona o nazwie `print`, które służą do wyświetlenia (wydrukowania) łańcucha znaków (tekstu znajdującego się w apostrofach) oraz wyniku wyrażenia liczbowego (2 do potęgi 100) do strumienia wyjścia. W Pythonie w wersji 2.5 oraz starszych w razie konieczności użycia znaków narodowych spoza zbioru ASCII (czyli na przykład polskich znaków) konieczne było również umieszczenie komentarza określającego kodowanie znaków (UTF-8)¹ w postaci `# -*- coding: utf-8 -*-`. Nie należy się teraz przejmować składnią tego kodu — w tym rozdziale interesuje nas jedynie sposób wykonywania go. W dalszej części książki omówiona zostanie instrukcja `print` i wyjaśnione to, dlaczego można podnieść 2 do potęgi 100 bez przepelnienia.

Taki plik z instrukcjami można utworzyć w dowolnym edytorze tekstu. Zgodnie z konwencją pliki Pythona otrzymują nazwy kończące się rozszerzeniem `.py`. Z technicznego punktu widzenia taki schemat nazewnictwa jest wymagany jedynie dla plików importowanych, co zostanie omówione w dalszej części książki, jednak większość plików Pythona dla spójności nazywa się w ten sam sposób.

Po wpisaniu tych instrukcji do pliku tekstowego trzeba przekazać Pythonowi, że ma ten plik *wykonać* — co oznacza wykonanie wszystkich instrukcji z pliku od góry do dołu, jedna po drugiej. Jak zobaczymy w kolejnym rozdziale, pliki programów Pythona można wykonywać za pomocą wiersza poleceń powłoki, klikając ich ikony, uruchamiając je w zintegrowanym środowisku programistycznym (IDE), a także za pomocą standardowych technik. Jeśli wszystko pójdzie dobrze, po wykonaniu pliku gdzieś w komputerze zobaczymy wynik dwóch instrukcji `print` — domyślnie w tym samym oknie, w którym byliśmy, kiedy wykonywaliśmy program:

```
Witaj, świecie!  
1267650600228229401496703205376
```

¹ Komentarz z kodowaniem zgodny był z PEP 0263 i służył do podawania kodowania pliku źródłowego Pythona. Informacja ta wykorzystywana była następnie przez analizator składniowy Pythona do interpretacji pliku zgodnie z podanym kodowaniem. W ten sposób możliwe było zapisywanie na przykład polskich znaków bezpośrednio w kodzie — *przyp. tłum.*

A oto, co stało się, kiedy wykonałem ten skrypt z wiersza poleceń na laptopie z zainstalowanym systemem Windows (aplikacja *Wiersz poleceń* dostępna jest w menu *Akcesoria*), by upewnić się, że nie popełniłem żadnych błędów literowych:

```
C:\temp> python script0.py  
Witaj, świecie!  
1267650600228229401496703205376
```

Właśnie udało nam się wykonać skrypt wyświetlający łańcuch znaków oraz liczbę. Być może ten kod nie zapewniłby nam zwycięstwa w żadnym konkursie programistycznym, ale dla zrozumienia podstaw wykonywania programów będzie wystarczający.

Z punktu widzenia Pythona

Krótki opis z poprzedniego podrozdziału jest dość standardowy dla języków skryptowych i generalnie jest to najczęściej wszystko, co muszą wiedzieć programiści Pythona. Do plików tekstowych wpisujemy kod, a same pliki wykonujemy za pośrednictwem interpretera. Kiedy jednak dajemy Pythonowi sygnał „Naprzód!”, w środku dzieje się trochę więcej. Choć znajomość mechanizmów wewnętrznych Pythona nie jest ściśle wymagana do programowania w tym języku, podstawowe zrozumienie struktury wykonawczej Pythona może pomóc nam zobaczyć wykonywanie programów w szerszej perspektywie.

Kiedy nakazujemy Pythonowi uruchomić skrypt, zanim kod zacznie się wykonywać, Python przeprowadza kilka kroków. Kod ten jest bowiem najpierw komplilowany na tak zwany kod bajtowy (ang. *byte code*), a następnie przesyłany do czegoś o nazwie „maszyna wirtualna”.

Kompilacja kodu bajtowego

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

W poprzednim akapicie wspomniałem, że proces ten jest *prawie* całkowicie przed nami ukryty. Jeśli proces Pythona ma uprawnienia do zapisu na naszym komputerze, kod bajtowy programów zostanie zapisany w plikach z rozszerzeniem *.pyc* (*.pyc* oznacza skompilowane źródło *.py*). Te pliki zaczynają się pojawiać na naszym komputerze po wykonaniu kilku programów obok odpowiadających im plików źródłowych (w tych samych katalogach).

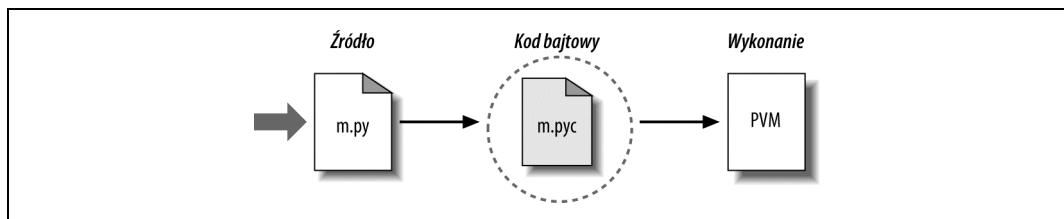
Python zapisuje taki kod bajtowy w celu optymalizacji szybkości wykonania. Następnym razem, kiedy będziemy wykonywać program, Python załaduje pliki *.pyc* i pominie etap komplikacji — o ile oczywiście nie zmieniliśmy kodu źródłowego od czasu, gdy kod bajtowy był zapisany ostatni raz. Python automatycznie sprawdza czas zapisu plików z kodem źródłowym oraz bajtowym, by wiadomo było, kiedy kod ten trzeba będzie skompilować ponownie. Jeśli kod źródłowy zapiszemy kolejny raz, przy następnym wykonywaniu automatycznie utworzony zostanie nowy kod bajtowy.

Jeśli Python nie może zapisać kodu bajtowego na naszym komputerze, program nadal będzie działał — kod bajtowy będzie tworzony w pamięci i po prostu usuwany po zakończeniu programu.² Ponieważ jednak pliki *.pyc* przyspieszają rozpoczęcie wykonywania programu, w przypadku większych aplikacji lepiej jest upewnić się, że są one zapisywane. Pliki kodu bajtowego to także jeden ze sposobów publikowania programów napisanych w Pythonie — Python z powodzeniem uruchomi program składający się z samych plików *.pyc*, nawet kiedy oryginalne pliki *.py* są nieobecne. Więcej informacji na temat innej opcji publikowania znajduje się w podrozdziale „Zamrożone pliki binarne”.

Maszyna wirtualna Pythona

Po skompilowaniu programu do kodu bajtowego (lub załadowaniu kodu bajtowego z istniejących plików *.pyc*) jest on przesyłany do wykonania do czegoś, co znane jest pod nazwą *maszyny wirtualnej Pythona* (Python Virtual Machine, PVM). „Maszyna wirtualna Pythona” brzmi bardzo poważnie, tak naprawdę jednak nie jest to ani oddzielnny program, ani coś, co musi być osobno instalowane. PVM to generalnie wielka pętla, która przechodzi przez instrukcje kodu bajtowego, jedna po drugiej, w celu wykonania działań i operacji z nimi związanych. Maszyna wirtualna Pythona jest silnikiem wykonawczym (ang. *runtime engine*) tego języka. Jest zawsze obecna jako część systemu Pythona i jest komponentem, który odpowiedzialny jest za samo wykonywanie skryptów. Z technicznego punktu widzenia jest ostatnim etapem interpretera Pythona.

Na rysunku 2.2 widać opisaną powyżej strukturę wykonawczą Pythona. Należy pamiętać, że cały ten stopień skomplikowania jest niewidoczny dla programistów. Kompilacja kodu bajtowego odbywa się automatycznie, a PVM jest częścią systemu Pythona, który instaluje się na komputerze. Programiści po prostu piszą kod i wykonują pliki składające się z instrukcji.



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

Wpływ na wydajność

Czytelnicy znający języki w pełni kompilowane, takie jak C czy C++, mogą zauważać kilka cech odróżniających je od modelu stosowanego w Pythonie. W Pythonie nieobecny jest etap budowania (ang. *build*) — kod uruchamiany jest natychmiast po napisaniu. Kod bajtowy Pythona nie jest również binarnym kodem maszynowym (na przykład w postaci instrukcji dla chipa firmy Intel). Kod bajtowy jest charakterystyczny dla Pythona.

² Kod bajtowy jest tak naprawdę zapisywany jedynie dla plików importowanych, a nie dla plików najwyższego poziomu programu. Importowanie zostanie omówione w rozdziale 3., a także w piątej części książki. Kod bajtowy nie jest również nigdy zapisywany dla kodu wpisywanego w wierszu poleceń, co także opisane jest w rozdziale 3.

Z powyższych przyczyn część kodu napisanego w Pythonie może nie działać tak szybko, jak kod w językach C czy C++, co opisano w rozdziale 1. To pętla maszyny wirtualnej Pythona, a nie chip procesora, musi zinterpretować kod bajtowy, a instrukcje kodu bajtowego wymagają więcej pracy od instrukcji procesora. Z drugiej strony, w przeciwieństwie do klasycznych interpreterów nadal istnieje etap komplikacji wewnętrznej — Python nie musi ciągle analizować i ponownie przetwarzanie każdej instrukcji w kodu źródłowego. Rezultat jest taki, że kod napisany w samym Pythonie działa z szybkością znajdującą się gdzieś pomiędzy szybkością tradycyjnych języków kompilowanych a szybkością tradycyjnych języków interpretowanych. Więcej informacji dotyczących wydajności Pythona znajduje się w rozdziale 1.

Wpływ na proces programowania

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

To wszystko sprawia, że cykl tworzenia oprogramowania jest o wiele krótszy. Nie istnieje konieczność wcześniejszego kompilowania i łączenia przed rozpoczęciem wykonywania. Wystarczy kod wpisać i wykonać. Powoduje to, że język ten jest bardzo dynamiczny — możliwe (i czasami bardzo wygodne) jest konstruowanie i wykonywanie przez programy Pythona innych programów Pythona w czasie wykonywania. Wbudowane funkcje `eval` oraz `exec` przyjmują i wykonują łańcuchy znaków zawierające kod Pythona. Taka struktura jest również przyczyną częstego wykorzystywania Pythona w dostosowywaniu produktów do własnych potrzeb — ponieważ kod Pythona może zmieniać się w locie, użytkownicy mogą modyfikować poszczególne części systemu na miejscu bez konieczności ponownego kompilowania kodu całego systemu.

Na bardziej podstawowym poziomie należy pamiętać, że w Pythonie istnieje tak naprawdę tylko faza *wykonywania* — nie ma początkowej fazy komplikacji, a wszystko dzieje się w trakcie działania programu. Ten model obejmuje również operacje, takie jak tworzenie funkcji czy klas oraz dodawanie modułów. Takie zdarzenia występują w językach bardziej statycznych przed wykonaniem, jednak w Pythonie mają miejsce wraz z wykonywaniem programu. Jak się niebawem okaże, rezultatem jest bardziej dynamiczne środowisko programistyczne, które może być nową jakością dla osób przyzwyczajonych do innych modeli.

Warianty modeli wykonywania

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

Alternatywne implementacje Pythona

W momencie pisania niniejszej książki istnieją trzy najważniejsze implementacje języka Python — *CPython*, *Jython* oraz *IronPython* — a także kilka implementacji pobocznych, takich jak *Stackless Python*. W skrócie mówiąc, CPython jest implementacją standardową, a wszystkie pozostałe mają swoje ściśle określone cele i role. Wszystkie implementują ten sam język, jednak w inny sposób wykonują programy.

CPython

Oryginalna i standardowa implementacja Pythona zazwyczaj nosi nazwę CPython, kiedy zesta-wia się ją z pozostałymi dwoma. Jej nazwa wzięła się od tego, że napisana jest w przenośnym kodzie w języku ANSI C. To jest właśnie Python, którego pobiera się ze strony <http://www.python.org> wraz z dystrybucjami ActivePython czy który instalowany jest automatycznie na większości komputerów z systemami Linux oraz Mac OS X. Jeśli na naszym komputerze znajduje się zainstalowana już wersja Pythona, najprawdopodobniej będzie to właśnie CPython — o ile oczywiście na przykład firma, w której pracujemy, nie wykorzystuje Pythona w jakiś inny, wyspecjalizowany sposób.

Jeśli nie jest nam potrzebne tworzenie w Pythonie skryptów dla aplikacji napisanych w Javie czy .NET, najprawdopodobniej będziemy korzystać ze standardowego systemu CPython. Ponieważ jest to referencyjna implementacja tego języka, najczęściej jest też najszybsza, najbardziej kompletna i ma największe możliwości w porównaniu z systemami alternatywnymi. Rysunek 2.2 odzwierciedla właśnie architekturę wykonywania tej implementacji.

Jython

System Jython (wcześniej znany jako JPython) jest alternatywną implementacją języka Python skierowaną na integrację z językiem programowania Java. Jython składa się z klas Javy kom-pilujących kod źródłowy Pythona do kodu bajtowego Javy, a następnie przekierowujących wynikowy kod bajtowy do maszyny wirtualnej Javy (Java Virtual Machine, JVM). Programiści nadal zapisują instrukcje Pythona w plikach tekstowych z rozszerzeniem *.py*, natomiast system Jython zastępuje środkową i prawą część rysunku 2.2 odpowiednikami z języka Java.

Celem systemu Jython jest pozwolenie na tworzenie w Pythonie skryptów dla aplikacji napi-sanych w Javie, podobnie do sposobu umożliwienia pisania skryptów dla aplikacji w języ-kach C czy C++ za pomocą implementacji CPython. Integracja tego systemu z Javą jest nie-zauważalna. Ponieważ kod napisany w Pythonie przekładany jest na kod bajtowy Javy, wygląda on i zachowuje się w czasie wykonywania zupełnie jak prawdziwy program napi-sany w Javie. Skrypty Jytthona mogą służyć jako aplenty webowe oraz serwlety czy tworzyć GUI oparte na Javie. Co więcej, Jython umożliwia również importowanie i wykorzystywanie klas Javy w kodzie napisanym w Pythonie w podobny sposób, jakby były one w Pythonie. Ponieważ jednak Jython jest wolniejszy i ma mniej możliwości od standardowej implemen-tacji CPython, zazwyczaj postrzegany jest jako narzędzie interesujące przede wszystkim dla programistów Javy szukających języka skryptowego, który będzie mógł być frontendem dla kodu napisanego w Javie.

IronPython

Trzecią i nowszą od systemów CPython oraz Jython implementacją Pythona jest IronPython. Został on zaprojektowany z myślą o pozwoleniu na integrację programów napisanych w Pythonie z aplikacjami stworzonymi dla platformy .NET firmy Microsoft przeznaczonej dla systemu Windows, a także dla jej odpowiednika dla Linuksa na licencji open source o nazwie Mono. Platforma .NET wraz ze swoim środowiskiem uruchomieniowym zaprojektowana jest w taki sposób, by stanowić warstwę niezależną od języka, podobnie do wcześniejszego modelu COM firmy Microsoft. IronPython pozwala programom napisanym w Pythonie na działanie zarówno jako komponenty klienta, jak i serwera dostępne z innych języków .NET.

W implementacji IronPython jest bardzo podobny do systemu Jython (i tak naprawdę tworzony był przez tę samą osobę) — zastępuje on środkową i prawą część rysunku 2.2 odpowiednikami pozwalającymi na wykonywanie w środowisku .NET. Podobnie jak Jython, IronPython ma szczególne przeznaczenie — jest interesujący przede wszystkim dla programistów zajmujących się integracją Pythona z komponentami .NET. Ponieważ implementacja ta rozwijana jest przez firmę Microsoft, IronPython może być w stanie korzystać z istotnych narzędzi optymalizacyjnych, które będą poprawiały jego wydajność. IronPython jest w momencie pisania niniejszej książki ciągle w fazie rozwoju. Więcej szczegółowych informacji na ten temat można znaleźć w Internecie lub na stronach poświęconych Pythonowi.³

Narzędzia do optymalizacji wykonywania

CPython, Jython oraz IronPython implementują język programowania Python w podobny sposób — kompilując kod źródłowy do kodu bajtowego i wykonując kod bajtowy na odpo-wiedniej maszynie wirtualnej. Inne systemy, takie jak kompilator JIT (ang. *just-in-time*) Psyco czy translator Shedskin C++, próbują zamiast tego zoptymalizować podstawowy model wyko-nywania. Na tym etapie zaznajamiania się z Pythonem znajomość tych narzędzi nie jest koniecz-na, jednak krótkie przyjrzenie się ich miejscu w modelu wykonywania może pomóc nam w lepszym zrozumieniu samego modelu.

Kompilator JIT Psyco

System Psyco nie jest kolejną implementacją Pythona, ale raczej komponentem rozszerzającym model wykonywania kodu bajtowego, który sprawia, że programy mogą działać szybciej. W kategoriach rysunku 2.2 Psyco jest ulepszeniem maszyny wirtualnej Pythona, które zbiera i wykorzystuje informacje o typach w czasie, gdy program działa, w celu przełożenia części kodu bajtowego programu na prawdziwy kod maszynowy, tak by przyspieszyć wykonywanie. Psyco wykonuje proces translacji bez wymagania zmian w kodzie czy osobnego etapu kom-pilacji w czasie tworzenia programu.

W skrócie, kiedy program zostaje uruchomiony, Psyco zbiera informacje o rodzajach obiektów, jakie są w nim przekazywane. Te informacje mogą zostać wykorzystane do wygene-

³ Jython i IronPython są zupełnie niezależnymi implementacjami Pythona kompilującymi kod źródłowy Pythona na potrzeby innych architektur wykonywania. Z programów napisanych w standardowej implementacji CPython również można uzyskać dostęp do programów napisanych dla platform Java czy .NET. Przykładowo systemy JPype i Python for .NET pozwalają na wywoływanie komponentów Javy czy .NET z kodu standardowego Pythona.

rowania bardzo wydajnego kodu maszynowego dostosowanego do tych typów obiektów. Po wygenerowaniu kod maszynowy zastępuje odpowiadającą mu część oryginalnego kodu bajtowego, co przyspiesza całkowitą szybkość wykonywania programu. Dzięki Psyco program z czasem — w trakcie działania — staje się szybszy. W idealnych przypadkach część kodu w Pythonie może dzięki Psyco stać się tak szybka, jak skompilowany kod w języku C.

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

Zostało udowodnione, że Psyco jest w stanie dramatycznie zwiększyć prędkość działania programów napisanych w Pythonie. Zgodnie z informacjami z witryny internetowej tego programu Psyco umożliwia przyspieszenie od dwu- do stu-krotnego — zazwyczaj cztero-krotne — w przypadku niezmodyfikowanego interpretera Pythona i niezmodyfikowanego kodu źródłowego, a jedynie z użyciem ładowanego dynamicznie modułu rozszerzenia w języku C. Co równie ważne, największe przyspieszenie można zaobserwować dla kodu algorytmicznego napisanego w czystym Pythonie — czyli właśnie tego rodzaju kodu, jaki normalnie przenosi się do języka C w celu jego optymalizacji. Dzięki Psyco taka migracja kodu traci na znaczeniu.

Psyco nie jest standardową częścią Pythona — system ten trzeba pobrać i zainstalować osobno. Nadal jest on swego rodzaju projektem badawczym, którego rozwój można na bieżąco śledzić w Internecie. W momencie pisania niniejszej książki Psyco można pobrać i zainstalować jako osobny produkt, choć wydaje się, że duża część tego systemu może zostać włączona do nowszego projektu PyPy będącego wysiłkiem na rzecz implementacji maszyny wirtualnej Pythona na nowo w kodzie Pythona w celu lepszego obsługiwanego optymalizacji, takich jak Psyco.

Chyba największą wadą Psyco jest to, że aktualnie generuje on kod maszynowy jedynie dla chipów o architekturze x86 firmy Intel, nawet jeśli w kategorii tej mieszą się systemy Windows, Linux i nowsze Macintoshe. Więcej informacji na temat tego rozszerzenia, a także innych systemów JIT można znaleźć na stronie <http://www.python.org>. Informacje o Psyco można również znaleźć na oficjalnej stronie tego systemu, która aktualnie znajduje się pod adresem <http://psyco.sourceforge.net>.

Translator Shedskin C++

Shedskin to nowy system, który podchodzi do wykonywania programów w Pythonie w inny sposób — próbuje przełożyć kod źródłowy Pythona na kod języka C++, który kompilator C++ danego komputera kompiluje następnie do kodu maszynowego. Reprezentuje on podejście do wykonywania kodu Pythona niezależne od platformy. W chwili pisania niniejszej książki Shedskin jest w fazie eksperymentalnej i nakłada na programy Pythona ograniczenia w zakresie typów statycznych, co w Pythonie nie jest normalnym zachowaniem, dlatego nie będziemy tego rozwiązania omawiać bardziej szczegółowo. Wstępne rezultaty wskazują jednak, że system ten ma potencjał, aby prześcignąć zarówno standardowego Pythona, jak i rozszerzenia Psyco, jeśli chodzi o szybkość wykonywania. Jest to bardzo obiecujący projekt. Szczegółowe informacje na temat jego aktualnego statusu można znaleźć w Internecie.

Zamrożone pliki binarne

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

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

Dzisiaj istnieją trzy najważniejsze systemy będące w stanie generować zamrożone pliki binarne — *py2exe* (dla systemu Windows), *PyInstaller* (podobny do py2exe, jednak działa również w systemach Linux oraz Unix i potrafi generować samoinstalujące się pliki binarne), a także *freeze* (oryginalny produkt). Narzędzia te trzeba pobrać niezależnie od Pythona, jednak są one darmowe. Są ciągle rozwijane, dlatego warto śledzić ich postęp na stronie <http://www.python.org> lub za pośrednictwem ulubionej wyszukiarki internetowej. By pokazać, jaki może być zakres przydatności tych systemów, wystarczy wspomnieć, że py2exe może „zamrozić” samodzielne programy wykorzystujące biblioteki Tkinter, PMW, wxPython oraz PyGTK GUI, a także programy korzystające z zestawu narzędzi programistycznych pygame czy programów klienta win32com.

Zamrożone pliki binarne nie są tym samym co dane wyjściowe z prawdziwego kompilatora — wykonują one kod bajtowy za pomocą maszyny wirtualnej. Dlatego z wyjątkiem możliwej poprawy błędów na początku, działają one z szybkością podobną do oryginalnych plików źródłowych. Zamrożone pliki binarne nie są małe (zawierają maszynę wirtualną Pythona), ale nie są też szczególnie duże — jak na dzisiejsze standardy. Ponieważ Python jednak osadzony jest w zamrożonych plikach binarnych, nie musi on być zainstalowany na maszynie docelowej, by móc wykonać program. Co więcej, ponieważ sam kod osadzony jest w pakiecie, jest on w rezultacie lepiej ukryty przed odbiorcą.

Taki schemat wykorzystujący pojedyncze pliki (pakiety) jest szczególnie atrakcyjny dla twórców oprogramowania komercyjnego. Program interfejsu użytkownika oparty na zestawie narzędzi Tkinter można na przykład zamrozić w pliku wykonywalnym i udostępniać jako samodzielny program na płycie CD czy w Internecie. Użytkownicy nie muszą instalować samego Pythona (czy w ogóle wiedzieć o jego istnieniu), żeby uruchomić udostępniany w ten sposób program.

Inne opcje wykonywania

Inne sposoby wykonywania programów napisanych w Pythonie mają nieco bardziej wyspecjalizowane cele:

- System *Stackless Python* to odmiana standardowej implementacji CPython, która nie zapisała stanu na stosie wywołań języka C. W ten sposób Pythona łatwiej jest przenieść na architektury z małymi stosami. Stackless Python udostępnia również opcje wydajnego wieloprzetwarzania, co otwiera nowe możliwości programistyczne, na przykład w postaci procedur współbieżnych.
- System *Cython* (oparty na pracy zespołu projektu *Pyrex*) to język hybrydowy łączący kod w Pythonie z możliwością wywoływanego funkcji języka C i wykorzystywania deklaracji typów z C dla zmiennych, parametrów oraz atrybutów klas. Kod napisany w Cythonie można skompilować do kodu języka C wykorzystującego API Python/C, który można następnie całkowicie skompilować. Choć Cython nie jest w pełni zgodny ze standardowym Pythonem, może się przydać do opakowywania zewnętrznych bibliotek języka C oraz tworzenia wydajnych rozszerzeń dla Pythona w C.

Szczegółowe i aktualne informacje na temat tych systemów można znaleźć za pomocą wyszukiwarki internetowej.

Przyszłe możliwości?

Wreszcie warto wspomnieć o tym, że omówiony tutaj model wykonywania Pythona jest tak naprawdę charakterystyczny dla aktualnej implementacji Pythona, a nie dla samego języka. Całkiem możliwe, że w czasie życia niniejszej książki pojawi się pełny, tradycyjny kompilator przekładający kod źródłowy Pythona na kod maszynowy (choć nie stało się tak w czasie ostatnich dwóch dziesięcioleci). W przyszłości mogą zostać przyjęte nowe formaty kodu bajtowego czy warianty implementacyjne. Na przykład:

- Projekt *Parrot* próbuje udostępnić wspólny format kodu bajtowego, maszyny wirtualnej i technik optymalizacyjnych dla różnych języków programowania (więcej informacji na ten temat można znaleźć na stronie <http://www.python.org>). Własna maszyna wirtualna Pythona wykonuje kod napisany w tym języku w sposób bardziej wydajny od systemu Parrot, jednak nie wiadomo, jak Parrot wyewoluje.
- Projekt *PyPy* jest wysiłkiem na rzecz nowej implementacji samej maszyny wirtualnej w języku Python, co powinno pozwolić na nowe techniki implementacyjne. Jego celem jest utworzenie szybkiej i elastycznej implementacji Pythona.
- Sponsorowany przez firmę Google projekt *Unladen Swallow* ma na celu co najmniej pięciokrotnie przyspieszenie standardowego Pythona, tak by język ten stał się na tyle szybki, aby móc w wielu kontekstach zastąpić C. Jest on zoptymalizowanym rozgałęzieniem implementacji CPython, które w zamierzeniach ma być w pełni zgodne z tym projektem, a jednocześnie znacznie szybsze. Projekt ten ma także nadzieję usunąć mechanizm wielowątkowości Global Interpreter Lock (GIL), który uniemożliwia prawdziwe nakładanie się wątków Pythona w czasie. Unladen Swallow znajduje się w początkowej fazie i jest twoorzony przez inżynierów Google jako projekt z dziedziny open source. Koncentruje się na Pythonie 2.6, jednak zmiany mogą zostać wprowadzone również do wersji 3.0. Wszelkie aktualne informacje na ten temat można znaleźć za pomocą wyszukiwarki.

Choć przyszłe schematy implementacyjne mogą w jakimś stopniu zmienić strukturę wykonywania Pythona, wydaje się, że kompilator kodu bajtowego przez jakiś czas pozostanie jeszcze standardem. Co więcej, dodawanie deklaracji ograniczających typy w celu obsługi komplikacji

statycznej zniszczyłoby elastyczność, zwięzłość, prostotę i ogólnego ducha Pythona. Ze względu na wysoce dynamiczną naturę Pythona wszystkie przyszłe implementacje będą raczej zawierały dziedzictwo dzisiejszej maszyny wirtualnej Pythona.

Podsumowanie rozdziału

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

Sprawdź swoją wiedzę — quiz

1. Co to jest interpreter Pythona?
2. Co to jest kod źródłowy?
3. Co to jest kod bajtowy?
4. Co to jest PVM?
5. Należy podać dwie nazwy wariantów standardowego modelu wykonywania Pythona.
6. Czym różnią się od siebie CPython, Jython oraz IronPython?

Sprawdź swoją wiedzę — odpowiedzi

1. Interpreter Pythona to program wykonujący napisane przez nas w Pythonie programy.
2. Kod źródłowy to instrukcje, które pisze się na potrzeby programu. Składa się on z tekstu umieszczonego w plikach tekstowych, które zazwyczaj kończą się rozszerzeniem `.py`.
3. Kod bajtowy to niskopoziomowa postać programu po skompilowaniu przez Pythona. Python automatycznie przechowuje kod bajtowy w plikach z rozszerzeniem `.pyc`.
4. PVM to maszyna wirtualna Pythona (skrót pochodzi od angielskiego *Python Virtual Machine*) — silnik wykonawczy Pythona, który interpretuje nasz skompilowany kod.
5. Psyco, Shedskin i zamrożone pliki binarne to różne alternatywy dla modelu wykonywania Pythona.
6. CPython to standardowa implementacja języka. Jython oraz IronPython implementują programy Pythona w taki sposób, by mogły one być wykorzystywane, odpowiednio, w środowiskach Java i .NET. Są alternatywnymi kompilatorami Pythona.

Jak wykonuje się programy?

Czas zacząć wykonywać jakiś kod. Skoro wiemy już nieco o wykonywaniu programów, możemy wreszcie zabrać się za prawdziwe programowanie w Pythonie. Od teraz zakładam, że każdy ma na swoim komputerze zainstalowanego Pythona. Jeśli tak nie jest, należy zatrzymać się tutaj i skorzystać z dodatku A — wskazówki dotyczące instalacji i konfiguracji można znaleźć właśnie tam.

Istnieje kilka różnych sposobów nakazania Pythonowi, by wykonał wpisywany przez nas kod. W niniejszym rozdziale omówimy wszystkie techniki uruchamiania programów będące obecnie w powszechnym użyciu. Nauczymy się, w jaki sposób *interaktywnie* wpisywać kod i jak zapisywać ten kod w plikach, tak by można go było uruchamiać za pomocą systemowego wiersza poleceń, kliknięcia ikony, importowania i przeładowywania modułów, wywołań exec oraz opcji z menu graficznych interfejsów użytkownika, takich jak IDLE.

Jeśli ktoś chce się po prostu dowiedzieć, jak szybko uruchomić program napisany w Pythonie, może się skusić na odczytanie jedynie części tego rozdziału odnoszących się do swojej własnej konfiguracji i przejść bezpośrednio do rozdziału 4. Nie warto jednak pomijać części poświęconych importowaniu modułów, ponieważ są one niezbędne do zrozumienia architektury programów Pythona. Każdego zachęcam również do przejrzenia przynajmniej części dotyczących IDLE i innych IDE, by jasne było, jakie narzędzia przydadzą się nam, kiedy zaczniemy pisać bardziej wyszukane programy.

Interaktywny wiersz poleceń

Chyba najprostszym sposobem wykonywania programów w Pythonie jest wpisywanie ich bezpośrednio do *interaktywnego wiersza poleceń* Pythona. Ten wiersz poleceń można uruchomić na wiele różnych sposobów — na przykład w IDE czy z konsoli systemowej. Zakładając, że interpreter zainstalowany jest w naszym systemie jako program wykonywalny, najbardziej neutralnym dla platformy sposobem rozpoczęcia sesji interpretera interaktywnego jest zazwyczaj wpisanie słowa `python` w wierszu poleceń systemu operacyjnego, bez żadnych argumentów. Na przykład:

```
% python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Wpisanie słowa `python` w wierszu poleceń powłoki, jak powyżej, rozpoczyna interaktywną sesję Pythona (znak `%` na początku kodu zastępuje w niniejszej książce znak zachęty systemu, nie jest elementem, który należy wpisać). Samo pojęcie wiersza poleceń powłoki jest dość uniwersalne, choć dostęp do niego może się różnić dla poszczególnych platform.

- W systemie Windows słowo `python` można wpisać w oknie konsoli DOS (zwanej *Wierszem poleceń*; zazwyczaj program ten można znaleźć pod opcją *Akcesoria* w menu *Start/Wszystkie programy*) lub korzystając z okna dialogowego *Start/Uruchom....*
- W systemach Unix, Linux oraz Mac OS X można wpisać to polecenie w powloce lub oknie terminala (na przykład w *xterm* lub konsoli powłoki, takiej jak *ksh* czy *csh*).
- Inne systemy mogą wykorzystywać podobne sposoby lub używać do tego celu własnych narzędzi. Urządzenia przenośne do uruchomienia sesji interaktywnej wymagają zazwyczaj kliknięcia ikony Pythona w oknie aplikacji.

Jeśli zmienna środowiskowa powłoki `PATH` nie zawiera katalogu instalacyjnego Pythona, słowo `python` trzeba będzie zastąpić pełną ścieżką do pliku wykonywalnego Pythona znajdującego się na naszym komputerze. W Uniksie i Linuksie oraz podobnych systemach często wystarczy `/usr/local/bin/python` lub `/usr/bin/python`. W systemie Windows można spróbować wpisać `C:\Python30\python` (w przypadku wersji 3.0).

```
C:\misc> c:\python30\python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Alternatywnie można przed wpisaniem słowa `python` wykorzystać polecenie zmieniające katalog, tak by wejść od razu do katalogu instalacyjnego Pythona (w systemie Windows można skorzystać z polecenia `cd c:\python30`):

```
C:\misc> cd C:\Python30
C:\Python30> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

W systemie Windows poza wpisaniem słowa `python` w oknie powłoki można także rozpoczęć sesję interaktywną, uruchamiając okno główne IDLE (omówione później) lub wybierając z wpisu dla Pythona w menu *Start* opcję *Python (command line)*, jak na rysunku 2.1 w rozdziale 2. Obie opcje uruchamiają sesję interaktywną Pythona o podobnej funkcjonalności; wpisywanie polecenia powłoki nie jest wymagane.

Interaktywne wykonywanie kodu

Bez względu na sposób uruchomienia sesja interaktywna Pythona rozpoczyna się od wpisania dwóch wierszy z tekstem informacyjnym (które zostaną usunięte z większości pozostałyzych przykładów z książki w celu zaoszczędzenia miejsca). Później użytkownik zachęcany jest do wpisania nowej instrukcji czy wyrażenia Pythona za pomocą znaków `>>>`. W sesji interaktywnej wynik wykonania kodu wyświetlany jest natychmiast po wierszach ze znakami `>>>` po naciśnięciu przycisku *Enter*.

Poniżej znajdują się na przykład wyniki dwóch instrukcji `print` (`print` jest tak naprawdę w Pythonie 3.0 wywołaniem funkcji, jednak w wersji 2.6 tak nie jest, stąd nawiasy wymagane są jedynie w 3.0>):

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

Nadal jeszcze nie ma co się przejmować szczegółami instrukcji `print` (składnią zajmiemy się w kolejnym rozdziale). W skrócie, instrukcje te wypisują łańcuch znaków Pythona oraz liczbę, co widać w wierszach z danymi wyjściowymi, które następują po każdym wierszu z danymi wejściowymi (`>>>`). `2 ** 8` oznacza w Pythonie 2 podniesione do potęgi 8.

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

```
>>> drwal = 'git'
>>> drwal
'git'
>>> 2 ** 8
256
>>>
% <== Żeby wyjść z sesji, użyj Ctrl+D (Unix) lub Ctrl+Z (Windows)
```

W powyższym kodzie pierwszy wiersz zapisuje wartość, przypisując ją do zmiennej. Dwa ostatnie wpisane wiersze (`drwal` oraz `2 ** 8`) są wyrażeniami, a ich wyniki wyświetlane są automatycznie. By wyjść z sesji interaktywnej i powrócić do wiersza poleceń powłoki, należy na komputerach z systemem Unix użyć skrótu klawiaturowego `Ctrl+D`, natomiast w systemach MS DOS i Windows — `Ctrl+Z`. W IDLE należy albo skorzystać ze skrótu `Ctrl+D`, albo po prostu zamknąć okno.

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

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

Do czego służy sesja interaktywna?

Sesja interaktywna wykonuje kod i zwraca wyniki w miarę wpisywania kodu, jednak nie zapisuje kodu w pliku. Choć oznacza to, że większość kodu nie wpisuje się w sesji interaktywnej, jest to świetna opcja do eksperymentowania z językiem i testowania plików programów w locie.

Eksperymentowanie

Ponieważ kod jest tam wykonywany natychmiast, sesja interaktywna jest idealnym miejscem na eksperymenty z językiem. W książce będzie często wykorzystywana do demonstrowania krótkich przykładów. Tak naprawdę to pierwsza reguła, jaką należy zapamiętać: jeśli mamy

jakiekolwiek wątpliwości co do tego, jak działa fragment kodu w Pythonie, wystarczy rozpoczęć sesję interaktywną i sprawdzić, co się stanie.

Załóżmy na przykład, że czytamy kod programu napisanego w Pythonie i natrafiamy na wyrażenie takie jak 'Mielonka!' * 8, którego znaczenia nie rozumiemy. W tym momencie możemy poświęcić dziesięć minut na przedzieranie się przez dokumentację i książki w celu dowiedzenia się, co robi ten kod, lub zamiast tego możemy po prostu wykonać go w sesji interaktywnej:

```
>>> 'Mielonka!' * 8 <== Nauka przez próbowanie  
'Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!'
```

Natychmiastowe informacje zwrotne otrzymywane w sesji interaktywnej to najczęściej najszyszybszy sposób na nauczenie się, co robi jakiś fragment kodu. Powyżej jasne staje się, że kod ten powtarza łańcuch znaków. W Pythonie znak * dla liczb oznacza mnożenie, jednak w przypadku łańcuchów znaków — powtarzanie. Przypomina to konkatenację łańcucha znaków do samego siebie wiele razy (więcej informacji o łańcuchach znaków znajduje się w rozdziale 4.).

Istnieje szansa, że eksperymentując w ten sposób, niczego nie zepsujemy — przynajmniej na razie. By wyrządzić jakieś prawdziwe szkody, na przykład usunąć pliki czy wykonać poleceńa powłoki, trzeba się naprawdę postarać i importować moduły w jawnym sposób. Zanim ten sposób stanie się naprawdę niebezpieczny, musielibyśmy poznać też nieco więcej informacji na temat interfejsów systemowych. Zwykły kod w Pythonie prawie zawsze można bezpiecznie wykonać.

Zobaczmy na przykład, co się stanie, kiedy w sesji interaktywnej popełnimy błąd:

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

W Pythonie użycie zmiennej przed przypisaniem do niej wartości zawsze jest błędem (w przeciwnym razie, gdyby zmienne były wypełniane za pomocą wartości domyślnych, pewne błędy mogłyby pozostać niezauważone). Później dowiemy się więcej na ten temat; ważne jest jednak to, że robiąc tego typu błędy, nie powodujemy zakończenia działania Pythona czy komputera. Zamiast tego otrzymujemy komunikat o błędzie wskazujący miejsce pomyłki wraz z wierszem kodu i możemy kontynuować sesję interaktywną czy skrypt. Po zapoznaniu się z Pythonem komunikaty o błędach wyświetlane przez ten język często w zupełności wystarczą do debugowania kodu (więcej informacji o debugowaniu znajduje się w ramce „Debugowanie kodu w Pythonie” na końcu rozdziału).

Testowanie

Oprócz służenia jako narzędzie do eksperymentowania w trakcie nauki języka interpreter interaktywny jest idealnym sposobem testowania kodu zapisanego w plikach. Pliki modułów można zaimportować interaktywnie; można również testować narzędzia, które pliki te definiują, wpisując kod je wywołujący w sesji interaktywnej.

Poniższy kod testuje na przykład funkcję w gotowym module udostępnianym wraz z Pythonem w jego bibliotece standardowej (wyświetla ona nazwę katalogu, w którym się aktualnie pracuje), jednak to samo będziemy mogli zrobić dla własnych plików modułów, kiedy już zaczniemy je tworzyć:

```
>>> import os  
>>> os.getcwd() <== Testowanie w locie  
'c:\\\\Python30'
```

Mówiąc bardziej ogólnie, w sesji interaktywnej można testować komponenty programu bez względu na ich źródło — można w niej importować i testować funkcje oraz klasy z własnych plików napisanych w Pythonie, wpisać wywołania do połączonych funkcji języka C czy korzystać z klas Javy pod Jythonem. Po części ze względu na interaktywną naturę Python obsługuje eksperymentalny i eksploracyjny styl programowania, który przyda się na początku przygody z tym językiem.

Wykorzystywanie sesji interaktywnej

Choć sesja interaktywna jest prosta w użyciu, zwłaszcza osoby początkujące powinny wziąć pod uwagę następujące wskazówki:

- **Należy wpisywać jedynie polecenia Pythona.** Przede wszystkim trzeba pamiętać o tym, że w sesji interaktywnej Pythona wpisuje się jedynie polecenia w tym języku, a nie wszelkie polecenia systemowe. Istnieją sposoby na wykonywanie poleceń systemowych w środku kodu Pythona (na przykład za pomocą `os.system`), ale nie są one tak proste i bezpośrednie, jak wpisanie poleceń od razu po znaku zachęty Pythona.
- **Instrukcje `print` wymagane są jedynie w plikach.** Ponieważ interpreter interaktywny automatycznie wyświetla wyniki wyrażeń, wpisywanie instrukcji `print` w sesji interaktywnej nie jest konieczne. Jest to miła opcja, ale często myląca dla użytkowników, którzy przechodzą do zapisywania kodu w plikach — w pliku z kodem, żeby zobaczyć dane wyjściowe, niezbędne jest użycie instrukcji `print`, ponieważ wyniki wyrażeń nie są zwracane automatycznie. Trzeba pamiętać, że instrukcja `print` jest niezbędna w plikach, natomiast w sesji interaktywnej już nie.
- **Nie należy (na razie) wcinać kodu wpisywanego w sesji interaktywnej.** Kiedy pisze się programy w Pythonie, obojętnie, czy interaktywnie, czy w pliku, należy pamiętać o rozpoznaniu wszystkich niezagieźdzonych instrukcji w pierwszej kolumnie (to znaczy jak najbardziej po lewej stronie). Jeśli tak się nie zrobi, Python może zwrócić komunikat o błędzie składni (`SyntaxError`), ponieważ odstęp po lewej stronie kodu uznawany jest za indentację grupującą instrukcje zagnieźdzoną. Aż do rozdziału 10. wszystkie pisane przez nas instrukcje będą niezagieźdzoną, dlatego taka informacja powinna nam na razie wystarczyć. Przy pierwszych eksperymentach z Pythonem kwestia ta nieodmiennie wprowadza użytkowników w zakłopotanie. Należy więc zapamiętać: początkowa spacja generuje komunikat o błędzie.
- **Należy uważać na zmianę znaku zachęty w instrukcjach złożonych.** Instrukcje *złożone* (wielowierszowe) pojawią się dopiero w rozdziale 4., a na powaźnie wręcz w rozdziale 10., jednak warto wiedzieć, że kiedy do sesji interaktywnej wpisuje się kolejny wiersz instrukcji kilkuwierszowej, znak zachęty może się zmienić. W prostym oknie powłoki znak zachęty zmienia się z `>>>` na `...` dla drugiego i każdego kolejnego wiersza. W interfejsie IDLE kolejne wiersze są z kolei automatycznie wcinane.

W rozdziale 10. przekonamy się, dlaczego jest to takie ważne. Na razie, kiedy w czasie wpisywania kodu napotkamy znak zachęty w postaci trzech kropek lub pustego wiersza, oznacza to najprawdopodobniej, że w jakiś sposób zmyliłyśmy Pythona, który sądzi, że wpisujemy instrukcję wielowierszową. By powrócić do normalnego znaku zachęty, należy nacisnąć `Enter` lub wybrać z klawiatury skrót `Ctrl+C`. Znaki zachęty `>>>` oraz `...` można zmienić (są one dostępne we wbudowanym module `sys`), jednak w przykładach zamieszczonych w książce zakładam, że taka zmiana nie została wykonana.

- **Instrukcje złożone w sesji interaktywnej kończy się pustym wierszem.** W sesji interaktywnej wstawienie pustego wiersza (naciśnięcie przycisku *Enter* na początku wiersza) jest niezbędne do poinformowania interaktywnego Pythona, że skończyliśmy już wpisywać instrukcję wielowierszową. Oznacza to zatem, że w celu wykonania instrukcji złożonej należy dwukrotnie nacisnąć przycisk *Enter*. W plikach puste wiersze nie są z kolei wymagane, a jeśli są obecne — zostaną zignorowane. Jeśli w sesji interaktywnej na końcu instrukcji złożonej nie naciśniemy przycisku *Enter* dwukrotnie, będzie się wydawać, jakbyśmy pozostały w stanie zawieszenia, ponieważ interpreter interaktywny nic nie zrobi — będzie czekać, aż znowu naciśniemy *Enter*!
- **Sesja interaktywna wykonuje po jednym wierszu naraz.** W sesji interaktywnej należy całkowicie zakończyć wykonywanie jednej instrukcji, by móc zacząć wpisywać kolejną. W przypadku prostych instrukcji jest to naturalne, ponieważ naciśnięcie przycisku *Enter* wykonuje wpisaną instrukcję. W przypadku instrukcji złożonych należy pamiętać o wstawieniu pustego wiersza w celu zakończenia instrukcji i wykonania jej przed rozpoczęciem wpisywania kolejnej.

Wpisywanie instrukcji wielowierszowych

Choć ryzykuję powtarzaniem się, w trakcie aktualniania tego rozdziału otrzymałem wiele e-maili od Czytelników, którzy natrafili na jakieś niespodzianki w związku z dwoma ostatnimi punktami — dlatego kwestia ta zasługuje na podkreślenie. Instrukcje wielowierszowe (inaczej: złożone) wprowadzone zostaną w kolejnym rozdziale, a ich składnią zajmiemy się bardziej formalnie w dalszej części książki. Ponieważ jednak ich zachowanie różni się nieco w plikach i w sesji interaktywnej, warto zwrócić uwagę na dwie kwestie.

Po pierwsze, należy pamiętać o kończeniu w sesji interaktywnej pustym wierszem wielowierszowych instrukcji złożonych, takich jak pętle `for` oraz testy `if`. Klawisz *Enter* należy naciągnąć dwa razy, by zakończyć całą instrukcję wielowierszową, a następnie ją wykonać. Przykładowo:

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

W pliku skryptu pusty wiersz po instrukcji złożonej nie jest jednak potrzebny — jest to wymagane *jedynie* w sesji interaktywnej. W plikach puste wiersze nie są wymagane, a jeśli są obecne — zostaną zignorowane. W sesji interaktywnej służą one do zakończenia instrukcji wielowierszowych.

Należy także pamiętać, że sesja interaktywna wykonuje *po jednej instrukcji naraz*. W celu wykonania pętli lub innej instrukcji wielowierszowej przed wpisaniem kolejnej instrukcji należy naciągnąć przycisk *Enter* dwukrotnie.

```
>>> for x in 'mielonka':  
...     print(x)           <== Dwukrotne naciśnięcie Enter przed wpisaniem nowej instrukcji  
... print('gotowe')  
File "<stdin>", line 3  
    print('gotowe')  
          ^  
SyntaxError: invalid syntax
```

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

Systemowe wiersze poleceń i pliki

Choć sesja interaktywna doskonale nadaje się do eksperymentów i testów, ma jedną ogólną wadę: programy w niej napisane znikają, kiedy tylko interpreter Pythona je wykona. Kod wpisywany interaktywnie nigdy nie jest przechowywany w pliku, dlatego nie można go uruchomić ponownie bez wpisywania go kolejny raz. Co prawda skopiowanie polecenia i ponowne wklejenie go może tu nieco pomóc, ale nie na dużą skalę, co będzie nam potrzebne, kiedy zaczniemy pisać większe programy. Żeby skopiować kod z sesji interaktywnej i wkleić go ponownie, trzeba by z niego wyciąć na przykład znaki zachęty Pythona czy dane wyjściowe programów — nie jest to do końca zgodne z metodologią nowoczesnego programowania!

By zapisać program na stałe, trzeba wpisać kod do plików, które są zazwyczaj znane jako *moduły*. Moduły to po prostu pliki tekstowe zawierające instrukcje Pythona. Po ich utworzeniu można nakazać interpreterowi Pythona wykonanie tych instrukcji w każdym pliku dowolną liczbę razy i na różne sposoby — za pomocą wiersza poleceń, kliknięcia ikony pliku czy z wykorzystaniem opcji interfejsu IDLE. Bez względu na sposób uruchomienia Python wykonuje kod z pliku modułu od góry do dołu za każdym razem, kiedy wykonyuje się plik.

Terminologia wykorzystywana w tej dziedzinie może być różna. Pliki modułów nazywa się na przykład czasami w Pythonie *programami* — program jest zatem uznawany za serię zadowanych wcześniej instrukcji przechowywanych w plikach, tak by można je było wielokrotnie wykonywać. Pliki modułów wykonywane w sposób bezpośredni są również czasami nazywane *skryptami* — co jest nieformalnym terminem oznaczającym zazwyczaj plik programu najwyższego poziomu. Niektórzy rezerwują pojęcie „moduł” dla plików importowanych z innych plików. Więcej informacji na temat znaczenia „najwyższego poziomu” i importowania za chwilę.

Bez względu na nazewnictwo na kolejnych kilku stronach omówimy sposoby wykonywania kodu wpisywanego do plików modułów. Na początek nauczymy się, jak można w najprostszym sposobie wykonywać pliki — podając ich nazwy w tym samym wierszu konsoli systemowej komputera, w którym wpisujemy słowo `python`. Choć niektórym może się to wydawać prymitywne, dla wielu programistów okno wiersza polecenia powłoki systemu w połączeniu z okiem edytora tekowego stanowi zintegrowane środowisko programistyczne w zupełności wystarczające do ich potrzeb.

Pierwszy skrypt

Zaczynamy od początku. Najpierw należy uruchomić ulubiony edytor tekstowy (na przykład *vi*, Notatnik czy IDLE) i wpisać do nowego pliku tekstowego o nazwie `script1.py` poniższe instrukcje Pythona:

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

Jest to nasz pierwszy oficjalny plik napisany w Pythonie (nie licząc dwuwierszowego skryptu z rozdziału 2.). Nie należy się szczególnie przejmować jego kodem; w skrócie plik ten:

- Importuje moduł Pythona (bibliotekę dodatkowych narzędzi) w celu pobrania nazwy platformy.
- Wykonuje trzy wywołania funkcji `print` w celu wyświetlenia wyników skryptu.
- Wykorzystuje utworzoną w momencie przypisania zmienną `x` do przechowania obiektu łańcucha znaków.
- Wykonuje różne działania na obiektach, które zaczniemy omawiać w kolejnym rozdziale.

Kod `sys.platform` jest tutaj po prostu łańcuchem identyfikującym typ komputera, na jakim pracujemy. Dane te znajdują się w standardowym module Pythona o nazwie `sys`, który należy zaimportować w celu załadowania (i znów, więcej informacji o importowaniu później).

Dla ubarwienia kodu dodałem do niego również formalne *komentarze* Pythona — tekst znajdujący się po znaku `#`. Komentarze mogą się pojawiać w osobnych wierszach lub po prawej stronie kodu w tym samym wierszu. Tekst po znaku `#` jest ignorowany i traktowany jako komentarz czytelny tylko dla ludzi; nie jest on uznawany za część składni instrukcji). Przy kopiowaniu kodu można komentarze również zignorować. W książce zazwyczaj wykorzystujemy inny styl formatowania do wizualnego wyróżnienia komentarzy, jednak w kodzie pojawiają się one jako zwykły tekst.

Ponownie należy na razie zignorować składnię kodu znajdującego się w pliku, wszystkiego dowieemy się później. Należy jednak zauważyc, że powyższy kod został wpisany do pliku, a nie w sesji interaktywnej. W międzyczasie udało nam się stworzyć w pełni funkcjonalny skrypt w języku Python.

Warto zwrócić uwagę na to, że nasz plik modułu nosi nazwę `script1.py`. Tak jak wszystkie inne pliki najwyższego poziomu, mógłby się nazywać po prostu `script1`, jednak pliki zawierające kod, który chcemy zaimportować do klienta, muszą mieć rozszerzenie `.py`. Importowanie zostanie omówione w dalszej części rozdziału. Ponieważ możemy zechcieć w przyszłości zaimportować plik, dobrym pomysłem będzie stosowanie rozszerzenia `.py` w przypadku większości tworzonych plików Pythona. Niektóre edytory tekstu wykrywają również pliki Pythona po ich charakterystycznym rozszerzeniu. Jeśli tego rozszerzenia nie ma, być może niektóre opcje — jak kolorowanie elementów składniowych czy automatyczna indentacja — nie będą dostępne.

Wykonywanie plików za pomocą wiersza poleceń

Po zapisaniu pliku tekstowego możemy zażądać od Pythona wykonania go, podając pełną nazwę pliku jako pierwszy argument polecenia `python` wpisanego w systemowej powłoce:

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

I tym razem takie polecenie powłoki systemowej wpisujemy w dowolnym miejscu służącym w naszym systemie za wiersz polecień — w oknie programu *Wiersz poleceń* systemu Windows, w oknie *xterm* czy podobnym. Należy pamiętać o zastąpieniu słowa `python` pełną ścieżką do katalogu, jeśli nie skonfigurowaliśmy ustawień zmiennej środowiskowej `PATH`.

Jeśli wszystko zadziała zgodnie z planem, powyższe polecenie powłoki sprawi, że Python wykona kod z tego pliku wiersz po wierszu, a my zobaczymy wynik trzech instrukcji `print` ze skryptu — nazwę platformy, liczbę 2 podniesioną do potęgi 100 oraz wynik tego samego

wyrażenia co wcześniej, powtarzającego łańcuch znaków (i znów, więcej na temat dwóch ostatnich opcji w rozdziale 4.).

Jeśli coś nie pójdzie zgodnie z planem, zobaczymy komunikat o błędzie. Należy się wtedy upewnić, że kod został wprowadzony do pliku dokładnie tak, jak pokazano wyżej, i spróbować ponownie. Opcje debugowania omówimy w ramce „Debugowanie kodu w Pythonie” na końcu rozdziału, natomiast na tym etapie książki najlepszym rozwiązaniem jest ślepie kopowanie.

Ponieważ ten schemat wykorzystuje do uruchamiania programów napisanych w Pythonie systemowy wiersz polecenia, zastosowanie mają wszelkie reguły składni powłoki. Można na przykład za pomocą specjalnej składni powłoki przekierować wynik wykonania skryptu Pythona do pliku, który się zapisuje, w celu sprawdzenia go w przyszłości:

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

W tym przypadku trzy pokazane wcześniej wiersze z wynikami zapisane zostaną do pliku *saveit.txt*, a nie wyświetcone w oknie wiersza polecenia. Takie działanie znane jest pod nazwą *przekierowania strumienia* (ang. *stream redirection*). Jest ono dostępne dla tekstu wejściowego i wyjściowego w systemach Windows oraz uniksowych. Nie ma to wiele wspólnego z samym Pythonem (Python po prostu obsługuje tę składnię), dlatego pominiemy tutaj szczegóły składni przekierowania.

Jeśli korzystamy z platformy Windows, poniższy przykład zadziała w taki sam sposób, jednak systemowy znak zachęty jest zazwyczaj inny:

```
C:\Python30> python script1.py
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Jak zawsze, jeśli zmienna środowiskowa PATH nie została odpowiednio ustawiona, należy podać pełną ścieżkę do Pythona lub wykonać polecenie zmieniające katalog w celu dojścia do odpowiedniej ścieżki:

```
D:\temp> C:\python30\python script1.py
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

We wszystkich aktualnych wersjach systemu Windows można również wpisać po prostu nazwę skryptu i pominąć nazwę samego Pythona. Ponieważ nowsze wersje tego systemu operacyjnego wykorzystują rejestr do odnalezienia programu, który powinien uruchomić plik, w celu wykonania pliku *.py* nie ma potrzeby w sposób jawny podawać nazwy programu („python”) w wierszu poleceń. Poprzednie polecenie można zatem w większości komputerów z systemem Windows skrócić do następującego:

```
D:\temp> script1.py
```

Wreszcie należy pamiętać, by podać pełną ścieżkę do skryptu, jeśli znajduje się on w katalogu innym od tego, w którym pracujemy. Poniższy wiersz polecenia uruchomiony w katalogu *D:\other* zakłada, że Python podany jest w zmiennej środowiskowej PATH, jednak uruchamia plik znajdujący się w innym katalogu:

```
D:\other> python c:\code\otherscript.py
```

Jeśli zmienna środowiskowa PATH nie zawiera katalogu Pythona i ani Python, ani plik skryptu nie znajdują się w katalogu, w którym pracujemy, w obu przypadkach należy użyć pełnych ścieżek:

```
D:\other> C:\Python30\python c:\code\otherscript.py
```

Wykorzystywanie wierszy poleceń i plików

Wykonywanie plików programów z systemowych wierszy poleceń jest również stosunkowo nieskomplikowaną opcją, w szczególności dla osób zaznajomionych z działaniem wierszy poleceń. Osoby poczynające powinny jednak zwrócić uwagę na kilka pułapek, których wyeliminowanie pozwoli uniknąć frustracji:

- **W systemie Windows należy uważać na automatyczne rozszerzenia.** Jeśli w systemie tym piszemy kod programów w programie Notatnik, kiedy przychodzi do zapisania naszej pracy, należy pamiętać o wybraniu opcji *Wszystkie pliki* i nadaniu plikowi odpowiedniego rozszerzenia — *.py*. W przeciwnym razie Notatnik zapisze plik z rozszerzeniem *.txt* (na przykład jako *script1.py.txt*), co sprawi, że w niektórych scenariuszach uruchamiania będzie trudniej go wykonać.

Co gorsza, system Windows domyślnie ukrywa rozszerzenia plików, więc jeśli nie zmieniliśmy opcji ich wyświetlania, możemy nawet nie zauważyc, że kod zapisany został jako plik tekstowy, a nie plik Pythona. Może nam to zdradzić ikona pliku — jeśli nie widać w niej małego węża, możemy znaleźć się w opałach. Innymi symptomami tego problemu są nie-pokolorowany kod w edytorze IDLE i pliki, które otwierają się do edycji, zamiast się wykonywać.

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

- **Rozszerzenia plików oraz ścieżki do katalogów należy dodawać w wierszu poleceń, jednak już nie dla importów.** W systemowym wierszu poleceń nie można zapomnieć o podaniu pełnej nazwy pliku, czyli należy korzystać z formy `python script1.py`, a nie `python script1`. Jednak instrukcje Pythona `import`, które omówione zostaną w dalszej części rozdziału, pomijają zarówno rozszerzenie *.py*, jak i ścieżkę do katalogu (czyli mają formę `import script1`). Może się to wydawać trywialne, ale mylenie tych dwóch sytuacji jest częstym źródłem błędów.

W wierszu poleceń jesteśmy w powłoce systemu, a nie Pythona, dlatego reguły wyszukiwania plików modułów Pythona nie mają tu zastosowania. Z tego powodu zawsze trzeba uwzględnić zarówno rozszerzenie *.py*, jak i, jeśli jest to konieczne, pełną ścieżkę do katalogu, w którym znajduje się wykonywany plik. Żeby na przykład wykonać plik znajdujący się w katalogu innym od tego, w którym obecnie pracujemy, zazwyczaj podaje się jego pełną ścieżkę (czyli `python d:\tests\script1.py`). Wewnątrz kodu Pythona wystarczy jednak napisać `import script1`, a za odnalezienie pliku odpowiedzialny będzie model wyszukiwania modułów Pythona omówiony w dalszej części książki.

- **W plikach należy korzystać z instrukcji `print`.** Tak, mówiliśmy już o tym, ale jest to tak często popełniany błąd, że warto powtórzyć tę informację jeszcze przynajmniej raz. W przeciwieństwie do kodu wpisywanego interaktywnie, by zobaczyć dane wyjściowe z plików programów, należy używać instrukcji `print`. Jeśli nie widzimy żadnych danych wyjściowych, należy sprawdzić, czy w plikach użyliśmy `print`. I znów, instrukcje `print` *nie* są wymagane w sesji interaktywnej, ponieważ Python automatycznie zwraca wyniki wyrażeń. Użycie `print` w takim kontekście nie jest błędem, jednak wymaga dodatkowego pisania.

Skrypty wykonywalne Uniksa (#!)

Osoby korzystające z Pythona w systemach Unix, Linux czy podobnych do Uniksa mogą również zamienić pliki z kodem Pythona w programy wykonywalne, podobnie jak robi się to w przypadku programów pisanych w języku powłoki, takich jak *csh* czy *ksh*. Takie pliki zazwyczaj nazywane są *skryptami wykonywalnymi* (ang. *executable scripts*). Uniksowe pliki wykonywalne to w uproszczeniu normalne pliki tekstowe zawierające instrukcje Pythona i mające dwie szczególne cechy charakterystyczne:

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

Przyjrzyjmy się teraz przykładowi przeznaczonemu dla systemu uniksowego. W edytorze tekstuowym należy utworzyć plik z kodem Pythona o nazwie *brian*:

```
#!/usr/local/bin/python
print('Zawsze patrz ' + 'na życie z humorem')      # + oznacza w łańcuchach znaków konkatenację
```

Ten specjalny pierwszy wiersz mówi systemowi operacyjnemu, gdzie znajduje się interpreter Pythona. Z technicznego punktu widzenia pierwszy wiersz jest komentarzem Pythona. Jak wspomniano wcześniej, wszystkie komentarze w programach w Pythonie rozpoczynają się od znaku `#` i rozciągają aż do końca wiersza. Są miejscem, w którym można w kodzie umieścić dodatkowe informacje przeznaczone dla innych osób. Kiedy jednak komentarz taki, jak w pierwszym wierszu powyżej, pojawi się w pliku, ma on specjalny charakter, ponieważ system operacyjny wykorzystuje go do odnalezienia interpretera, który może wykonać kod znajdujący się w dalszej części pliku.

Warto również zwrócić uwagę na to, że plik ten nazwany jest po prostu *brian*, bez rozszerzenia `.py` wykorzystywanego wcześniej w przypadku modułów. Dodanie `.py` do nazwy nie zaszkodzi (i może pomóc nam pamiętać, że ten plik jest programem Pythona), jednak ponieważ nie planujemy, by w przyszłości inne moduły importowały kod z tego pliku, jego nazwa jest bez znaczenia. Jeśli nadamy temu plikowi prawa do wykonywania za pomocą polecenia powłoki `chmod +x brian`, możemy wykonywać go z powłoki systemu operacyjnego tak samo, jakby był programem binarnym:

```
% brian
Zawsze patrz na życie z humorem
```

Informacja dla użytkowników systemu Windows — metoda opisana powyżej jest sztuczką z Uniksa i może nie działać na Waszych platformach. Nie ma się czym martwić, wystarczy skorzystać ze zwykłej techniki z wierszem poleceń opisanej wcześniej. Nazwę pliku należy w sposób jawny podać w wierszu z poleceniem `python`¹:

¹ Jak wspomnieliśmy wcześniej przy okazji omawiania wierszy poleceń, nowsze wersje systemu Windows pozwalają także na wpisanie samej nazwy pliku `.py` — wykorzystują one rejestr systemowy do ustalenia, że plik powinien zostać otwarty za pomocą Pythona (wpisanie `brian.py` jest zatem odpowiednikiem wpisania `python brian.py`). Ten tryb wiersza poleceń jest podobny do zapisu ze znakami `#!` z Uniksa, choć w Windowsie

```
C:\misc> python brian
Zawsze patrz na życie z humorem
```

W tym przypadku nie jest nam potrzebny specjalny komentarz ze znakami `#!` (choć Python po prostu go zignoruje, jeśli będzie on obecny), a plik nie musi mieć praw do wykonywania. Tak naprawdę, kiedy chcemy wykonywać te same pliki zarówno w systemie Unix, jak i Windows, uprościmy sobie życie, jeśli zawsze do uruchamiania programów będziemy korzystać z wiersza poleceń, a nie ze skryptów uniksowych.

Sztuczka z wyszukiwaniem za pomocą env w Uniksie

W systemach uniksowych możemy uniknąć kodowania ścieżki do interpretera Pythona na stałe, podając komentarz z pierwszego wiersza w następującej formie:

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

Kiedy zapiszemy ten wiersz w takiej postaci, program `env` lokalizuje interpreter Pythona zgodnie z ustawieniami wyszukiwania systemowego (w większości powłok uniksowych oznacza to przeszukanie wszystkich katalogów podanych w zmiennej środowiskowej `PATH`). Ten schemat może być bardziej przenośny, gdyż nie musimy podawać ścieżki instalacyjnej Pythona na komputerze w pierwszym wierszu wszystkich skryptów.

Zakładając, że wszędzie będziemy mieli dostęp do `env`, skrypty będą działały zawsze, bez względu na miejsce zainstalowania Pythona w naszym systemie — wystarczy tylko zmienić wartość zmiennej środowiskowej `PATH` na wszystkich platformach, a nie we wszystkich pierwszych wierszach każdego skryptu. Oczywiście powyższy zapis zakłada, że program `env` znajduje się zawsze w tym samym miejscu (w niektórych komputerach może na przykład znajdować się w `/sbin`, `/bin` lub jeszcze gdzie indziej). Jeśli tak nie jest, przenośność tego rozwiązania nie jest zbyt duża.

Kliknięcie ikony pliku

W systemie Windows otwieranie plików za pomocą klikania ich ikon jest łatwe — dzięki rejestrowi. Python automatycznie rejestruje się jako program otwierający pliki programów napisanych w tym języku po kliknięciu ich. Z tego powodu możliwe jest uruchomienie napisanych przez nas programów w Pythonie przez proste kliknięcie (lub podwójne kliknięcie) ikon ich plików za pomocą kurSORA myszy.

W systemach innych od Windows najprawdopodobniej uda się zrobić to samo, jednak ikony, eksplorator plików, schematy nawigacyjne i inne elementy mogą się nieco różnić. W niektórych systemach uniksowych być może konieczne będzie zarejestrowanie rozszerzenia `.py` za pomocą GUI eksploratora plików, dodanie skryptowi praw do wykonywania za pomocą omówionej wcześniej sztuczki ze znakami `#!` lub skojarzenie typu MIME pliku z aplikacją lub poleceniem za pomocą edycji jakichś plików, instalacji programów czy z użyciem innych narzędzi. Jeśli kliknięcie pliku nic nie da, trzeba będzie poszukać więcej informacji na ten temat w dokumentacji eksploratora plików danego systemu.

dzieje się to dla całego systemu, a nie dla pliku. Warto zauważyć, że niektóre *programy* mogą zinterpretować i wykorzystać pierwszy wiersz ze znakami `#!` w systemie Windows w podobny sposób jak w Uniksie, ale powłoka systemowa DOS całkowicie ignoruje ten wiersz.

Kliknięcie ikony w systemie Windows

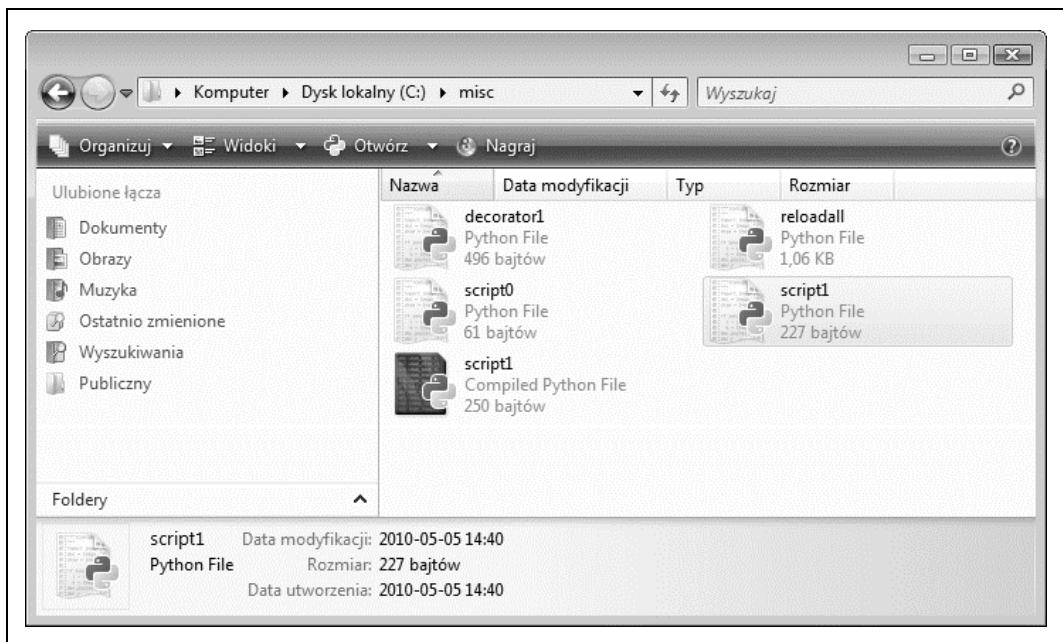
Żeby zilustrować tę kwestię, wykorzystamy utworzony wcześniej skrypt `script1.py`, powtórzony poniżej w celu uniknięcia konieczności przewracania strony:

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

Jak widzieliśmy wcześniej, plik `script1.py` można zawsze uruchomić za pomocą systemowego wiersza poleceń:

```
C:\misc> c:\python30\python script1.py
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

Kliknięcie ikony pliku pozwala nam jednak na uruchomienie go bez konieczności wpisywania czegokolwiek. Ikonę tego pliku można na przykład znaleźć po wybraniu z menu *Start* opcji *Komputer* (lub *Mój komputer* w systemie Windows XP) i przejściu do odpowiedniego katalogu na dysku; zobaczymy wtedy eksplorator plików widoczny na rysunku 3.1 (wykorzystano system Windows Vista). Pliki źródłowe Pythona mają w systemie Windows białe tło, natomiast pliki z kodem bajtowym — czarne. Żeby skorzystać z ostatnich wprowadzonych zmian, należy kliknąć (lub w inny sposób uruchomić) plik z kodem źródłowym. By uruchomić plik widoczny na rysunku, wystarczy kliknąć ikonę pliku `script1.py`.



Rysunek 3.1. W systemie Windows pliki programów Pythona pokazują się w eksploratorze plików jako ikony i mogą automatycznie zostać uruchomione po ich dwukrotnym kliknięciu przyciskiem myszy (choć w ten sposób możemy nie zobaczyć wyświetlanych danych wyjściowych czy komunikatów o błędach)

Sztuczka z funkcją input

Niestety, w systemie Windows rezultat kliknięcia ikony pliku może nie być szczególnie satysfakcyjny. Tak naprawdę powyższy przykładowy skrypt generuje po kliknięciu denerwujące „mignięcie” — nie jest to ten rodzaj informacji zwrotnych, na jakie liczą poczatkującymi programiści Pythona! Nie jest to błąd, ma jednak swoją przyczynę w sposobie, w jaki wersja Pythona przeznaczona dla systemu Windows radzi sobie z wyświetlonymi danymi wyjściowymi.

Domyślnie Python generuje wyskakujące czarne okno konsoli DOS, które służy jako wejście i wyjście dla klikniętego pliku. Jeśli skrypt po prostu wyświetla coś i kończy działanie, wtedy pokazuje się okno konsoli, w którym wyświetlany jest tekst, a następnie w momencie skończenia programu okno konsoli zamyka się. O ile nie jesteśmy bardzo szybcy albo nasz komputer nie jest bardzo wolny, nie damy rady w ogóle zobaczyć danych wyjściowych. Choć takie zachowanie jest prawidłowe, raczej nie o to nam chodziło.

Na szczęście istnieje proste obejście tego problemu. Jeśli potrzebujemy zachować dane wyjściowe pliku uruchomionego za pomocą kliknięcia ikony, wystarczy umieścić na dole skryptu wywołanie wbudowanej funkcji `input` (w wersji 2.6 była to funkcja `raw_input` — zgodnie z informacjami ze wskazówkami poniżej). Na przykład:

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

Generalnie funkcja `input` wczytuje kolejny wiersz standardowych danych wejściowych, czekając, jeśli żaden wiersz nie jest teraz dostępny. Rezultat będzie więc taki, że skrypt zostanie zatrzymany, tym samym zachowując okno z danymi wyjściowymi widoczne na rysunku 3.2. Sytuacja zmieni się dopiero po naciśnięciu przycisku *Enter*.



Rysunek 3.2. Kiedy klikamy ikonę programu w systemie Windows, będziemy mogli zobaczyć wyświetcone dane wyjściowe po dodaniu wywołania funkcji `input()` na samym końcu skryptu. Z tej sztuczki należy jednak korzystać tylko w tym kontekście!

Skoro już pokazałem tę sztuczkę, warto podkreślić, że jest ona wymagana jedynie w systemie Windows — tylko wtedy, gdy nasz skrypt wyświetla jakiś tekst i kończy działanie, oraz gdy uruchamiamy plik, klikając jego ikonę. To wywołanie powinno się dodać na dole plików najwyższego poziomu tylko i wyłącznie wtedy, gdy wszystkie trzy warunki są spełnione.

Nie ma sensu dodawać go w innych sytuacjach (chyba że ktoś lubi bez powodu naciskać przycisk *Enter*).² Może się to wydawać oczywiste, jednak to kolejny często popełniany na moich kursach błąd.

Zanim przejdziemy dalej, warto zwrócić uwagę na to, że zastosowane tutaj wywołanie `input` jest dla wejścia odpowiednikiem tego, co w przypadku wyjścia robi się za pomocą instrukcji `print`. To najprostszy sposób wczytania danych wejściowych od użytkownika i jest on o wiele bardziej uniwersalny, niż sugeruje to przykład. Przykładowo funkcja `input`:

- opcjonalnie przyjmuje łańcuch znaków, który zostanie wyświetlony jako zachęta (na przykład `input('Naciśnij przycisk Enter w celu zakończenia działania programu')`),
- zwraca do skryptu wiersz tekstu wczytany jako łańcuch znaków (na przykład `nextinput = input()`),
- obsługuje przekierowania strumienia wejścia na poziomie powłoki systemowej (na przykład `python spam.py < input.txt`) — podobnie jak instrukcja `print` robi to dla wyjścia.

Funkcję `input` będziemy wykorzystywać w nieco bardziej zaawansowany sposób w dalszej części tekstu — w rozdziale 10. będziemy z niej na przykład korzystać w interaktywnej pętli.



Uwaga na temat wersji: Osoby pracujące w Pythonie 2.6 lub wcześniejszych wersjach w powyższym kodzie powinny w miejsce funkcji `input()` użyć `raw_input()`. Starsza wersja zmieniła w Pythonie 3.0 nazwę na `input()`. W Pythonie 2.6 również istnieje funkcja `input`, jednak analizuje ona łańcuchy znaków, tak jakby były one kodem programu wpisany do skryptu, przez co nie zadziała ona w tym kontekście (pusty łańcuch znaków będzie błędem). Funkcja `input()` z Pythona 3.0 (oraz `raw_input()` z wersji 2.6) zwraca po prostu wprowadzony tekst jako łańcuch znaków, bez analizowania go. By uzyskać w Pythonie 3.0 efekt funkcji `input` z 2.6, należy użyć `eval(input())`.

Inne ograniczenia klikania ikon

Nawet z zastosowaniem sztuczki z `input` klikanie ikon plików ma swoje ograniczenia. Możemy na przykład nie zobaczyć komunikatów o błędach Pythona. Jeśli skrypt wygeneruje błąd, tekst informacji o błędzie zapisywany jest do wyskakującego okna konsoli — które natychmiast potem znika! Co więcej, dodanie wywołania funkcji `input` do pliku tym razem nie pomoże, ponieważ skrypt skończy działanie, zanim dojdzie do tego wywołania. Innymi słowy, nie będziemy w stanie ustalić, co poszło nie tak.

Ze względu na te ograniczenia najlepiej jest chyba potraktować klikanie ikon plików jako sposób uruchamiania programów po ich sprawdzeniu i usunięciu błędów lub po przekierowaniu danych wejściowych do pliku. Szczególnie na początku lepiej jest korzystać z innych technik — na przykład systemowego wiersza poleceń czy IDLE (opcji omówionej w dalszej

² Można również całkowicie zatrzymać wyskakiwanie okien konsoli DOS dla plików klikanych w systemie Windows. Pliki, których nazwy kończą się rozszerzeniem `.pyw`, wyświetlają jedynie okna skonstruowane przez skrypt, a nie domyślne okno konsoli DOS. Pliki `.pyw` to po prostu pliki źródłowe `.py`, które w systemie Windows zachowują się w taki właśnie, specjalny, sposób. Są one wykorzystywane przede wszystkim w interfejsach użytkownika pisanych w Pythonie, które tworzą własne okna, często w połączeniu z różnymi technikami zapisu wyświetlanych danych wejściowych oraz błędów do plików.

części rozdziału, zatytułowanej „Interfejs użytkownika IDLE”) — tak by nie przegapić wygenerowanych komunikatów o błędach i móc zobaczyć normalne dane wyjściowe bez uciekania się do sztuczek. Kiedy w dalszej części książki omawiać będziemy wyjątki, pokażemy, że możliwe jest przechwytywanie błędów i radzenie sobie z nimi w taki sposób, by nie powodowały zakończenia programu. W późniejszym omówieniu instrukcji `try` będzie można znaleźć alternatywny sposób powstrzymania okna konsoli przed zamknięciem się w momencie napotkania błędu.

Importowanie i przeładowywanie modułów

Dotychczas mówiliśmy o importowaniu modułów bez wyjaśnienia, co ten termin oznacza. Moduły oraz architekturę programów omówimy bardziej szczegółowo w części piątej książki, jednak ponieważ importowanie jest również jednym ze sposobów uruchamiania programów, w tym podroziale wprowadzimy podstawy modułów, które przydadzą się nam już na początku.

W uproszczeniu: każdy plik z kodem źródłowym Pythona, którego nazwa kończy się rozszerzeniem `.py`, jest modułem. Inne pliki mogą uzyskać dostęp do elementów modułu, importując ten moduł — operacje `import` ładują inny plik i przyznają dostęp do jego zawartości. Zawartość modułu jest udostępniana poprzez jego atrybuty (coś, co omówimy za moment).

Taki oparty na modułach model usług okazuje się fundamentalną ideą stojącą za architekturą programów w Pythonie. Większe programy zazwyczaj mają postać większej liczby plików modułów, które importują narzędzia z innych plików modułów. Jeden z modułów staje się plikiem głównym (lub *najwyższego poziomu*) i jest tym, który wykorzystywany jest do rozpoczęcia całego programu.

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

Jeśli na przykład zaczniemy sesję interaktywną (z systemowego wiersza poleceń, z menu *Start*, z IDLE czy w inny sposób), możemy uruchomić wcześniej utworzony plik `script1.py` za pomocą prostej instrukcji `import` (należy jednak najpierw pamiętać o usunięciu wiersza z wywołaniem funkcji `input`, dodanego w poprzednim podroziale, gdyż inaczej będziemy musieli niepotrzebnie nacisnąć przycisk `Enter`):

```
C:\misc> c:\python30\python
>>> import script1
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

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

```
>>> import script1
>>> import script1
```

Takie zachowanie jest celowe — importowanie jest zbyt kosztowną operacją, by powtarzać ją częściej niż raz na plik czy na uruchomienie programu. Jak się okaże w rozdziale 21., operacja `import` musi odnaleźć pliki, skompilować je do kodu bajtowego i dopiero wtedy wykonać.

Jeśli naprawdę chcemy zmusić Pythona do ponownego wykonania pliku w tej samej sesji (bez jej zatrzymania i ponownego uruchomienia), musimy zamiast tego wywołać wbudowaną funkcję `reload` dostępną w module biblioteki standardowej `imp` (funkcja ta dostępna jest także w Pythonie 2.6 jako zwykła funkcja wbudowana, jednak w 3.0 tak już nie jest):

```
>>> from imp import reload                                # W 3.0 trzeba załadować z modulu
>>> reload(script1)
win32
65536
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
<module 'script1' from 'script1.py'>
>>>
```

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

Pozwala to na edycję i pobranie nowego kodu w locie w czasie bieżącej sesji interaktywnej Pythona. W tej sesji na przykład druga instrukcja `print` w pliku `script1.py` została między pierwszym importem a wywołaniem funkcji `reload` zmieniona w taki sposób, by wypisać wynik działania 2 ** 16.

Funkcja `reload` oczekuje podania nazwy już załadowanego modułu, dlatego przed przeładowaniem modułu trzeba go najpierw poprawnie zaimportować. Warto zauważyc, że funkcja ta oczekuje również zastosowania nawiasów wokół nazwy obiektu modułu, natomiast `import` nie ma takich wymagań. Funkcja `reload` jest funkcją *wywoływaną*, natomiast `import` jest instrukcją.

Z tego powodu do `reload` trzeba w nawiasach przekazać nazwę modułu jako argument i dzięki temu dostajemy dodatkowy wiersz danych wyjściowych po przeładowaniu modułu. Ostatni wiersz to po prostu wyświetlona reprezentacja wartości zwracanej przez wywołanie funkcji `reload`, czyli obiekt modułu Pythona. Więcej informacji na temat wykorzystywania funkcji znajduje się w rozdziale 16.



Uwaga na temat wersji: W Pythonie 3.0 przesunięto funkcję wbudowaną `reload` do modułu biblioteki standardowej `imp`. Nadal przeładowuje ona pliki, jednak by móc z niej korzystać, trzeba ją zaimportować. W wersji 3.0 należy wykonać kod `import imp` i użyć `imp.reload(M)` lub wykonać `from imp import reload` i użyć `reload(M)`, jak w przykładzie wyżej. Instrukcje `import` oraz `from` zostaną omówione w kolejnym podrozdziale, a bardziej formalnie — w dalszej części książki.

Dla osób pracujących w Pythonie 2.6 (czy też ogólnie w wersji 2.X) `reload` dostępne jest w postaci funkcji wbudowanej, dlatego importowanie nie jest konieczne. W Pythonie 2.6 `reload` dostępne jest w *obu* formach — funkcji wbudowanej oraz funkcji modułu — w celu ułatwienia przejścia do wersji 3.0. Innymi słowy, przeładowywanie jest w wersji 3.0 nadal dostępne, jednak w celu wykonania wywołania `reload` niezbędny jest dodatkowy wiersz kodu pobierający funkcję.

Przesunięcie `reload` w Pythonie 3.0 spowodowane było po części pewnymi znymi problemami związanymi z użyciem `reload` i instrukcji `from`, z którymi spotkamy się w kolejnym podrozdziale. Mówiąc w skrócie, zmienne załadowane za pomocą instrukcji `from` nie są bezpośrednio aktualniane za pomocą funkcji `reload`, tak jak ma to miejsce w przypadku zmiennych dostępnych za pomocą instrukcji `import`. Jeśli zmienna nie zmienia się po użyciu `reload`, warto spróbować użyć instrukcji `import` i referencji do zmiennych w postaci `moduł.atrybut`.

Więcej o modułach — atrybuty

Importowanie i przeładowywianie modułów stanowi naturalną opcję uruchamiania, ponieważ operacja `import` na samym końcu wykonuje pliki. W szerszym kontekście moduły pełnią jednak rolę *bibliotek* narzędzi, co zostanie omówione w części piątej książki. Mówiąc ogólnie, moduł jest po prostu pakietem nazw zmiennych, znany jako *przestrzeń nazw* (ang. *namespace*). Nazwy w tym pakiecie nazywane są *atrybutami* — atrybut jest po prostu nazwą zmiennej, która jest przywiązana do określonego obiektu (takiego jak moduł).

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

By to zilustrować, wykorzystajmy edytor tekstu do utworzenia krótkiego pliku modułu o nazwie `myfile.py`, zawierającego następującą treść:

```
title = "Sens życia"
```

Być może jest to najprostszy moduł Pythona na świecie (zawiera tylko jedną instrukcję przypisania), jednak wystarczy nam on do zilustrowania podstaw. Kiedy zaimportujemy ten plik, jego kod jest wykonywany, by wygenerować atrybuty modułu. Instrukja przypisania tworzy atrybut modułu o nazwie `title`.

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

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

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

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

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

Jak zobaczymy w szczegółach nieco później, instrukcja `from` jest podobna do `import`, jednak dodatkowo przypisuje nazwy w importowanych komponentach. Z technicznego punktu widzenia instrukcja ta kopiuje *atrybuty* modułu w taki sposób, że stają się one *zmiennymi* w kodzie importującym. Dzięki temu do zainportowanego łańcucha znaków można się

tym razem odnosić za pomocą samego `title` (zmiennej), a nie `myfile.title` (odniesienia do atrybutu modułu).³

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

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

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

Ten plik (`threenames.py`) przypisuje trzy zmienne i tym samym generuje trzy atrybuty dla plików zewnętrznych. Trzy swoje zmienne wykorzystuje również w instrukcji `print`, co widać po wykonaniu tego pliku jako pliku najwyższego poziomu:

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

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

```
% python
>>> import threenames                         # Pobranie całego modułu.
skecz z martwą papugą
>>>
>>> threenames.b, threenames.c
('z martwą', 'papugą')
>>>
>>> from threenames import a, b, c            # Skopiowanie kilku nazw
>>> b, c
('z martwą', 'papugą')
```

Wyniki wyświetlane są w nawiasach, ponieważ tak naprawdę są *krotkami* (rodzajem obiektu omówionego w kolejnej części książki); kwestię tę możemy na razie zignorować.

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

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

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

Powyższy kod wykonałem w Pythonie 2.6 oraz 3.0. Starsze wersje Pythona mogą zwracać mniejszą liczbę zmiennych. Kiedy wywołuje się funkcję `dir` i w nawiasach przekazuje się jej nazwę importowanego modułu, zwraca ona wszystkie atrybuty znajdujące się w tym module. Niektóre ze zwracanych nazw dostajemy gratis — nazwy z początkowymi i końcowymi podwójnymi znakami `_` są wbudowane i zawsze definiowane przez Pythona; mają specjalne znaczenie dla interpretera. Zmienne zdefiniowane w naszym kodzie przez przypisanie (`a`, `b` oraz `c`) pokazują się na końcu listy będącej wynikiem wywołania funkcji `dir`.

Moduły i przestrzeń nazw

Importowanie modułów to sposób wykonywania kodu z pliku, jednak — jak pokażemy w dalszej części książki — moduły to również największe struktury w programach napisanych w Pythonie.

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

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



Instrukcja `import a from`: Powiniem wspomnieć, że instrukcja `from` w pewnym sensie niszczy cel modułów, jakim jest dzielenie przestrzeni nazw — ponieważ `from` kopiuje zmienne z jednego pliku do drugiego, może powodować nadpisywanie zmiennych o tych samych nazwach w pliku importującym (bez żadnego ostrzeżenia). Powoduje to właściwie złączenie ze sobą przestrzeni nazw, przynajmniej w zakresie skopiowanych zmiennych.

Z tego powodu niektóre osoby zalecają wykorzystywanie instrukcji `import` w miejsce `from`. Ja sam nie pójdę tak daleko — `from` nie tylko wymaga mniej pisania, ale także ten potencjalny problem w praktyce pojawi się rzadko. Poza tym jest to coś, co możemy sami kontrolować, wymieniając zmienne w instrukcji `from`. Dopóki rozumiemy, że zmiennym tym przypisane zostaną wartości, nie jest to bardziej niebezpieczne od tworzenia instrukcji przypisania — kolejnej opcji, z której pewnie będziemy chcieli skorzystać.

Uwagi na temat używania instrukcji `import` i `reload`

Z jakiegoś powodu, kiedy ludzie dowiadują się o możliwości wykonywania plików za pomocą instrukcji `import` i `reload`, wielu z nich skupia się na tej możliwości i zapomina o innych opcjach uruchamiania, które zawsze wykonują aktualną wersję kodu (czyli o klikaniu ikon, opcjach z menu IDLE, a także systemowym wierszu poleceń). Może to jednak szybko prowadzić do zamieszania — trzeba pamiętać o importowaniu, zanim będzie można przeładować moduł, o używaniu nawiasów tylko przy instrukcji `reload`, a przede wszystkim o samej konieczności

używania `reload` w celu uzyskania dostępu do aktualnej wersji kodu. Co więcej, przeładowania nie są przechodnie — przeładowanie jednego modułu powoduje jedynie przeładowanie tego modułu, a nie wszystkich pozostałych, które może on importować, dlatego czasami konieczne jest wykonanie tej operacji dla większej liczby plików.

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

Dodatkowo możemy wpaść w tarapaty, kiedy użyjemy modułów w nietypowy sposób w tym miejscu książki. Jeśli na przykład będziemy chcieli zaimportować plik modułu przechowywany w katalogu innym od bieżącego, będzie trzeba przeskoczyć do rozdziału 21., w którym omówiona jest ścieżka *wyszukiwania modułów*.

Na razie, jeśli już musimy coś importować, w celu uniknięcia komplikacji lepiej będzie zachować wszystkie pliki w katalogu, w którym pracujemy.⁴

Z drugiej strony, okazuje się jednak, że importowanie i przeładowywane modułów sprawdza się przy testowaniu. Można preferować to podejście, jednak jeśli stale będzie się napotykało te same problemy, lepiej będzie dać sobie spokój.

Wykorzystywanie `exec` do wykonywania plików modułów

Tak naprawdę istnieje więcej sposobów wykonywania kodu przechowywanego w plikach modułów, niż na razie zdradziliśmy. Przykładowo wywołanie wbudowanej funkcji `exec(open('module.py').read())` jest innym sposobem uruchamiania plików z sesji interaktywnej bez konieczności importowania i późniejszego przeładowywania ich. Każde wywołanie `exec` wykonuje aktualną wersję pliku, bez konieczności późniejszego przeładowywania go (plik `script1.py` ma taki kształt jak po przeładowaniu z poprzedniego podrozdziału):

```
C:\misc> c:\python30\python
>>> exec(open('script1.py').read())
win32
65536
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
...modyfikacja pliku script1.py w oknie edytora tekstowego...
>>> exec(open('script1.py').read())
win32
4294967296
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
```

⁴ Informacja dla osób umierających z ciekawości: Python szuka zaimportowanych modułów we wszystkich katalogach wymienionych w `sys.path` — liście łańcuchów znaków z nazwami katalogów znajdującej się w module `sys` Pythona, inicjalizowanej ze zmiennej środowiskowej `PYTHONPATH` — a także w zbiorze standardowych katalogów. Jeśli chcemy zaimportować moduł z katalogu innego od katalogu roboczego, musi on być wymieniony w ustawieniach `PYTHONPATH`. Więcej informacji na ten temat znajduje się w rozdziale 21.

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

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

```
>>> x = 999
>>> exec(open('script1.py').read())      # Kod domyślnie wykonywany jest w tej przestrzeni nazw
...te same dane wyjściowe...
>>> x                                # Jego przypisania mogą nadpisać tutejsze zmienne
'Mielonka!'
```

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



Uwaga na temat wersji: W Pythonie 2.6 poza formą `exec(open('module.py'))` dostępna jest również funkcja wbudowana `execfile('module.py')`; obie automatycznie wczytują zawartość pliku. Obydwie postacie są równoważne z formą `exec(open('module.py').read())`, która jest nieco bardziej skomplikowana, natomiast działa zarówno w wersji 2.6, jak i 3.0.

Niestety, żadna z prostszych form z Pythona 2.6 nie jest dostępna w wersji 3.0, co oznacza, że obecnie, by w pełni zrozumieć tę technikę, należy przyswoić sobie zarówno zagadnienia związane z plikami, jak i metodami ich wczytywania (wygląda to w istocie, jakby estetyka wygrała w 3.0 z praktycznością). Właściwie postać funkcji `exec` z Pythona 3.0 wymaga tak dużo pisania, że najlepszym wyjściem wydaje się unikanie jej — zazwyczaj najlepiej jest uruchamiać pliki, wpisując odpowiednie polecenia w systemowym wierszu poleceń lub korzystając z opcji menu IDLE opisanych w kolejnym podrozdziale. Więcej informacji na temat interfejsów plików wykorzystywanych przez formę funkcji `exec` z Pythona 3.0 znajduje się w rozdziale 9.

Interfejs użytkownika IDLE

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

⁵ IDLE oficjalnie jest niedokładnym rozwinięciem od IDE, ale tak naprawdę swą nazwę zawdzięcza członkowi grupy Monty Python — Ericowi Idle.

W skrócie mówiąc, IDLE to graficzny interfejs użytkownika, który pozwala na edycję, wykonywanie i przeglądanie programów w Pythonie, a także ich debugowanie. Co więcej, IDLE to program napisany w Pythonie, wykorzystujący zestaw narzędzi do GUI o nazwie *tkinter* (w wersji 2.6 znany jako Tkinter), dzięki czemu działa na większości platform, w tym Microsoft Windows, X Windows (dla platform Linux, Unix i podobnych do Uniksa), a także Mac OS (zarówno Classic, jak i OS X). Dla wielu osób IDLE jest łatwą w użyciu alternatywą dla wierszy poleceń; jest także mniej podatny na błędy od klikania ikon.

Podstawy IDLE

Przejdźmy od razu do przykładu. IDLE łatwo jest uruchomić w systemie Windows — w menu *Python* znajdującym się w menu *Start* interfejs ten ma osobny wpis (pokazany wcześniej rysunek 2.1). IDLE można również uruchomić, klikając prawym przyciskiem myszy ikonę programu w Pythonie. W niektórych systemach uniksowych skrypt najwyższego poziomu IDLE trzeba uruchamiać z wiersza poleceń lub klikając ikonę pliku *idle.pyw* lub *idle.py* znajdującego się w podkatalogu *idlelib* w katalogu *Lib* Pythona. W systemie Windows IDLE to skrypt Pythona, który obecnie można znaleźć w katalogu *C:\Python30\Lib\idlelib* (lub *C:\Python26\Lib\idlelib* w Pythonie 2.6).⁶

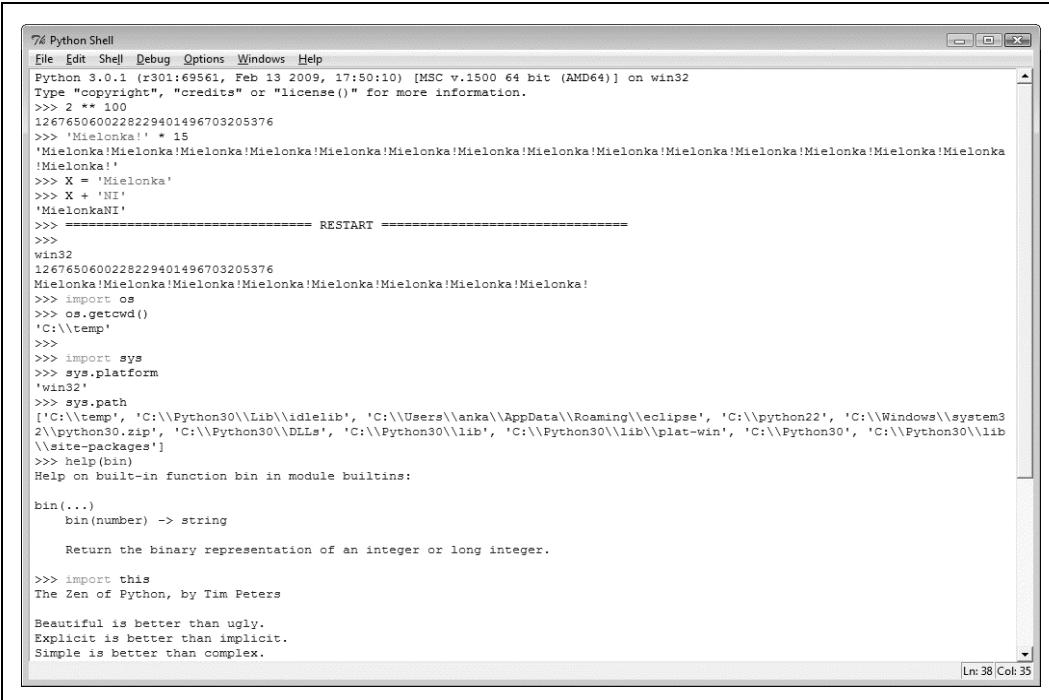
Na rysunku 3.3 widać, jak wygląda IDLE zaraz po uruchomieniu w systemie Windows. Okno powłoki Pythona, które otwiera się na początku, jest głównym oknem z sesją interaktywną (widać znaki zachęty `>>>`). Działa jak wszystkie inne sesje interaktywne — wpisywany kod jest wykonywany natychmiast — i służy jako narzędzie do testowania.

IDLE wykorzystuje znajome menu ze skrótami klawiaturowymi przypisanymi większości operacji. Do utworzenia (lub edycji) pliku z kodem źródłowym w IDLE należy otworzyć okno edytora tekstowego. W oknie głównym wybiera się menu rozwijane *File*, a następnie opcję *New Window* otwierającą okno edytora tekstowego (lub *Open...* w przypadku edycji istniejącego pliku). Pokażą się nowe okno, które będzie oknem edytora tekstowego IDLE. W nim wpisywany i wyświetlany jest kod tworzonego lub modyfikowanego pliku.

Choć w książce tego nie widać, IDLE *koloruje* kod zgodnie ze składnią — zarówno ten wpisany w oknie głównym, jak i we wszystkich oknach edycyjnych. Inny kolor mają słowa kluczowe, inny literaly i tak dalej. Pozwala to na lepszy obraz komponentów kodu (a także pozwala zauważać błędy; przykładowo rozciągające się na kilka wierszy łańcuchy znaków mają jeden kolor).

By wykonać plik edytowany w IDLE, należy otworzyć okno edytora tekstowego pliku, a następnie z menu *Run* tego okna wybrać opcję *Run Module* (lub skorzystać z odpowiedniego skrótu klawiaturowego podanego w menu). Python polecí nam najpierw zapisać plik, jeśli

⁶ IDLE to program napisany w Pythonie, w którym do stworzenia GUI tego programu wykorzystano zestaw narzędzi do budowy GUI o nazwie *tkinter* (w Pythonie 2.6 — Tkinter), znajdujący się w bibliotece standar-dowej. Dzięki temu IDLE jest przenośny. Oznacza to jednak, że, by z niego korzystać, niezbędna jest obsługa *tkinter* w Pythonie. Wersja Pythona przeznaczona dla systemu Windows domyślnie obsługuje *tkinter*, jednak niektórzy użytkownicy Linuksa oraz Uniksa mogą być zmuszeni do zainstalowania odpowiedniej obsługi *tkinter* (polecam yum *tkinter* może w niektórych dystrybucjach Linuksa być wystarczające; więcej wska-zówek na temat instalacji znajduje się w dodatku A). Również system Mac OS X może mieć wszystko zainsta-lowane; trzeba na komputerze szukać polecenia lub skryptu *idle*.



```
74 Python Shell
File Edit Shell Debug Options Windows Help
Python 3.0.1 (r301:69561, Feb 13 2009, 17:50:10) [MSC v.1500 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 ** 100
1267650600228229401496703205376
>>> 'Mielonka' * 15
'Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
!Mielonka'
>>> X = 'NI'
>>> X + 'NI'
'MielonkaNI'
>>> ===== RESTART =====
>>>
win32
1267650600228229401496703205376
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
>>> import os
>>> os.getcwd()
'C:\ttemp'
>>>
>>> import sys
>>> sys.platform
'win32'
>>> sys.path
['C:\\\\temp', 'C:\\\\Python30\\\\Lib\\\\idlelib', 'C:\\\\Users\\\\anka\\\\AppData\\\\Roaming\\\\eclipse', 'C:\\\\python22', 'C:\\\\Windows\\\\system3
2\\\\python30.zip', 'C:\\\\Python30\\\\DLLs', 'C:\\\\Python30\\\\lib', 'C:\\\\Python30\\\\lib\\\\plat-win', 'C:\\\\Python30', 'C:\\\\Python30\\\\lib
\\\\site-packages']
>>> help(bin)
Help on built-in function bin in module builtins:

bin(...)
    bin(number) -> string

    Return the binary representation of an integer or long integer.

>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
```

Rysunek 3.3. Okno główne powłoki Pythona w interfejsie programistycznym IDLE działające w systemie Windows. By utworzyć nowy plik źródłowy (New Window) lub otworzyć istniejący (Open...), należy wykorzystać odpowiednie polecenia z menu File. W menu Run edytowanego pliku można znaleźć polecenie Run Module wykonujące kod w tym oknie

zmodyfikowaliśmy go od czasu, gdy był otwierany czy ostatni raz zapisywany, a zapomnieliśmy o zapisaniu zmian — co jest często popełnianym błędem, gdy bardzo się zaangażujemy w programowanie.

Kiedy plik wykonuje się w ten sposób, dane wyjściowe oraz ewentualne komunikaty o błędach pokazują się w głównym oknie interaktywnym (oknie powłoki Pythona). Na rysunku 3.3 w trzech wierszach po wierszu ze słowem RESTART, gdzieś w połowie okna, widać na przykład wykonanie skryptu *script1.py* otwartego w osobnym oknie edycji. Komunikat RESTART informuje nas, że proces wykonywania kodu użytkownika rozpoczął się ponownie w celu wykonania edytowanego skryptu. Komunikat ten służy również do oddzielenia danych wyjściowych ze skryptu (nie pojawia się, jeśli IDLE uruchomiliśmy bez podprocesu kodu użytkownika — więcej informacji na ten temat za moment).



Wskazówka dnia: Jeśli chcemy powtórzyć poprzednie polecenia w głównym oknie interaktywnym IDLE, możemy skorzystać z kombinacji klawiszy *Alt+P* w celu przewinięcia wstecz w historii poleceń i *Alt+N* do przewinięcia do przodu (na niektórych komputerach Macintosh — *Ctrl+P* i *Ctrl+N*). Wcześniejsze polecenia zostaną wyświetlone i mogą zostać zmodyfikowane oraz ponownie wykonane. Można również ponownie wywołać polecenie poprzez umieszczenie nad nim kurSORA, skopiowanie i wklejenie, ale te czynności z reguły wymagają większego wysiłku. Poza IDLE polecenia w sesji interaktywnej w systemie Windows można wywoływać ponownie za pomocą klawiszy strzałek.

Korzystanie z IDLE

IDLE jest darmowy, łatwy w użyciu, przenośny i dostępny automatycznie na większości platform. Polecam go osobom początkującym, ponieważ ułatwia on wiele szczegółów i nie zakłada, że każdy ma wcześniejsze doświadczenie w korzystaniu z systemowego wiersza poleceń. IDLE jest jednak nieco ograniczony w porównaniu z bardziej zaawansowanymi, komercjalnymi środowiskami programistycznymi. W celu uniknięcia pewnych często spotykanych pułapek poniżej znajduje się lista najważniejszych kwestii, o których powinny pamiętać osoby zaczynające swoją przygodę z tym interfejsem.

- **Przy zapisywaniu plików trzeba w jawny sposób dodawać rozszerzenie .py.** Wspomniam o tym przy okazji omawiania plików, jednak jest to częsty błąd popełniany w IDLE, w szczególności przez użytkowników systemu Windows. IDLE nie dodaje rozszerzenia .py automatycznie przy zapisywaniu plików. Należy pamiętać o samodzielnych dopisaniu tego rozszerzenia, kiedy plik zapisuje się po raz pierwszy. Jeśli tego nie zrobimy, będziemy w stanie uruchomić plik z systemowego wiersza poleceń czy IDLE, natomiast nie będziemy mogli tego pliku zimportować interaktywnie lub do innego modułu.
- **Skrypty należy uruchamiać poprzez wybór z menu Run opcji Run Module w oknie edytora tekstu, a nie poprzez interaktywne importowanie i przeładowywanie modułów.** Wcześniej widzieliśmy, że można wykonać plik poprzez jego interaktywne zainportowanie. Takie podejście może się jednak znacznie skomplikować, ponieważ po zmianach pliki trzeba ręcznie przeładowywać. Skorzystanie z opcji *Run Module* w IDLE zawsze wykonuje najbardziej aktualną wersję pliku, podobnie jak w przypadku użycia systemowego wiersza poleceń. Zachęca również użytkownika do uprzedniego zapisania pliku, o ile jest to potrzebne (kolejny błąd często popełniany poza IDLE).
- **Przeładowywać trzeba jedynie moduły testowane w sposób interaktywny.** Tak jak w przypadku wiersza poleceń powłoki, opcja *Run Module* zawsze wykonuje najbardziej aktualną wersję zarówno pliku najwyższego poziomu, jak i wszelkich importowanych przez niego modułów. Z tego powodu stosowanie *Run Module* eliminuje wiele nieporozumień związanych z importowaniem. Niezbędne jest jedynie przeładowywanie modułów, które importujemy i testujemy w sposób interaktywny. Jeśli zamiast tej opcji postanowimy skorzystać z importowania i przeładowywania, warto pamiętać o skrótach klawiaturowych *Alt+P* i *Alt+N*, które służą do ponownego użycia wcześniejszych poleceń.
- **IDLE można dostosować do własnych potrzeb.** By zmienić czcionki tekstu i kolory, w dowolnym oknie IDLE należy z menu *Options* wybrać opcję *Configure*. Można również zmienić działania przypisane do kombinacji klawiszy czy ustawienia dotyczące wcinania tekstu. Więcej wskazówek można znaleźć w menu rozwijanym *Help*.
- **W IDLE nie ma na razie opcji czyszczącej ekran.** Prośba o dodanie tej opcji często się powtarza (być może dlatego, że podobne IDE już ją udostępniają), więc może zostanie ona w końcu dodana. Na razie jednak nie da się wyczyścić tekstu z okna sesji interaktywnej. Jeśli chcemy usunąć ten tekst z widoku, możemy albo przytrzymać klawisz *Enter*, albo wpisać pętlę Pythona wyświetlającą serię pustych wierszy (nikt oczywiście nie korzysta z tej ostatniej opcji, jednak brzmi ona jak bardziej zaawansowana technika programistyczna od naciśnięcia klawisza *Enter*!).
- **GUI tkinter oraz programy z wątkami mogą nie działać zbyt dobrze w IDLE.** Ponieważ IDLE jest programem napisanym w Pythonie z użyciem interfejsu tkinter, może się zawieść, kiedy wykonuje się pewien typ zaawansowanych programów napisanych z użyciem

tych technologii. W najnowszych wersjach IDLE, które wykonują kod użytkownika w jednym procesie, a kod samego graficznego interfejsu użytkownika IDLE w innym, nie jest to aż tak dokuczliwe, jednak w niektórych przypadkach (zwłaszcza w programach wykorzystujących wielowątkowość) może się zdarzyć. Nasz kod może nie sprawiać takich problemów, jednak generalna zasada brzmi, że bezpieczniej jest wykorzystywać IDLE do edycji programów tego typu, natomiast do uruchamiania ich lepiej jest stosować inne opcje, takie jak klikanie ikon czy używanie systemowych wierszy poleceń. Kiedy mamy wątpliwości, czy nasz kod zawodzi tylko w IDLE, należy go przetestować poza tym środowiskiem.

- **Kiedy wystąpią problemy z połączeniem, warto spróbować uruchomić IDLE w trybie z jednym procesem.** Ponieważ IDLE wymaga komunikacji pomiędzy oddzielnymi procesami użytkownika i GUI, czasami ma problem z wystartowaniem na niektórych platformach (najbardziej zauważalne jest to w przypadku komputerów z systemem Windows z uwagi na programy typu firewall blokujące połączenia). Jeśli napotkamy ten typ błędów, zawsze można uruchomić IDLE z wiersza poleceń w sposób zmuszający go do działania w trybie z jednym procesem, bez podprocesu kodu użytkownika, co pozwala uniknąć problemów z komunikacją. Tryb ten wymuszany jest przez użycie opcji wiersza poleceń `-n`. W systemie Windows należy na przykład uruchomić okno aplikacji Wiersz polecenia i wpisać polecenie `idle.py -n` w katalogu `C:\Python30\Lib\idlelib` (żeby przejść do tego katalogu, czasami trzeba najpierw skorzystać z polecenia `cd`).
- **Należy pamiętać o pewnych ułatwieniach dostępnych tylko w IDLE.** IDLE stara się ułatwiać życie osobom początkującym, jednak niektóre z rozwiązań nie będą miały zastosowania poza tym programem. IDLE wykonuje na przykład nasz skrypt w swojej własnej interaktywnej przestrzeni nazw, dlatego zmienne z tego kodu automatycznie pojawiają się w sesji interaktywnej IDLE — nie zawsze niezbędne jest wykonywanie polecenia `import` w przypadku dostępu do nazw najwyższego poziomu z plików, które już były wykonywane. Może to być przydatne, ale także mylące, ponieważ poza środowiskiem IDLE przed użyciem nazw z innych plików najpierw trzeba je zaimportować.

IDLE automatycznie nie tylko przechodzi do katalogu wykonanego właśnie pliku, ale także dodaje katalog tego pliku do ścieżki wyszukiwania importowanych modułów. Jest to przydatna opcja pozwalająca na importowanie plików bez ustawiania ścieżki wyszukiwania, jednak nie będzie to działało w ten sam sposób, kiedy pliki będziemy wykonywać poza IDLE. Korzystanie z takich opcji nie jest niczym niewłaściwym, jednak należy pamiętać, że są to właściwości IDLE, a nie samego Pythona.

Zaawansowane opcje IDLE

Poza podstawowymi opcjami, takimi jak edycja i uruchamianie plików, IDLE udostępnia również bardziej zaawansowane możliwości, w tym debugger oraz przeglądarkę obiektów. Dostęp do debugera odbywa się w IDLE za pomocą menu *Debug*, natomiast przeglądarka obiektów znajduje się w menu *File*. Przeglądarka pozwala na przechodzenie do plików oraz obiektów w nich się znajdujących po ścieżce wyszukiwania modułów. Kliknięcie pliku lub obiektu otwiera odpowiednie źródło w oknie edytora tekstowego.

Debugowanie w IDLE odbywa się po wybraniu opcji *Debugger* z menu *Debug* w oknie głównym programu, a następnie uruchomieniu skryptu za pomocą opcji *Run Module* z menu *Run*

w oknie edytora tekstowego. Po włączeniu debugera w kodzie można ustawić punkty kontrolne, klikając prawym przyciskiem myszy odpowiednie wiersze w edytorze tekstowym; służą one do zatrzymywania wykonywania programu. Można również na przykład wyświetlić zmienne, a także obserwować wykonywanie kodu w trakcie debugowania — aktualny wiersz kodu jest wyróżniany.

W przypadku mniej skomplikowanych operacji debugowania można również kliknąć prawym przyciskiem myszy tekst komunikatu o błędzie w celu szybkiego przejścia do wiersza kodu, w którym wystąpił błąd — dzięki takiej możliwości naprawa i ponowne wykonanie kodu odbywają się bardzo szybko. Dodatkowo edytor tekstowy IDLE zawiera wiele przydatnych programistom narzędzi, w tym automatyczną indentację (wcinanie kodu) czy zaawansowane opcje wyszukiwania tekstu i plików. Ponieważ graficzny interfejs użytkownika IDLE jest bardzo interaktywny, warto samemu poeksperymentować z tym programem, by poznać dalsze jego narzędzia.

Inne IDE

Ponieważ IDLE jest darmowy, przenośny i jest standardową częścią Pythona, stanowi dobre pierwsze środowisko programistyczne, z którym warto się zapoznać, jeśli planuje się wykorzystywać jakikolwiek system typu IDE. Osobom początkującym polecam wykonywanie ćwiczeń z książki właśnie w IDLE, o ile nie jest się jeszcze zaznajomionym z trybem wykorzystywania wiersza poleceń i preferuje się takie rozwiązanie. Istnieje jednak wiele alternatywnych środowisk programistycznych; niektóre z nich są znacznie bardziej rozbudowane od IDLE i mają więcej możliwości. Poniżej znajduje się lista kilku najpopularniejszych.

Eclipse i PyDev

Eclipse to zaawansowane środowisko programistyczne na licencji open source. Powstało jako IDE dla języka Java, jednak po zainstalowaniu dodatku PyDev (lub innego dodatku o podobnych funkcjach) obsługuje również programowanie w Pythonie. Eclipse jest popularnym wyborem dla programowania w Pythonie, a możliwości tego programu znacznie wykraczają poza opcje dostępne w IDLE. Obejmują one między innymi uzupełnianie kodu, kolorowanie składni, analizę składni, refaktoryzację kodu oraz debugowanie. Wadą tego systemu wydaje się to, że jest on spory, a pewne możliwości mogą wymagać instalowania rozszerzeń na licencji *shareware* (z czasem może się to zmienić). Mimo tych ograniczeń osoby czujące, że czas przejść z IDLE na coś bardziej zaawansowanego, powinny zainteresować się połączeniem Eclipse i PyDev. Eclipse można pobrać ze strony <http://www.eclipse.org/>, natomiast PyDev — <http://pydev.org/>.

Komodo

Komodo jest zaawansowanym środowiskiem programistycznym przeznaczonym dla Pythona i innych języków programowania. Zawiera opcje kolorowania składni, edycji tekstu, debugowania, a także wiele innych. Komodo oferuje również wiele możliwości, które nie są dostępne w IDLE, w tym pliki projektów, integrację z systemem kontroli źródła, debugowanie wyrażeń regularnych, a także program oparty na przeciąganiu i upuszczaniu, generujący kod w Pythonie, i tkinter, który służy do interaktywnego implementowania zaprojektowanych GUI. W tej chwili Komodo nie jest darmowy; program ten można pobrać ze strony <http://www.activestate.com>.

NetBeans IDE dla Pythona

NetBeans to graficzne środowisko programistyczne na licencji open source udostępniające wiele zaawansowanych opcji przydatnych programistom Pythona — między innymi uzupełnianie kodu, automatyczną indentację i kolorowanie kodu, wskazówki edytora, składanie kodu, refaktoryzację, debugowanie, testy pokrycia oraz projekty. Za jego pomocą można programować zarówno w CPythonie, jak i w Jythonie. Tak jak Eclipse, NetBeans wymaga dodatkowych kroków instalacyjnych wykraczających poza potrzebne w IDLE, jednak wiele osób uważa, że jego możliwości są tego warte. Najświeższe informacje na temat tego środowiska oraz odnośniki można znaleźć za pomocą wyszukiwarki.

PythonWin

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

Inne

Wiem o istnieniu około dziesięciu innych szeroko wykorzystywanych środowisk programistycznych (na przykład komercyjne *Wing IDE* czy *PythonCard*), jednak brak mi tutaj miejsca na rzetelne ich opisanie. Z pewnością z czasem pojawi się jeszcze większa ich liczba. Prawie każdy edytor tekstowy dla programistów obsługuje obecnie w jakimś stopniu programowanie w Pythonie; czasem opcje te są od razu wbudowane, a innym razem trzeba je zainstalować osobno. Edytory Emacs i Vim mają dość niezłą obsługę Pythona.

Zamiast próbować opisać tutaj wszystkie dostępne opcje, lepiej będzie skierować osoby zainteresowane na stronę <http://www.python.org> lub zachęcić każdego do wyszukania edytorów Pythona za pomocą wyszukiwarki — zapytanie „Python editors” powinno zwrócić stronę Wiki, na której znajdują się informacje o wielu środowiskach programistycznych oraz opcjach edytorów tekstowych.

Inne opcje wykonywania kodu

Omówiliśmy już, w jaki sposób kod wykonuje się interaktywnie, oraz jak na różne sposoby uruchamia się kod zapisany w plikach — na przykład za pomocą systemowych wierszy poleceń, importowania i exec czy graficznych środowisk programistycznych. W większości przypadków przedstawionych w książce to wystarczy. Istnieją jednak inne sposoby wykonywania kodu napisanego w Pythonie, z których większość ma specjalne lub zawężone przeznaczenie. Poniżej przyjrzymy się kilku z nich.

Osadzanie wywołań

W pewnych specyficznych zastosowaniach kod Pythona może również być automatycznie wykonywany przez system go zawierający. W takich przypadkach mówimy, że programy w Pythonie są *osadzone* w innym programie, to znaczy przez niego wykonywane. Sam kod

w Pythonie może na przykład zastać umieszczony w pliku tekstowym, przechowywany w bazie danych, pobrany ze strony HTML czy odczytany z dokumentu XML. Z punktu widzenia wykonywania to inny system (a nie my sami) może nakazać Pythonowi wykonanie kodu, który utworzyliśmy.

Taki osadzony tryb wykonywania jest często wykorzystywany w dostosowywaniu programów do potrzeb użytkowników końcowych. Program z grą może na przykład zezwalać na dokonywanie modyfikacji za pomocą dostępnego dla użytkownika osadzonego kodu napisanego w Pythonie, który znajduje się w strategicznych punktach aplikacji. Użytkownicy mogą modyfikować ten typ systemu, udostępniając lub zmieniając kod w Pythonie. Ponieważ kod w tym języku jest interpretowany, nie istnieje konieczność ponownej komplikacji całego systemu w celu uwzględnienia zmian (więcej informacji na temat wykonywania kodu w Pythonie znajduje się w rozdziale 2.).

W tym trybie system zawierający wykonywany kod może być napisany w językach C, C++ czy nawet Java, jeśli korzystamy z Jythona. Można na przykład tworzyć i uruchamiać ciągi kodu w Pythonie wewnętrz programu w języku C, wywołując funkcje w API wykonawczym Pythona (zbiorze usług eksportowanych przez biblioteki tworzone, kiedy Python kompilowany jest na danym komputerze):

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

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

Niniejsza książka nie jest poświęcona integracji kodu napisanego w C i Pythonie, jednak warto pamiętać, że w zależności od tego, w jaki sposób nasza organizacja planuje wykorzystywać Pythona, nie my możemy być osobami uruchamiającymi stworzone w Pythonie programy. Bez względu na okoliczności nadal będziemy mogli wykorzystać omówione techniki uruchamiania interaktywnego lub za pomocą plików w celu przetestowania kodu oddzielnie od systemu, w którym się go osadzi.⁷

Zamrożone binarne pliki wykonywalne

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

⁷ Więcej informacji na temat osadzania Pythona w kodzie napisanym w językach C i C++ można znaleźć w książce *Programming Python* (O'Reilly). API osadzające może bezpośrednio wywoływać funkcje Pythona czy ładować moduły. Warto również zauważyć, że system Jython pozwala na wywoływanie kodu w Pythonie przez programy w Javie za pomocą API opartego na Javie (klasy interpretera Pythona).

Uruchamianie kodu w edytorze tekstowym

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

Jeszcze inne możliwości uruchamiania

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

Przyszłe możliwości

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

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

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

Jaką opcję wybrać?

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

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

Debugowanie kodu w Pythonie

Oczywiście żadnemu z moich czytelników ani studentów nie zdarzą się błędy w kodzie (*tu wstawić odpowiedniego emotikona*), jednak na potrzeby naszych przyjaciół, którzy mogą mieć tego rodzaju problemy, przyjrzyjmy się strategiom stosowanym przez prawdziwych programistów Pythona, którzy chcą debugować swój kod.

- **Nie robimy nic.** Pisząc to, nie mam na myśli, że programiści Pythona nie debugują swojego kodu. Kiedy jednak w programie napisanym w tym języku popełnimy błąd, otrzymujemy od razu bardzo przydatny i czytelny komunikat o błędzie (którego jeszcze nie widział, z pewnością niedługo zobaczy). Osobom znającym Pythona, zwłaszcza na potrzeby własnego kodu, często to wystarcza — dość przeczytać komunikat o błędzie i naprawić podany wiersz w podanym pliku. Dla wielu osób to właśnie oznacza „debugowanie w Pythonie”. Może to jednak nie być idealne rozwiązanie w przypadku dużych systemów, których nie napisaliśmy samodzielnie.
- **Wstawiamy instrukcję `print`.** Chyba najczęściej wykorzystywany przez programistów Pythona sposobem debugowania kodu (za pomocą tej metody debuguję kod i ja) jest wstawianie instrukcji `print` i ponowne wykonywanie kodu. Ponieważ kod w Pythonie wykonywany jest natychmiast po wprowadzeniu zmian, zazwyczaj jest to najszybszy sposób, aby uzyskać więcej informacji niż za pomocą samych komunikatów o błędach. Instrukcje `print` nie muszą być szczególnie wyszukane — proste „Jestem tutaj” albo wyświetlenie wartości zmiennych często da nam wystarczający kontekst. Należy jedynie pamiętać o usunięciu lub przeniesieniu do komentarza (czyli dodaniu z przodu znaku #) wykorzystanych instrukcji `print` przed opublikowaniem kodu!
- **Używamy debugera z graficznego interfejsu użytkownika IDE.** W przypadku większych systemów, których nie jesteśmy autorami, a także na potrzeby osób początkujących, które chcą bardziej szczegółowo śledzić kod, większość środowisk programistycznych dla Pythona zawiera jakiś rodzaj obsługi debugowania dostępnego za pomocą kliknięcia myszą. Także IDLE zawiera debuger, jednak w praktyce nie wydaje się on używany zbyt często — być może dlatego, że nie ma wiersza poleceń, a może przez to, że wstawienie do kodu instrukcji `print` jest szybsze od konfigurowania sesji debugowania w GUI. By dowiedzieć się więcej na ten temat, warto zajrzeć do pomocy programu IDLE albo po prostu spróbować samemu. Podstawowy interfejs debugera opisany jest w podrozdziale „Zaawansowane opcje IDLE” w niniejszym rozdziale. Pozostałe środowiska programistyczne, takie jak Eclipse, NetBeans, Komodo czy Wing IDE, także oferują zaawansowane debugery obsługiwane za pomocą kliknięcia myszą. Więcej informacji na ich temat można znaleźć w dokumentacji.

- **Używamy debugera wiersza poleceń pdb.** W celu uzyskania najwyższego stopnia kontroli nad kodem Python udostępnia debugger kodu źródłowego o nazwie `pdb`, będący modelem biblioteki standardowej tego języka. W `pdb` polecenia wpisuje się wiersz po wierszu, wyświetla zmienne, ustawia i kasuje punkty kontrolne czy pozwala kontynuować kod do punktu wstrzymania lub błędu. Debugger `pdb` można uruchomić interaktywnie za pomocą importowania, a także jako skrypt najwyższego poziomu. Bez względu na sposób uruchomienia, z uwagi na to, że polecenia wpisuje się w celu kontrolowania sesji, jest to narzędzie o dużych możliwościach. Zawiera ono także funkcję postmortem, którą można wykonać po wystąpieniu wyjątku w celu otrzymania informacji z momentu wystąpienia błędu. Więcej informacji na temat `pdb` znajduje się w dokumentacji biblioteki Pythona oraz w rozdziale 35.
- **Inne opcje.** W przypadku specyficznych wymagań w zakresie debugowania kodu dodatkowe narzędzia można znaleźć wśród programów na licencji open source — w tym obsługę programów wielowątkowych, osadzonego kodu czy dołączania procesów. Przykładowo system `Winpdb` to samodzielny debugger z zaawansowaną obsługą debugowania i działającymi na różnych platformach interfejsami GUI oraz konsolą.
Powyższe opcje zyskają na znaczeniu, kiedy zaczniemy pisać większe skrypty. Chyba najlepszą wiadomością dotyczącą debugowania jest jednak to, że błędy są w Pythonie wykrywane i zgłasiane, a nie po cichu ignorowane czy powodujące zakończenie działania całego systemu. Tak naprawdę same błędy są dobrze zdefiniowanym mechanizmem znanym jako *wyjątki*, które można przechwytywać i przetwarzanie (więcej informacji na ten temat znajduje się w siódmej części książki). Popełnianie błędów nigdy nie jest oczywiście przyjemnością, jednak mówiąc z pozycji osoby, która pamięta czasy, gdy debugowanie oznaczało wyciągnięcie kalkulatora szesnastkowego i przechodzenie w pocie czoła przez stosy wydruków zrzutów z pamięci, obsługę debugowania w Pythonie uznaję za element sprawiający, że błędy stały się o wiele mniej dotkliwe, niż mogłyby być.

Podsumowanie rozdziału

W niniejszym rozdziale przyjrzaliśmy się popularnym sposobom uruchamiania programów napisanych w Pythonie — wykonywaniu kodu wpisanego w sposób interaktywny, wykonywaniu kodu przechowywanego w plikach za pomocą systemowego wiersza poleceń, klikaniu ikon plików, importowaniu modułów, wywołań `exec`, a także środowiskom programistycznym z graficznym interfejsem użytkownika, takim jak IDLE. Celem tego rozdziału było dostarczenie użytkownikowi odpowiedniej ilości informacji, która wystarczy mu do zaczęcia pisania kodu, co zrobimy już w kolejnej części książki. W części tej zaczniemy omawiać sam język, rozpoczynając od jego typów podstawowych.

Najpierw jednak pora na tradycyjny quiz końcowy, który sprawdzi wiadomości omówione w niniejszym rozdziale. Ponieważ jest to również ostatni rozdział tej części książki, po quizie można znaleźć zbiór bardziej rozbudowanych ćwiczeń, które sprawdzą naszą znajomość zagadnień poruszonych w całej części. Pomoc przy rozwiązywaniu ćwiczeń można znaleźć w dodatku B; dodatek ten można również wykorzystać do odświeżenia wiedzy przedstawionej w książce. Polecam zajrzeć do niego po skończeniu wykonywania ćwiczeń.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób można rozpoczęć interaktywną sesję interpretera?
2. Gdzie wpisuje się wiersz z poleceniem pozwalającym na uruchomienie pliku skryptu?
3. Należy podać cztery lub większą liczbę sposobów wykonania kodu zapisanego w pliku skryptu.
4. Należy podać dwie pułapki związane z klikaniem ikon plików w systemie Windows.
5. Dlaczego konieczne może być przeładowanie modułu?
6. W jaki sposób można w IDLE wykonać skrypt?
7. Należy podać dwie pułapki związane z wykorzystywaniem IDLE.
8. Co to jest przestrzeń nazw i jaki ma ona związek z plikami modułów?

Sprawdź swoją wiedzę — odpowiedzi

1. W systemie Windows sesję interaktywną rozpoczyna się, klikając przycisk *Start*, wybierając opcję *Wszystkie programy*, klikając wpis z Pythonem i wybierając opcję *Python (command line)*. Ten sam efekt można uzyskać w systemie Windows i na innych platformach, wpisując `python` do wiersza poleceń w systemowym oknie konsoli (w systemie Windows będzie to program o nazwie *Wiersz polecenia*). Alternatywą jest uruchomienie IDLE, ponieważ główne okno powłoki Pythona w tym programie jest sesją interaktywną. Jeśli nie ustaliliśmy zmiennej systemowej `PATH` w taki sposób, by była w stanie odnaleźć Pythona, być może najpierw trzeba będzie za pomocą polecenia `cd` przejść do miejsca, w którym zainstalowany jest Python, lub zamiast wpisywać `python`, podać pełną ścieżkę (na przykład `C:\Python30\python` w systemie Windows).
2. Wiersze poleceń można wpisywać w dowolnej konsoli systemowej naszej platformy — Wierszu polecenia w systemie Windows, xterm czy oknie terminala w systemach Unix, Linux oraz Mac OS X.
3. Kod z pliku skryptu (a tak naprawdę modułu) można wykonać za pomocą systemowego wiersza poleceń, kliknięcia ikony pliku, importowania i przeładowywania, wbudowanej funkcji `exec`, a także wybrania pozycji z menu graficznego interfejsu użytkownika IDE, na przykład opcji *Run Module* z menu *Run* w IDLE. W systemie Unix kod można także wykonywać jako plik wykonywalny za pomocą sztuczki z `#!/`, a niektóre platformy obsługują bardziej wyspecjalizowane techniki uruchamiania, takie jak przeciąganie i upuszczanie. Dodatkowo niektóre edytory tekstu oferują unikalne sposoby wykonywania kodu napisanego w Pythonie, a niektóre programy w Pythonie udostępniane są jako samodzielne, wykonywalne „zamrożone pliki binarne”. Niektóre systemy wykorzystują kod w Pythonie w trybie osadzenia, gdzie jest on wykonywany automatycznie przez program zawierający napisany w języku takim, jak C, C++ czy Java. Ta ostatnia technika służy zazwyczaj do udostępnienia użytkownikowi warstwy pozwalającej na dostosowanie programu do własnych potrzeb.
4. Skrypty wyświetlające coś i kończące swoje działanie sprawiają, że plik wyjściowy natychmiast znika, zanim jeszcze zobaczymy dane wyjściowe (dlatego czasem przydaje

się sztuczka z `input`). Komunikaty o błędach wygenerowane przez skrypt również pojawiają się w oknie, które zamyka się, zanim zdążyliśmy przejrzeć jego zawartość (między innymi dlatego lepszym rozwiązaniem najczęściej okazuje się korzystanie z systemowego wiersza poleceń lub środowiska programistycznego, takiego jak IDLE).

5. Python domyślnie importuje (ładuje) moduł tylko raz na dany proces, dlatego jeśli zmienimy jego kod źródłowy i chcemy wykonać nową wersję bez zatrzymywania i ponownego uruchamiania Pythona, trzeba go będzie przeładować. Przed przeładowaniem modułu trzeba go przynajmniej raz zainportować. Wykonywanie plików z kodem z systemowego wiersza poleceń powłoki, za pomocą kliknięcia ikony lub poprzez środowisko programistyczne, takie jak IDLE, sprawia, że problem ten znika, ponieważ te sposoby za każdym razem wykonują bieżącą wersję kodu źródłowego pliku.
6. Wewnątrz okna edytora tekstowego pliku, który chcemy wykonać, należy wybrać z menu *Run* opcję *Run Module*. Wykonuje ona kod źródłowy z tego okna jako plik skryptu najwyższego poziomu i wyświetla jego dane wyjściowe w interaktywnym oknie powłoki Pythona.
7. Pewne typy programów nadal mogą zawiesić IDLE — w szczególności wielowątkowe programy GUI (zagadnienie wielowątkowości pozostaje poza zakresem niniejszej książki). IDLE ma również pewne cechy, które mogą powodować problemy poza tym środowiskiem, na przykład to, że zmienne skryptu są automatycznie importowane do zakresu interaktywnego, co stosowane jest tylko w IDLE, a nie w samym Pythonie.
8. Przestrzeń nazw to pakiet zmiennych (czyli nazw). Przybiera ona w Pythonie formę obiektu wraz z atrybutami. Każdy plik modułu automatycznie staje się przestrzenią nazw, czyli pakietem zmiennych odzwierciedlającym przypisania wykonane na najwyższym poziomie pliku. Przestrzenie nazw pomagają zapobiec konfliktom nazw w programach napisanych w Pythonie — ponieważ każdy plik modułu jest niezależną przestrzenią nazw, by korzystać z nazw z innego pliku, konieczne jest uprzednio jego zainportowanie.

Sprawdź swoją wiedzę — ćwiczenia do części pierwszej

Czas zabrać się za samodzielne tworzenie kodu. Pierwsza sesja z ćwiczeniami jest dość prosta, choć kilka z pytań zawiera odniesienia do tematów, które pojawią się w kolejnych rozdziałach. Odpowiedzi na poniższe pytania znajdują się w podrozdziale „Część I Wprowadzenie” dodatku B. Ćwiczenia oraz ich rozwiązania czasami zawierają informacje dodatkowe, które nie są omówione w tekście głównym książki, dlatego warto zajrzeć do odpowiedzi nawet wtedy, gdy na wszystkie pytania odpowiedzieliśmy samodzielnie.

1. *Interakcja.* Wykorzystując systemowy wiersz poleceń, IDLE lub inną metodę, należy uruchomić interaktywny wiersz poleceń Pythona (z zachętą `>>>`) i wpisać wyrażenie "Witaj, sir Robinie!" wraz z cudzysłowem. Ten łańcuch znaków powinien zostać nam zwrotny. Celem tego ćwiczenia jest skonfigurowanie naszego środowiska w taki sposób, by było ono w stanie korzystać z Pythona. W pewnych okolicznościach być może najpierw trzeba będzie wykonać polecenie powłoki `cd`, wpisać pełną ścieżkę do pliku wykonywalnego Pythona czy dodać ścieżkę do Pythona do zmiennej środowiskowej `PATH`. W razie potrzeby zmienną `PATH` można ustawić w pliku `.chrc` lub `.kshrc`, tak by Python był zawsze

dostępny w systemach opartych na Uniksie. W systemie Windows można skorzystać z *setup.bat*, *autoexec.bat* lub GUI zmiennych środowiskowych. Pomoc w zakresie ustawiania zmiennych środowiskowych można znaleźć w dodatku A.

2. *Programy*. W wybranym edytorze tekstowym należy utworzyć prosty plik modułu zawierający pojedynczą instrukcję `print('Witaj, module!')` i zapisać go jako *module1.py*. Teraz należy wykonać ten plik z wykorzystaniem dowolnej opcji — IDLE, kliknięcia ikony, przekazania do interpretera Pythona w systemowym wierszu poleceń powłoki (na przykład `python module1.py`), wywołania wbudowanej funkcji `exec` czy importowania i przeładowywania. Najlepiej będzie poeksperymentować z wykonywaniem pliku za pomocą maksymalnie dużej liczby technik zaprezentowanych w niniejszym rozdziale. Które techniki wydają się najłatwiejsze? (Na to pytanie nie ma oczywistej jednoznacznej odpowiedzi).
3. *Moduły*. Teraz należy uruchomić sesję interaktywną Pythona (z zachetą `>>>`) i zaimportować moduł napisany w ćwiczeniu 2. Warto spróbować przenieść plik do innego katalogu i raz jeszcze zaimportować go z poprzedniego (czyli uruchomić Pythona w oryginalnym katalogu, z którego importowaliśmy moduł). Co się stanie? (Wskazówka: czy plik z kodem bajtowym *module1.pyc* nadal znajduje się w oryginalnym katalogu?).
4. *Skrypty*. Jeśli nasza platforma obsługuje tę opcję, należy dodać na górze pliku modułu *module1.py* wiersz ze znakami `#!`, nadać plikowi prawa do wykonywania i uruchomić go bezpośrednio jako plik wykonywalny. Co musi zawierać pierwszy wiersz? Znaki `#!` zazwyczaj mają znaczenie tylko na platformach Unix, Linux i podobnych do Uniksa, takich jak Mac OS X. Osoby pracujące w systemie Windows mogą zamiast tego spróbować uruchomić plik, podając samą jego nazwę w oknie konsoli DOS bez poprzedzającego ją słowa `python` (taka opcja zadziała w nowszych wersjach tego systemu) lub za pomocą okna dialogowego *Uruchom...* z menu *Start*.
5. *Błędy i debugowanie*. Należy poeksperymentować z wpisywaniem wyrażeń matematycznych i instrukcji przypisania w interaktywnym wierszu poleceń Pythona. Przy okazji należy wpisać wyrażenia `2 ** 500` oraz `1 / 0`, a także odwołać się do niezdefiniowanej nazwy zmiennej, tak jak zrobiliśmy to w tekście rozdziału. Co się stanie?

Nie każdy wie, że kiedy robimy błąd, przetwarzamy wyjątki (zagadnienie to zostanie omówione szerzej w części siódmej). Jak się okazuje, wywołujemy właśnie coś, co znane jest pod nazwą *domyślnego programu obsługi wyjątków* (ang. *default exception handler*) — logiki wyświetlającej standardowe komunikaty o błędach. Jeśli błąd nie zostanie przechwycony przez nas, przechwyci go program obsługi wyjątków, który w odpowiedzi wyświetli standardowy komunikat.

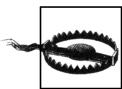
Wyjątki są w Pythonie powiązane z pojęciem *debugowania*. Jeśli dopiero zaczynamy programowanie w tym języku, domyślne komunikaty o błędach najprawdopodobniej nam wystarczą — podają one przyczynę błędu, a także pokazują wiersze kodu aktywne w momencie wystąpienia błędu.Więcej informacji na temat debugowania znajduje się w ramce „Debugowanie kodu w Pythonie” na końcu tego rozdziału.

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

```
L = [1, 2]                                # Utworzenie listy dwuelementowej  
L.append(L)                               # Dodanie L jako elementu do samej siebie  
L                                         # Wyświetlenie L
```

Co się dzieje? We wszystkich nowszych wersjach Pythona można zobaczyć dziwnie wyglądające dane wyjściowe, które omówimy w dodatku z rozwiązaniami do ćwiczeń, a które

okażą się mieć większy sens po omówieniu referencji w dalszej części książki. Osobom z wersją Pythona starszą od 1.5.1 na większości platform najprawdopodobniej pomoże kombinacja klawiszy *Ctrl+C*. Dlaczego określona wersja Pythona reaguje w taki sposób na powyższy kod?



Osoby z wersją Pythona starszą od 1.5.1 (miejmy nadzieję, że dzisiaj nie ma już takich osób!) powinny przed wykonaniem testu upewnić się, że ich komputer jest w stanie zatrzymać program za pomocą jakiegoś rodzaju kombinacji przerywającej. W przeciwnym razie powinny się przygotować na naprawdę długie oczekiwanie.

7. *Dokumentacja*. Należy poświęcić przynajmniej siedemnaście minut na przeglądanie bibliotek Pythona oraz dokumentacji w celu sprawdzenia, jakie narzędzia dostępne są w tej bibliotece i jaka jest struktura zbioru dokumentacji. By zapoznać się z lokalizacją najważniejszych zagadnień w dokumentacji, potrzeba przynajmniej tyle czasu; kiedy będziemy to mieli za sobą, o wiele łatwiej będzie nam znaleźć to, czego potrzebujemy. Dokumentację można znaleźć za pomocą opcji *Python* z menu *Start* (w przypadku systemu Windows), w opcji *Python Docs* menu rozwijanego *Help* w IDLE, a także w Internecie, pod adresem <http://www.python.org/doc>. Więcej informacji na temat dostępnych źródeł dokumentacji (włącznie z PyDoc i funkcją *help*) znajduje się w rozdziale 15. Osoby mające chwilę wolnego czasu mogą się udać na oficjalną stronę Pythona, a także przejrzeć witrynę z repozytorium rozszerzeń zewnętrznych PyPy. Kluczowymi zasobami są dokumentacja z Python.org i strony z wyszukiwaniem.

Typy i operacje

Wprowadzenie do typów obiektów Pythona

Niniejszy rozdział rozpoczyna naszą wycieczkę po języku Python. W pewnym sensie w Pythonie „robi się coś z różnymi rzeczami”. To „coś” ma postać operacji (działan), takich jak dodawanie czy konkatenacja, natomiast „różne rzeczy” to obiekty, na których wykonuje się owe operacje. W tej części książki skupimy się właśnie na owych „różnych rzeczach”, jak również na tym, co mogą z nimi robić nasze programy.

Mówiąc bardziej formalnym językiem, w Pythonie dane przybierają postać *obiektów* — albo wbudowanych obiektów udostępnianych przez Pythona, albo obiektów tworzonych za pomocą Pythona lub innych narzędzi zewnętrznych, takich jak biblioteki rozszerzeń języka C. Choć definicję tę nieco później rozbudujemy, obiekty są generalnie fragmentami pamięci z wartościami i zbiorami powiązanych operacji.

Ponieważ obiekty są najbardziej podstawowym elementem Pythona, ten rozdział rozpoczniemy od przeglądu obiektów wbudowanych w sam język.

Tytułem wstępu warto jednak najpierw ustalić, jak niniejszy rozdział wpisuje się w całość Pythona. Programy napisane w Pythonie można rozbić na moduły, instrukcje, wyrażenia i obiekty — w następujący sposób:

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

Omówienie modułów zamieszczone w rozdziale 3. uwzględnia najwyższy poziom w tej hierarchii. Rozdziały tej części książki odwołują się do najbliższego jej poziomu, czyli wbudowanych obiektów oraz wyrażeń, które tworzy się w celu korzystania z tych obiektów.

Po co korzysta się z typów wbudowanych?

Osoby używające języków niższego poziomu, takich jak C czy C++, wiedzą, że większość ich pracy polega na implementowaniu *obiektów* — znanych również jako *struktury danych* — tak by reprezentowały one komponenty w dziedzinie naszej aplikacji. Konieczne jest rozplanowanie struktur pamięci, zarządzanie przydzielaniem pamięci czy zaimplementowanie procedur wyszukiwania i dostępu. Te zadania są tak żmudne (i podatne na błędy), na jakie wyglądają, i zazwyczaj odciągają programistę od prawdziwych celów programu.

W typowych programach napisanych w Pythonie większość tej przyziemnej pracy nie jest konieczna. Ponieważ Python udostępnia typy obiektów jako nieodłączną część samego języka, zazwyczaj nie istnieje konieczność kodowania implementacji obiektów przed rozpoczęciem rozwiązywania prawdziwych problemów. Tak naprawdę, o ile oczywiście nie mamy potrzeby korzystania ze specjalnych metod przetwarzania, które nie są dostępne w obiektach wbudowanych, prawie zawsze lepiej będzie skorzystać z gotowego typu obiektu, zamiast tworzyć własne. Poniżej znajduje się kilka przyczyn takiego stanu rzeczy.

- **Obiekty wbudowane sprawiają, że programy łatwo się pisze.** W przypadku prostych zadań obiekty wbudowane często wystarczą nam do stworzenia struktur właściwych dla określonych problemów. Od ręki dostępne są narzędzia o sporych możliwościach, jak zbiory (listy) i tabele, które można przeszukiwać (słowniki). Wiele zadań można wykonać, korzystając z samych obiektów wbudowanych.
- **Obiekty wbudowane są komponentami rozszerzeń.** W przypadku bardziej zaawansowanych zadań być może nadal konieczne będzie udostępnianie własnych obiektów, wykorzystywanie klas Pythona czy interfejsów języka C. Jednak jak okaże się w dalszej części książki, obiekty implementowane ręcznie są często zbudowane na bazie typów wbudowanych, takich jak listy czy słowniki. Strukturę danych stosu można na przykład zaimplementować jako klasę zarządzającą wbudowaną listą lub dostosowującą tę listę do własnych potrzeb.
- **Obiekty wbudowane często są bardziej wydajne od własnych struktur danych.** Wbudowane obiekty Pythona wykorzystują już zoptymalizowane algorytmy struktur danych, które zostały zaimplementowane w języku C w celu zwiększenia szybkości ich działania. Choć możemy samodzielnie napisać podobne typy obiektów, zazwyczaj trudno nam będzie osiągnąć ten sam poziom wydajności, jaki udostępniają obiekty wbudowane.
- **Obiekty wbudowane są standardową częścią języka.** W pewien sposób Python zapożycza zarówno od języków opierających się na obiektach wbudowanych (jak na przykład LISP), jak i języków, w których to programista udostępnia implementacje narzędzi czy własnych platform (jak C++). Choć można w Pythonie implementować własne, unikalne typy obiektów, nie trzeba tego robić, by zacząć programować w tym języku. Co więcej, ponieważ obiekty wbudowane są standardem, zawsze pozostaną one takie same. Rozwiązania własnościowe zazwyczaj mają tendencję do zmian ze strony na stronę.

Innymi słowy, obiekty wbudowane nie tylko ułatwiają programowanie, ale mają także większe możliwości i są bardziej wydajne od większości tego, co tworzy się od podstaw. Bez względu na to, czy zdecydujemy się implementować nowe typy obiektów, obiekty wbudowane stanowią podstawę każdego programu napisanego w Pythonie.

Najważniejsze typy danych w Pythonie

W tabeli 4.1 zaprezentowano przegląd wbudowanych obiektów Pythona wraz ze składnią wykorzystywaną do kodowania ich *literalów* — czyli wyrażeń generujących te obiekty.¹ Niektóre z typów powinny dla osób znających inne języki programowania wyglądać znajomo. Liczby i łańcuchy znaków reprezentują, odpowiednio, wartości liczbowe i tekstowe. Pliki udostępniają natomiast interfejsy służące do przetwarzania plików przechowywanych na komputerze.

Tabela 4.1. Przegląd obiektów wbudowanych Pythona

Typ obiektu	Przykładowy literał (tworzenie)
Liczby	1234, 3.1415, 3+4j, Decimal, Fraction
Łańcuchy znaków	'mielonka', "Brian", b'a\x01c'
Listy	[1, [2, 'trzy'], 4]
Słowniki	{'jedzenie': 'mielonka', 'smak': 'mniam'}
Krotki	(1, 'mielonka', 4, 'U')
Pliki	myfile = open('jajka', 'r')
Zbiory	set('abc'), {'a', 'b', 'c'}
Inne typy podstawowe	Wartości Boolean, typy, None
Typy jednostek programu	Funkcje, moduły, klasy (część IV, V i VI książki)
Typy powiązane z implementacją	Kod skompilowany, śladы stosu (część IV i VII książki)

Tabela 4.1 nie jest kompletna, ponieważ *wszystko*, co przetwarzamy w programie napisanym w Pythonie, jest tak naprawdę rodzajem obiektu. Kiedy na przykład wykonujemy w Pythonie dopasowanie tekstu do wzorca, tworzymy obiekty wzorców, natomiast kiedy tworzymy skrypty sieciowe, wykorzystujemy obiekty gniazd. Te pozostałe typy obiektów tworzy się przede wszystkim za pomocą importowania i wykorzystywania modułów; każdy z nich wiąże się z pewnym typem zachowania.

Jak zobaczymy w dalszych częściach książki, *jednostki programów*, takie jak funkcje, moduły i klasy, także są w Pythonie obiektami — są one tworzone za pomocą instrukcji oraz wyrażeń, takich jak `def`, `class`, `import` czy `lambda`, i można je swobodnie przekazywać w skryptach bądź przechowywać w innych obiektach. Python udostępnia również zbiór *typów powiązanych z implementacją*, takich jak obiekty skompilowanego kodu, które są zazwyczaj bardziej przedmiotem zainteresowania osób tworzących narzędzia niż twórców aplikacji. Zostaną one omówione w późniejszych częściach książki.

Pozostałe typy obiektów z tabeli 4.1 nazywane są zazwyczaj *typami podstawowymi*, ponieważ są one tak naprawdę wbudowane w sam język. Oznacza to, że istnieje określona składnia wyrażeń służąca do generowania większości z nich. Na przykład kiedy wykonamy poniższy kod:

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

¹ W niniejszej książce pojęcie *literał* oznacza po prostu wyrażenie, którego składnia generuje obiekt — czasami nazywane również *stałą*. Warto zauważyć, że „stała” nie oznacza wcale obiektów czy zmiennych, które nigdy nie mogą być zmienione (czyli pojęcie to nie ma związku z `const` z języka C++ czy określeniem „niezmiennej” [ang. *immutable*] z Pythona — zagadnienie to omówione zostanie w dalszej części rozdziału).

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

Co równie ważne: kiedy tworzymy jakiś obiekt, wiążemy go z określonym zbiorem operacji. Na łańcuchach znaków można wykonywać tylko operacje dostępne dla łańcuchów znaków, natomiast na listach — tylko te dla list. Jak się za chwilę okaże, Python jest językiem z *typami dynamicznymi* (to znaczy automatycznie przechowuje za nas informacje o typach, zamiast wymagać kodu z deklaracją), jednak jego typy są *silne* (to znaczy na obiekcie można wykonać tylko te operacje, które są poprawne dla określonego typu).

Z funkcjonalnego punktu widzenia typy obiektów z tabeli 4.1 są bardziej ogólne i mają większe możliwości, niż bywa to w innych językach. Jak się okaże, już same listy i słowniki mają wystarczająco duże możliwości, by zlikwidować większość pracy związanej z obsługą zbiorów i wyszukiwania w językach niższego poziomu. Listy udostępniają uporządkowane zbory innych obiektów, natomiast słowniki przechowują obiekty i ich klucze. Oba typy danych mogą być zagnieżdżane, mogą rosnąć i kurczyć się na życzenie oraz mogą zawierać obiekty dowolnego typu.

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

Liczby

Dla osób, które zajmowały się już programowaniem czy tworzeniem skryptów, niektóre typy danych z tabeli 4.1 będą wyglądały znajomo. Nawet dla osób niemających nic wspólnego z programowaniem liczby wyglądają dość prosto. Zbiór podstawowych obiektów Pythona obejmuje typowe rodzaje liczb: całkowite (liczby bez części ułamkowej), zmiennoprzecinkowe (w przybliżeniu liczby z przecinkiem), a także bardziej egzotyczne typy liczbowe (liczby zespolone z liczbami urojonymi, liczby stałoprzecinkowe, liczby wymierne z mianownikiem i licznikiem, a także pełne zbioru).

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

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

Warto zwrócić uwagę na wynik ostatniego działania. Typ liczby całkowitej Pythona 3.0 automatycznie udostępnia dodatkową precyzję dla tak dużych liczb, kiedy jest to potrzebne (w Pythonie 2.6 osobny typ długiej liczby całkowitej w podobny sposób obsługiwał liczby zbyt duże dla zwykłego typu liczby całkowitej). Można na przykład w Pythonie obliczyć $2^{1000000}$ do potęgi 1000000 jako liczbę całkowitą (choć pewnie lepiej byłoby nie wyświetlać wyniku tego działania — z ponad trzystoma tysiącami cyfr będzie mieli trochę poczekać!).

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

Kiedy zaczniemy eksperymentować z liczbami zmiennoprzecinkowymi, z pewnością natknimy się na coś, co na pierwszy rzut oka może wyglądać nieco dziwnie:

```
>>> 3.1415 * 2 # repr: jako kod  
6.2830000000000004  
>>> print(3.1415 * 2) # str: w postaci przyjaznej dla użytkownika  
6.283
```

Pierwszy wynik nie jest błędem — to kwestia sposobu wyświetlania. Okazuje się, że każdy obiekt można wyświetlić na dwa sposoby — z pełną precyzją (jak w pierwszym wyniku powyżej) oraz w formie przyjaznej dla użytkownika (jak w drugim wyniku). Pierwsza postać znana jest jako `repr` obiektu (jak w kodzie), natomiast druga jest przyjazną dla użytkownika `str`. Różnica ta zacznie mieć znaczenie, kiedy przejdziemy do używania klas. Na razie, kiedy coś będzie dziwnie wyglądało, należy wyświetlić to za pomocą wywołania wbudowanej instrukcji `print`.

Poza wyrażeniami w Pythonie znajduje się kilka przydatnych modułów liczbowych. *Moduły* są po prostu pakietami dodatkowych narzędzi, które musimy zaimportować, by móc z nich skorzystać.

```
>>> import math  
>>> math.pi  
3.1415926535897931  
>>> math.sqrt(85)  
9.219544572928871
```

Moduł `math` zawiera bardziej zaawansowane narzędzia liczbowe w postaci funkcji, natomiast moduł `random` wykonuje generowanie liczb losowych, a także losowe wybieranie (tutaj z listy Pythona omówionej w dalszej części rozdziału):

```
>>> import random  
>>> random.random()  
0.59268735266273953  
>>> random.choice([1, 2, 3, 4])  
1
```

Python zawiera również bardziej egzotyczne obiekty liczb, takie jak liczby zespolone, liczby stałoprzecinkowe, liczby wymierne, a także zbiory i wartości Boolean (logiczne). Można również znaleźć różne dodatkowe rozszerzenia na licencji open source (na przykład dla macierzy czy wektorów). Szczegółowe omówienie tych typów zostawimy sobie na później.

Jak na razie omawialiśmy Pythona w funkcji prostego kalkulatora. Żeby jednak oddać sprawiedliwość jego typom wbudowanym, warto przejść do omówienia łańcuchów znaków.

Łańcuchy znaków

Łańcuchy znaków (ang. *strings*) wykorzystywane są do przechowywania informacji tekstowych, a także dowolnych zbiorów bajtów. Są pierwszym przykładem tego, co w Pythonie znane jest pod nazwą *sekwencji* — czyli uporządkowanych zbiorów innych obiektów. Sekwencje zachowują porządek zawieranych elementów od lewej do prawej strony. Elementy te są przechowywane i pobierane zgodnie z ich pozycją względną. Mówiąc dokładnie, łańcuchy znaków to sekwencje łańcuchów składających się z pojedynczych znaków. Pozostałe typy sekwencji to omówione później listy oraz krotki.

Operacje na sekwencjach

Jako sekwencje łańcuchy znaków obsługują operacje zakładające pozycyjne uporządkowanie elementów. Jeśli na przykład mamy łańcuch czteroznakowy, możemy zweryfikować jego długość za pomocą wbudowanej funkcji `len` i pobrać jego elementy za pomocą wyrażeń *indeksujących*:

```
>>> S = 'Mielonka'  
>>> len(S)                                     # Długość  
8  
>>> S[0]                                       # Pierwszy element w S, indeksowanie rozpoczyna się od zera  
'M'  
>>> S[1]                                       # Drugi element od lewej  
'i'
```

W Pythonie indeksy zakodowane są jako wartość przesunięcia od początku łańcucha, dlatego rozpoczynają się od zera. Pierwszy element znajduje się pod indeksem 0, kolejny pod indeksem 1 i tak dalej.

Warto tutaj zwrócić uwagę na przypisanie łańcucha znaków do *zmiennej* o nazwie `S`. Szczegółowe omówienie tego, jak to działa, odłożymy na później (zwłaszcza do rozdziału 6.), natomiast warto wiedzieć, że zmiennych Pythona nigdy nie trzeba deklarować z wyprzedzeniem. Zmienna tworzona jest, kiedy przypisujemy do niej wartość; można do niej przypisać dowolny typ obiektu i zostanie zastąpiona swoją wartością, kiedy pojawi się w wyrażeniu. Przed użyciem jej wartości musi najpierw zostać przypisana. Na cele niniejszego rozdziału wystarczy nam wiedza, że by móc zapisać obiekt w celu późniejszego użycia, musimy przypisać ten obiekt do zmiennej.

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

```
>>> S[-1]                                         # Pierwszy element od końca S  
'a'  
>>> S[-2]                                         # Drugi element od końca S  
'k'
```

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

```
>>> S[-1]                                         # Ostatni element w S  
'a'  
>>> S[len(S)-1]                                 # Indeks ujemny zapisany w bardziej skomplikowany sposób  
'a'
```

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

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

```
>>> S                                     # Łąćuch z ośmioma znakami
'Mielonka'
>>> S[1:3]                                # Wycinek z S z indeksami od 1 do 2 (bez 3)
'ie'
```

Wycinki najłatwiej jest sobie wyobrazić jako sposób pozwalający na ekstrakcję całej *kolumny* z łańcucha znaków za jednym razem. Ich ogólna forma, $X[I:J]$, oznacza: „Zwróć wszystko z X od przesunięcia I aż do przesunięcia J , ale bez niego”. Wynik zwracany jest w nowym obiekcie. Druga operacja z listingu wyżej zwraca na przykład wszystkie znaki w łańcuchu znaków S od przesunięcia 1 do przesunięcia 2 (czyli 3–1) jako nowy łańcuch znaków. Wynikiem jest wycinek składający się z dwóch znaków znajdujących się w środku łańcucha S .

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

```
>>> S[1:]                                 # Wszystko po pierwszym znakiem (1:len(S))
'ielonka'
>>> S                                     # Łąćuch S się nie zmienił
'Mielonka'
>>> S[0:7]                                # Wszystkie elementy bez ostatniego
'Mielonk'
>>> S[:7]                                  # To samo co S[0:7]
'Mielonk'
>>> S[:-1]                                 # Wszystkie elementy bez ostatniego w łatwiejszej postaci
'Mielonk'
>>> S[:]                                   # Całość S jako kopia najwyższego poziomu (0:len(S))
'Mielonka'
```

Warto zwrócić uwagę na to, że do ustalenia granic wycinków mogą również zostać wykorzystane ujemne wartości przesunięcia. Widać również, że ostatnia operacja w rezultacie kopiuje cały łańcuch znaków. Jak się okaże później, kopowanie łańcucha znaków nie ma sensu, jednak to polecenie może się przydać w przypadku sekwencji takich, jak listy.

Wreszcie tak samo jak pozostałe sekwencje, łańcuchy znaków obsługują również *konkatenację* (ang. *concatenation*) z użyciem znaku + (łączącą dwa łańcuchy znaków w jeden), a także *powtórzanie* (ang. *repetition*), czyli zbudowanie nowego łańcucha znaków poprzez powtórzenie innego:

```
>>> S                                     # Konkatenacja
'Mielonka'
>>> S + 'xyz'
'Mielonkaxyz'
>>> S                                     # S pozostaje bez zmian
'Mielonka'
>>> S * 8                                 # Powtóżenie
'MielonkaMielonkaMielonkaMielonkaMielonkaMielonkaMielonkaMielonka'
```

Warto zwrócić uwagę na to, że znak plusa (+) oznacza coś innego w przypadku różnych obiektów — dodawanie dla liczb, a konkatenację dla łańcuchów znaków. Jest to właściwość Pythona, którą w dalszej części książki będziemy nazywali *polimorfizmem*. Oznacza to, że każda

operacja uzależniona jest od typu obiektów, na jakich się ją wykonuje. Jak się okaże przy okazji omawiania typów dynamicznych, polimorfizm odpowiada za związkowość i elastyczność kodu napisanego w Pythonie. Ponieważ typy nie są ograniczone, operacja napisana w Pythonie może zazwyczaj automatycznie działać na wielu różnych typach obiektów, o ile obsługują one zgodny z nią interfejs (jak operacja+powyżej). W Pythonie jest to bardzo ważna koncepcja, do której jeszcze powrócimy.

Niezmienność

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

```
>>> S
'Mielonka'
>>> S[0] = 'z'                                     # Niezmienne obiekty nie mogą być modyfikowane
...pominięto tekst błędu...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]                                # Można jednak tworzyć wyrażenia budujące nowe obiekty
>>> S
'zielonka'
```

Każdy obiekt Pythona klasyfikowany jest jako niezmienny (niemodyfikowalny) bądź zmienny (modyfikowalny). Wśród typów podstawowych niezmienne są liczby, łańcuchy znaków oraz krotki. Listy i słowniki można dowolnie zmieniać w miejscu. Niezmienność można wykorzystać do zagwarantowania, że obiekt pozostanie stałym w czasie całego cyklu działania programu.

Metody specyficzne dla typu

Każda z omówionych dotychczas operacji na łańcuchach znaków jest tak naprawdę operacją na sekwencjach, która będzie działała na innych typach sekwencji Pythona, w tym na listach i krotkach. Poza tymi wspólnymi dla sekwencji operacjami łańcuchy znaków obsługują również własne operacje dostępne w formie *metod* (funkcji dołączanych do obiektu i wywoływanych za pomocą odpowiedniego wyrażenia).

Metoda `find` jest na przykład podstawową operacją działającą na podłańcuchach znaków (zwraca ona wartość przesunięcia przekazanego podłańcucha znaków lub `-1`, jeśli podłańcuch ten nie jest obecny). Metoda `replace` służy z kolei do globalnego odszukiwania i zastępowania.

```
>>> S.find('ie')                                    # Odnalezienie przesunięcia podłańcucha
1
>>> S
'Mielonka'

>>> S.replace('ie', 'XYZ')                         # Zastąpienie wystąpień podłańcucha innym
'MXYZlonka'
>>> S
'Mielonka'
```

I znów, mimo że w metodach łańcucha znaków występuje nazwa zmiennej, nie zmieniamy oryginalnych łańcuchów znaków, lecz tworzymy nowe. Ponieważ łańcuchy znaków są niezmienne, trzeba to robić właśnie w taki sposób. Metody łańcuchów znaków to pierwsza pomoc w przetwarzaniu tekstu w Pythonie. Inne dostępne metody pozwalają dzielić łańcuch znaków na podłańcuchy w miejscu wystąpienia ogranicznika (co przydaje się w analizie składniowej), zmieniać wielkość liter, sprawdzać zawartość łańcuchów znaków (cyfry, litery i inne znaki) lub usuwać białe znaki (ang. *whitespace*) z końca łańcucha.

```
>>> line = 'aaa,bbb,cccc,dd'
>>> line.split(',')
['aaa', 'bbb', 'cccc', 'dd']                                     # Podzielenie na ograniczniku na listę podłańcuchów
>>> S = 'mielonka'
>>> S.upper()                                                       # Konwersja na wielkie litery
'MIELONKA'

>>> S.isalpha()                                                     # Sprawdzenie zawartości: isalpha, isdigit
True

>>> line = 'aaa,bbb,cccc,dd\n'
>>> line = line.rstrip()                                           # Usunięcie białych znaków po prawej stronie
>>> line
'aaa,bbb,cccc,dd'
```

Łańcuchy znaków obsługują również zaawansowane operacje zastępowania znane jako *formatowanie*, dostępne zarówno w postaci wyrażeń (oryginalne rozwiązanie), jak i wywołania metody łańcuchów znaków (nowość w wersjach 2.6 oraz 3.0):

```
>>> '%s, jajka i %s' % ('mielonka', 'MIELONKA!')           # Wyrażenie formatujące (wszystkie wersje)
'mielonka, jajka i MIELONKA!'
>>> '{0}, jajka i {1}'.format('mielonka', 'MIELONKA!') # Metoda formatująca (2.6, 3.0)
'mielonka, jajka i MIELONKA!'
```

Jedna uwaga: choć operacje na sekwencjach są uniwersalne, metody takie nie są — choć niektóre typy współdzielą pewne nazwy metod. Metody łańcuchów znaków działają wyłącznie na łańcuchach znaków i na niczym innym. Generalnie zbiór narzędzi Pythona składa się z warstw — uniwersalne operacje, które dostępne są dla większej liczby typów danych, występują jako wbudowane funkcje lub wyrażenia (na przykład `len(X)`, `X[0]`), natomiast operacje specyficzne dla określonego typu są wywołaniami metod (jak `aString.upper()`). Odnalezienie odpowiednich narzędzi w tych kategoriach stanie się bardziej naturalne z czasem, natomiast w kolejnym podrozdziale znajduje się kilka wskazówek przydatnych już teraz.

Otrzymanie pomocy

Metody wprowadzone w poprzednim podrozdziale są reprezentatywną, choć dość niewielką próbką tego, co dostępne jest dla łańcuchów znaków. Generalnie w niniejszej książce omówienie metod obiektów nie jest kompletne. W celu uzyskania większej liczby szczegółów zawsze można wywołać wbudowaną funkcję `dir`, która zwraca listę wszystkich atrybutów dostępnych dla danego obiektu. Ponieważ metody są atrybutami obiektów, zostaną na tej liście wyświetcone. Zakładając, że zmienna `S` nadal jest łańcuchem znaków, oto jej atrybuty w Pythonie 3.0 (w wersji 2.6 nieznacznie się one różnią):

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__str__', '__subclasshook__']
```

```
↳ '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
↳ '__subclasshook__', '__formatter_field_name_split__', '__formatter_parser',
↳ 'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
↳ 'format', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
↳ 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
↳ 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
↳ 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
↳ 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Nazwy z dwoma znakami `_` nie będą nas raczej interesować aż do późniejszej części książki, kiedy będziemy omawiać przeciążanie operatorów w klasach. Reprezentują one implementację obiektu łańcucha znaków i są dostępne w celu ułatwienia dostosowania obiektu do własnych potrzeb. Ogólna reguła jest taka, że podwójne znaki `_` na początku i końcu to konwencja wykorzystywana w Pythonie do szczegółów implementacyjnych. Nazwy bez tych znaków są metodami, które można wywoływać na łańcuchach znaków.

Funkcja `dir` podaje same nazwy metod. Żeby dowiedzieć się, co te metody robią, można przekazać je do funkcji `help`.

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

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

Funkcja `help` jest jednym z kilku interfejsów do systemu kodu udostępnianego wraz z Pythonem i znanego pod nazwą *PyDoc* — narzędzia służącego do pobierania dokumentacji z obiektów. W dalszej części książki pokażemy, że PyDoc może również zwracać swoje raporty w formacie HTML.

Można również próbować uzyskać pomoc dla całego łańcucha znaków (na przykład `help(S)`), jednak w rezultacie otrzymamy więcej, niż chcielibyśmy zobaczyć, to znaczy informacje o każdej metodzie łańcuchów znaków. Lepiej jest pytać o określona metodę, jak na listingu powyżej.

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

Inne sposoby kodowania łańcuchów znaków

Dotychczas przyglądaliśmy się operacjom na sekwencjach na przykładzie łańcuchów znaków, a także metodom przeznaczonym dla tego typu danych. Python udostępnia również różne sposoby kodowania łańcuchów znaków, które bardziej szczegółowo omówione zostaną nieco później. Przykładowo znaki specjalne można reprezentować jako sekwencje ucieczki z ukośnikiem lewym.

```
>>> S = 'A\nB\tC'                                # \n to koniec wiersza, \t to tabulator
>>> len(S)                                      # Każde jest jednym znakiem
5

>>> ord('\n')                                    # \n jest bajtem z wartością binarną 10 w ASCII
10
```

```
>>> S = 'A\0B\0C'                                # \0, binarny bajt zerowy, nie kończy łańcucha znaków
>>> len(S)
5
```

Python pozwala na umieszczanie łańcuchów znaków zarówno w cudzysłowach, jak i apostrofach (oznaczają one to samo). Zawiera również wielowierszową postać literała łańcucha znaków umieszczaną w trzech cudzysłowach lub trzech apostrofach. Kiedy użyta zostanie ta forma, wszystkie wiersze są ze sobą łączone, a w miejscu złamania wiersza dodawane są odpowiednie znaki złamania wiersza. Jest to pewna konwencja syntaktyczna, która przydaje się do osadzania kodu HTML czy XML w skrypcie Pythona.

```
>>> msg = """aaaaaaaaaaaaaa
bbb '' bbbbbbbbbb " bbbbbbb ' bbbb
cccccccccccccc """
>>> msg
'\naaaaaaaaaaaaaa\nbbb\ '' \' bbbbbbbbbb " bbbbbbb ' bbbb\ncccccccccccccc'
```

Python obsługuje również „surowy” literał łańcucha znaków wyłączający mechanizm ucieczki z ukośnikiem lewym (rozpoczyna się on od litery r), a także format Unicode obsługujący znaki międzynarodowe. W wersji 3.0 podstawowy typ łańcucha znaków str obsługuje także znaki Unicode (co ma sens, biorąc pod uwagę, że tekst ASCII jest po prostu częścią Unicode), natomiast typ bytes reprezentuje surowe łańcuchy bajtowe. W Pythonie 2.6 Unicode jest odrębnym typem, natomiast str obsługuje zarówno łańcuchy osmiobitowe, jak i dane binarne. W wersji 3.0 zmieniły się także pliki zwracające i przyjmujące str dla tekstu i bytes dla danych binarnych. Ze wszystkimi specjalnymi formami łańcuchów znaków spotkamy się w kolejnych rozdziałach.

Dopasowywanie wzorców

Zanim przejdziemy dalej, warto odnotować, że żadna z metod łańcuchów znaków nie obsługuje przetwarzania tekstu opartego na wzorach. Dopasowywanie wzorców tekstowych jest zaawansowanym narzędziem pozostającym poza zakresem niniejszej książki, jednak osoby znające inne skryptowe języki programowania pewnie będą zainteresowane tym, że dopasowywanie wzorców w Pythonie odbywa się z wykorzystaniem modułu o nazwie re. Moduł ten zawiera analogiczne wywołania dla wyszukiwania, dzielenia czy zastępowania, jednak dzięki użyciu wzorców do określania podłańcuchów znaków możemy być o wiele bardziej ogólni.

```
>>> import re
>>> match = re.match('Witaj,[ \t]*(.*?Robinie', 'Witaj, sir Robinie')
>>> match.group(1)
'sir '
```

Powyższy przykład szuka podłańcucha znaków zaczynającego się od słowa Witaj,, po którym następuje od zera do większej liczby tabulatorów lub spacji, po nich dowolne znaki, które będą zapisane jako dopasowana grupa, a na końcu słowo Robinie. Kiedy taki podłańcuch zostanie odnaleziony, jego części dopasowane przez części wzorca umieszczone w nawiasach dostępne są jako grupy. Poniższy wzorzec wybiera na przykład trzy grupy rozdzielone ukośnikami prawymi:

```
>>> match = re.match('/(.*)/(.*)/(.*)', '/usr/home/drwal')
>>> match.groups()
('usr', 'home', 'drwal')
```

Dopasowywanie wzorców samo w sobie jest dość zaawansowanym narzędziem do przetwarzania tekstu, natomiast w Pythonie udostępniana jest nawet obsługa jeszcze bardziej

zaawansowanych technik przetwarzania tekstu i języka, w tym analizy składniowej XML i analizy języka naturalnego. Dość jednak informacji o łańcuchach znaków — czas przejść do kolejnego typu danych.

Listy

Obiekt listy jest w Pythonie najbardziej ogólnym rodzajem sekwencji dostępnym w tym języku. Listy to uporządkowane pod względem pozycji zbiory obiektów dowolnego typu; nie mają one ustalonej wielkości. Są również *zmienne* (ang. *mutable*) — w przeciwieństwie do łańcuchów znaków listy można modyfikować w miejscu, przypisując coś do odpowiednich wartości przesunięcia, a także za pomocą różnych metod.

Operacje na sekwencjach

Ponieważ listy są sekwencjami, obsługują wszystkie operacje na sekwencjach przedstawione przy okazji omawiania łańcuchów znaków. Jedyną różnicą jest to, że wynikiem zazwyczaj są listy, a nie łańcuchy znaków. Mając na przykład listę trzyelementową:

```
>>> L = [123, 'mielonka', 1.23]          # Trzyelementowa lista z elementami różnego typu
>>> len(L)                                # Liczba elementów listy
3
```

możemy ją zindeksować, sporządzić z niej wycinki i wykonać pozostałe operacje zaprezentowane na przykładzie łańcuchów znaków:

```
>>> L[0]                                  # Indeksowanie po pozycji
123
>>> L[:-1]                               # Wycinkiem listy jest nowa lista
[123, 'mielonka']
>>> L + [4, 5, 6]                         # Konkatenacja także zwraca nową listę
[123, 'mielonka', 1.23, 4, 5, 6]
>>> L                                     # Oryginalna lista pozostaje bez zmian
[123, 'mielonka', 1.23]
```

Operacje specyficzne dla typu

Listy Pythona powiązane są z tablicami obecnymi w innych językach programowania, jednak zazwyczaj mają większe możliwości. Przede wszystkim nie mają żadnych ograniczeń odnośnie typów danych — przykładowo lista z listingu powyżej zawiera trzy obiekty o zupełnie różnych typach (liczbę całkowitą, łańcuch znaków i liczbę zmiennoprzecinkową). Co więcej, wielkość listy nie jest określona. Oznacza to, że może ona rosnąć i kurczyć się zgodnie z potrzebami i w odpowiedzi na operacje specyficzne dla list.

```
>>> L.append('NI')                      # Rośnięcie: dodanie obiektu na końcu listy
>>> L
[123, 'mielonka', 1.23, 'NI']
>>> L.pop(2)                            # Kurczanie się: usunięcie obiektu ze środka listy
1.23
>>> L                                    # "del L[2]" również usuwa element z listy
[123, 'mielonka', 'NI']
```

Na powyższym listingu metoda `append` rozszerza rozmiar listy i wstawia na jej końcu element. Metoda `pop` (lub jej odpowiednik, instrukcja `del`) usuwa następnie element znajdujący się na pozycji o podanej wartości przesunięcia, co sprawia, że lista się kurczy. Pozostałe metody list pozwalają na wstawienie elementu w dowolnym miejscu listy (`insert`) czy usunięcie elementu o podanej wartości (`remove`). Ponieważ listy są zmienne, większość ich metod zmienia obiekt listy w miejscu, a nie tworzy nowy obiekt.

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']

>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

Zaprezentowana wyżej metoda listy o nazwie `sort` porządkuje listę w kolejności rosnącej, natomiast metoda `reverse` odwraca ją. W obu przypadkach lista modyfikowana jest w sposób bezpośredni.

Sprawdzanie granic

Choć listy nie mają ustalonej wielkości, Python nie pozwala na odnoszenie się do elementów, które nie istnieją. Indeksowanie poza końcem listy zawsze jest błędem, podobnie jak przypisywanie elementu poza jej końcem.

```
>>> L
[123, 'mielonka', 'NI']

>>> L[99]
...pominieto tekst bledu...
IndexError: list index out of range

>>> L[99] = 1
...pominieto tekst bledu...
IndexError: list assignment index out of range
```

Takie rozwiązanie jest celowe, ponieważ zazwyczaj błędem jest próba przypisania elementu poza końcem listy (szczególnie paskudne jest to w języku C, który nie sprawdza błędów w tak dużym stopniu jak Python). Zamiast w odpowiedzi po cichu powiększać listę, Python zgłasza błąd. By powiększyć listę, należy zamiast tego wywołać odpowiednią metodę, taką jak `append`.

Zagnieżdżanie

Jedną z miłych właściwości podstawowych typów danych w Pythonie jest obsługa dowolnego zagnieżdżania. Możliwe jest zagnieżdżanie tych obiektów w dowolnej kombinacji i na dowolną głębokość (czyli można na przykład utworzyć listę zawierającą słownik, który z kolei zawiera kolejną listę). Bezpośrednim zastosowaniem tej właściwości jest reprezentowanie w Pythonie macierzy czy, inaczej, tablic wielowymiarowych. Lista mieszcząca zagnieżdżone listy doskonale się sprawdzi w prostych aplikacjach:

```
>>> M = [[1, 2, 3],           # Macierz 3x3 w postaci list zagnieżdzonych
          [4, 5, 6],           # Kod może się rozciągać na kilka wierszy, jeśli znajduje się w nawiasach
          [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Powyżej zakodowaliśmy listę zawierającą trzy inne listy. W rezultacie otrzymujemy reprezentację macierzy liczb 3×3 . Dostęp do takiej struktury można uzyskać na wiele sposobów:

```
>>> M[1]                                     # Pobranie drugiego wiersza  
[4, 5, 6]  
  
>>> M[1][2]                                 # Pobranie drugiego wiersza, a z niego trzeciego elementu  
6
```

Pierwsza operacja pobiera cały drugi wiersz, natomiast druga pobiera trzeci element z tego wiersza. Łączenie ze sobą indeksów wciąża nas coraz głębiej i głębiej w zagnieżdżoną strukturę.²

Listy składane

Poza operacjami dla sekwencji i metodami list Python zawiera bardziej zaawansowaną operację znaną pod nazwą *list comprehension* (ang. *list comprehension*), która świetnie się przydaje do przetwarzania struktur takich, jak nasza macierz. Przypuśćmy na przykład, że potrzebujemy pobrać drugą kolumnę naszej prostej macierzy. Łatwo jest za pomocą zwykłego indeksowania pobierać wiersze, ponieważ macierz przechowywana jest wierszami. Podobnie łatwo jest pobrać kolumnę za pomocą wyrażenia listy składanej:

```
>>> col2 = [row[1] for row in M]           # Zebranie elementów z drugiej kolumny  
>>> col2  
[2, 5, 8]  
  
>>> M                                      # Macierz pozostaje bez zmian  
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Listy składane pochodzą z notacji zbiorów. Służą do budowania nowej listy poprzez wykonanie wyrażenia na każdym elemencie po kolej, jeden po drugim, od lewej strony do prawej. Listy składane kodowane są w nawiasach kwadratowych (by podkreślić fakt, że tworzą listę) i składają się z wyrażenia i konstrukcji pętli, które dzielą ze sobą nazwę zmiennej (tutaj `row`). Powyższa lista składana oznacza mniej więcej: „Zwróć `row[1]` dla każdego wiersza w macierzy `M` w nowej liście”. Wynikiem jest nowa lista zawierająca drugą kolumnę macierzy.

Listy składane mogą w praktyce być bardziej skomplikowane:

```
>>> [row[1] + 1 for row in M]             # Dodanie 1 do każdego elementu w drugiej kolumnie  
[3, 6, 9]  
  
>>> [row[1] for row in M if row[1] % 2 == 0] # Odfiltrowanie elementów nieparzystych  
[2, 8]
```

Pierwsza operacja z listingu powyżej dodaje 1 do każdego pobranego elementu, natomiast druga wykorzystuje wyrażenie `if` do odfiltrowania liczb nieparzystych z wyniku, używając do tego reszty z dzielenia (wyrażenia `z %`). Listy składane tworzą nowe listy wyników, jednak mogą również zostać wykorzystane do iteracji po dowolnym obiekcie, na którym jest to możliwe. Listing poniżej prezentuje użycie list składanych do przejęcia zakodowanej listy współrzędnych oraz łańcucha znaków:

² Taka struktura macierzy jest dobra dla zadań na małą skalę, jednak w przypadku poważniejszego przetwarzania liczb lepiej będzie skorzystać z jednego z rozszerzeń liczbowych do Pythona, takiego jak system *NumPy* na licencji open source. Takie narzędzia są w stanie przechowywać i przetwarzać duże macierze o wiele bardziej wydajnie od zagnieżdżonej struktury list. O NumPy mówi się, że zmienia Pythona w darmowy i bardziej zaawansowany odpowiednik systemu MatLab, a organizacje takie, jak NASA, Los Alamos czy JPMorgan Chase wykorzystują to narzędzie w swojej pracy naukowej i finansowej. Więcej informacji na ten temat można znaleźć w Internecie.

```

>>> diag = [M[i][i] for i in [0, 1, 2]]      # Pobranie przekątnej z macierzy
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'mielonka']    # Powtórzenie znaków w łańcuchu
>>> doubles
['mm', 'ii', 'ee', 'll', 'oo', 'nn', 'kk', 'aa']

```

Listy składane i pokrewne konstrukcje, takie jak funkcje wbudowane `map` oraz `filter`, są trochę zbyt zaawansowane, by omawiać je teraz bardziej szczegółowo. Najważniejsze w tym krótkim wprowadzeniu jest to, że Python zawiera zarówno narzędzia podstawowe, jak i bardziej zaawansowane. Listy składane są opcjonalne, jednak w praktyce często okazują się przydatne i nierzadko pozwalają zyskać na szybkości przetwarzania. Działają na każdym typie danych będącym sekwencją, a także na pewnych typach, które sekwencjami nie są. Wróćmy do nich w dalszej części książki.

Jako zapowiedź tego, co będzie dalej, możemy napisać, że w nowszych wersjach Pythona składnia złożień w nawiasach może być wykorzystywana do tworzenia *generatorów* produkujących wyniki na żądanie (wbudowana funkcja `sum` sumuje na przykład elementy w sekwencji):

```

>>> G = (sum(row) for row in M)           # Utworzenie generatora sum wierszy
>>> next(G)                            # iter(G) nie jest tu wymagane
6
>>> next(G)                            # Wykonanie protokołu iteracji
15

```

Funkcja wbudowana `map` wykonuje podobne zadanie, generując wyniki wykonania funkcji dla elementów. Opakowanie jej w wywołanie `list` wymusza zwrócenie wszystkich wartości w Pythonie 3.0.

```

>>> list(map(sum, M))                  # Wykonanie funkcji sum dla elementów M
[6, 15, 24]

```

W Pythonie 3.0 składnię złożień można także wykorzystać do tworzenia zbiorów oraz słowników:

```

>>> {sum(row) for row in M}           # Utworzenie zbioru sum wierszy
{24, 6, 15}
>>> {i : sum(M[i]) for i in range(3)}  # Utworzenie tabeli klucz-wartość sum wierszy
{0: 6, 1: 15, 2: 24}

```

Tak naprawdę listy, zbiory oraz słowniki można w wersji 3.0 tworzyć za pomocą złożień:

```

>>> [ord(x) for x in 'mieloonka']      # Lista numerów porządkowych znaków
[109, 105, 101, 108, 111, 111, 110, 107, 97]
>>> {ord(x) for x in 'mieloonka'}       # Zbiory usuwają duplikaty
{97, 101, 105, 107, 108, 109, 110, 111}
>>> {x: ord(x) for x in 'mieloonka'}    # Klucze słownika są unikalne
{'a': 97, 'e': 101, 'i': 105, 'k': 107, 'm': 109, 'l': 108, 'o': 111, 'n': 110}

```

By jednak zrozumieć obiekty, takie jak generatory, zbiory oraz słowniki, musimy przejść nieco dalej.

Słowniki

Słowniki są w Pythonie czymś zupełnie innym — nie są zupełnie sekwencjami, są za to znane jako *odwzorowania* (ang. *mapping*). Odwzorowania są również zbiorami innych obiektów, jednak obiekty te przechowują po kluczu, a nie ich pozycji względnej. Odwzorowania nie zachowują

żadnej niezawodnej kolejności od lewej do prawej strony, po prostu łączą klucze z powiązanymi z nimi wartościami. Słowniki, jedyny typ odwzorowania w zbiorze typów podstawowych Pythona, są również zmienne — mogą być modyfikowane w miejscu i podobnie do list, mogą rosnąć i kurczyć się na życzenie.

Operacje na odwzorowaniach

Kiedy słowniki zapisze się w formie literałów, znajdują się w nawiasach klamrowych i składają się z serii par *klucz*: *wartość*. Słowniki są przydatne wtedy, gdy chcemy powiązać zbiór wartości z kluczami — na przykład opisać właściwości czegoś. Rozważmy na przykład poniższy słownik składający się z trzech elementów (których kluczami będą *jedzenie*, *ilość* i *kolor*):

```
>>> D = {'jedzenie': 'Mielonka', 'ilość': 4, 'kolor': 'różowy'}
```

W celu zmiany wartości powiązanych z kluczami słownik można zindeksować po kluczu. Operacja indeksowania słownika wykorzystuje tę samą składnię co wykorzystywana w sekwencjach, jednak elementem znajdującym się w nawiasach kwadratowych jest klucz, a nie pozycja względna.

```
>>> D['jedzenie'] # Pobranie wartości klucza "jedzenie"
'Mielonka'

>>> D['ilość'] += 1 # Dodanie 1 do wartości klucza "ilość"
>>> D
{'jedzenie': 'Mielonka', 'kolor': 'różowy', 'ilość': 5}
```

Choć forma literala z nawiasami klamrowymi jest czasami używana, częściej widzi się słowniki tworzone w inny sposób. Poniższy przykład rozpoczyna się od pustego słownika, który jest następnie wypełniany po jednym kluczu na raz. W przeciwieństwie do przypisania poza granicami listy, co jest zakazane, przypisania do nowych kluczy słownika tworzą te klucze.

```
>>> D = {}
>>> D['imię'] = 'Robert' # Tworzenie kluczy przez przypisanie
>>> D['zawód'] = 'programista'
>>> D['wiek'] = 40

>>> D
{'wiek': 40, 'zawód': 'programista', 'imię': 'Robert'}

>>> print(D['imię'])
Robert
```

Powyżej wykorzystaliśmy klucze jako nazwy pól w spisie opisującym kogoś. W innych zastosowaniach słowniki mogą być również wykorzystywane do zastępowania operacji wyszukiwania — zindeksowanie słownika po kluczu jest często najszybszą metodą zakodowania wyszukiwania w Pythonie. Jak dowiemy się później, słowniki można także tworzyć przez przekazywanie argumentów będących słowami kluczowymi do nazwy typu, jak w przykładzie `dict(imię='Robert', zawód='programista', wiek=40)` tworzącym ten sam słownik.

Zagnieżdżanie raz jeszcze

W poprzednim przykładzie słownik wykorzystaliśmy do opisania hipotetycznej osoby za pomocą trzech kluczy. Założmy jednak, że informacje są bardziej złożone. Być może konieczne będzie zanotowanie imienia i nazwiska, a także kilku zawodów czy tytułów. Prowadzi to do

innego zastosowania zagnieźdzania obiektów w praktyce. Poniższy słownik — zakodowany od razu jako literal — zawiera bardziej ustrukturyzowane informacje:

```
>>> rec = {'dane osobowe': {'imię': 'Robert', 'nazwisko': 'Zielony'},  
           'zawód': ['programista', 'inżynier'],  
           'wiek': 40.5}
```

W powyższym przykładzie znowu mamy na górze słownik z trzema kluczami (o kluczach dane osobowe, zawód oraz wiek), natomiast wartości stają się nieco bardziej skomplikowane. Zagnieżdzony słownik z danymi osobowymi może pomieścić kilka informacji, natomiast zagnieżdzona lista z zawodem mieści kilka ról i można ją w przyszłości rozszerzyć. Dostęp do elementów tej struktury można uzyskać w podobny sposób jak w przypadku pokazanej wcześniej macierzy, jednak tym razem indeksami będą klucze słownika, a nie wartości przesunięcia listy.

```
>>> rec['dane osobowe']                                     # 'dane osobowe' to zagnieżdzony słownik  
{'nazwisko': 'Zielony', 'imię': 'Robert'}  
  
>>> rec['dane osobowe']['nazwisko']                      # Indeksowanie zagnieżdzonego słownika  
'Zielony'  
  
>>> rec['zawód']                                         # 'zawód' to zagnieżdzona lista  
['programista', 'inżynier']  
>>> rec['zawód'][-1]                                       # Indeksowanie zagnieżdzonej listy  
'inżynier'  
  
>>> rec['zawód'].append('leśniczy')                      # Rozszerzenie listy zawodów Roberta  
>>> rec  
{'wiek': 40.5, 'zawód': ['programista', 'inżynier', 'leśniczy'], 'dane osobowe':  
    {'nazwisko': 'Zielony', 'imię': 'Robert'}}}
```

Warto zwrócić uwagę na to, jak ostatnia operacja rozszerza osadzoną listę z zawodami. Ponieważ lista zawodów jest fragmentem pamięci oddzielnym od zawierającego ją słownika, może dowolnie rosnąć i kurczyć się (rozkład pamięci obiektów omówiony zostanie w dalszej części książki).

Prawdziwym powodem pokazania tego przykładu jest chęć zademonstrowania *elastyczności* podstawowych typów obiektów Pythona. Jak widać, zagnieżdzanie pozwala na budowanie skomplikowanych struktur informacji w sposób łatwy i bezpośredni. Zbudowanie podobnej struktury w języku niskiego poziomu, takim jak C, byłoby żmudne i wymagałoby o wiele większej ilości kodu. Musielibyśmy zaprojektować i zadeklarować układ struktury i tablic, wypełnić je wartościami i wreszcie połączyć wszystko ze sobą. W Pythonie wszystko to dzieje się automatycznie — jedno wyrażenie tworzy za nas całą zagnieżdzoną strukturę obiektów. Tak naprawdę jest to jedna z najważniejszych zalet języków skryptowych, takich jak Python.

Co równie ważne, w języku niższego poziomu musielibyśmy uważnie czyścić przestrzeń zajmowaną przez obiekty, które jej już dłużej nie potrzebują. W Pythonie, kiedy znika ostatnia referencja do obiektu (na przykład gdy do zmiennej przypisze się coś innego), całe miejsce w pamięci zajmowane przez tę strukturę obiektu jest automatycznie zwalniane.

```
>>> rec = 0                                                 # Miejsce zajmowane przez obiekt zostaje odzyskane
```

Z technicznego punktu widzenia Python korzysta z czegoś, co znane jest pod nazwą *czyszczenia pamięci* (ang. *garbage collection*). Nieużywana pamięć jest czyszczona w miarę wykonywania programu, co zwalnia nas z odpowiedzialności za zarządzanie takimi szczegółami w naszym kodzie. W Pythonie miejsce to jest odzyskiwane natychmiast, kiedy tylko zniknie ostatnia referencja do

obiektu. W dalszej części książki omówimy tę kwestię bardziej szczegółowo. Na razie powinna nam wystarczyć wiedza, że możemy swobodnie korzystać z obiektów, bez konieczności troszczenia się o tworzenie dla nich miejsca w pamięci i zwalnianie go.³

Sortowanie kluczy — pętle for

Słowniki, będąc odwzorowaniami, obsługują jedynie dostęp do elementów po ich kluczu. Obsługują jednak również operacje specyficzne dla tego typu z wywołaniami metod przydatnymi w wielu często spotykanych sytuacjach.

Jak wspomniano wcześniej, ponieważ słowniki nie są sekwencjami, nie zachowują żadnej kolejności elementów od lewej do prawej strony. Oznacza to, że jeśli utworzymy słownik i wyświetlimy jego zawartość, klucze mogą zostać zwrócone w innej kolejności, niż je wpisaliśmy.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}  
>>> D  
{'a': 1, 'c': 3, 'b': 2}
```

Co jednak można zrobić, kiedy potrzebne nam jest wymuszenie określonej kolejności elementów słownika? Jednym z często spotykanych rozwiązań jest tu pobranie listy kluczy za pomocą metody słownika `keys`, posortowanie tej listy za pomocą metody listy `sort`, a następnie przejęcie wyników za pomocą pętli `for`. Należy pamiętać o dwukrotnym naciśnięciu przycisku `Enter` po utworzeniu poniższego kodu pętli — zgodnie z informacjami z rozdziału 3, pusty wiersz oznacza w sesji interaktywnej „dalej”, natomiast w niektórych interfejsach znak zachęty się zmienia.

```
>>> Ks = list(D.keys())                                     # Nieuporządkowana lista kluczy  
>>> Ks  
['a', 'c', 'b']  
  
>>> Ks.sort()                                              # Posortowana lista kluczy  
>>> Ks  
['a', 'b', 'c']  
  
>>> for key in Ks:  
    print(key, '=>', D[key])                                # Iteracja przez posortowane klucze  
                                                       # <== Tutaj należy dwukrotnie nacisnąć Enter  
  
a => 1  
b => 2  
c => 3
```

Ten proces składa się jednak z trzech etapów, natomiast jak zobaczymy w kolejnych rozdziałach, w nowszych wersjach Pythona można go wykonać w jednym etapie — dzięki nowszej funkcji wbudowanej o nazwie `sorted`. Wywołanie `sorted` zwraca wynik i sortuje różne typy obiektów; w tym przypadku automatycznie sortuje klucze słownika.

```
>>> D  
{'a': 1, 'c': 3, 'b': 2}  
  
>>> for key in sorted(D):
```

³ Jedna uwaga: należy pamiętać, że utworzony przed chwilą spis `rec` mógłby być prawdziwym wpisem do bazy danych, gdybyśmy użyli systemu *trwałości obiektów* (ang. *object persistence*) Pythona — łatwego sposobu przechowywania obiektów Pythona w plikach lub bazach danych dostępnych za pomocą klucza. Nie będziemy teraz omawiać szczegółów tego rozwiązania; po więcej informacji na ten temat należy sięgnąć do omówionych w dalszej części książki modułów `pickle` oraz `shelve`.

```
print(key, '=>', D[key])
```

```
a => 1  
b => 2  
c => 3
```

Poza zaprezentowaniem słowników powyższy przypadek posłużył nam jako pretekst do wprowadzenia pętli `for` dostępnej w Pythonie. Pętla `for` jest prostym i wydajnym sposobem przejścia wszystkich elementów sekwencji i wykonania bloku kodu dla każdego elementu po kolei. Zdefiniowana przez użytkownika zmienna pętli (tutaj: `key`) wykorzystana jest do referencji do aktualnie przechodzonego elementu. Wynikiem tego kodu jest wyświetlenie kluczów i wartości słownika w kolejności posortowanej po kluczach.

Pętla `for`, a także jej bardziej ogólny kuzyn — pętla `while`, są podstawowymi sposobami kodowania powtarzalnych zadań w skryptach jako instrukcji. Tak naprawdę jednak pętla `for`, a także jej krewniak — listy składane, to operacja na sekwencjach. Zadziała na każdym obiekcie będącym sekwencją, podobnie do list składanych — a nawet na pewnych obiektach, które sekwencjami nie są. Poniżej widać przykład przechodzenia znaków w łańcuchu i wyświetlenia wersji każdej z tych znaków napisanego wielką literą.

```
>>> for c in 'mielonka':  
    print(c.upper())
```

```
M  
I  
E  
L  
O  
N  
K  
A
```

Pętla `while` Pythona jest narzędziem bardziej ogólnym i nie jest ograniczona do przechodzenia sekwencji:

```
>>> x = 4  
>>> while x > 0:  
    print('mielonka!' * x)  
    x -= 1  
mielonka!mielonka!mielonka!  
mielonka!mielonka!mielonka!  
mielonka!mielonka!  
mielonka!
```

Instrukcje, składnia i narzędzia powiązane z pętlami omówione zostaną bardziej szczegółowo w dalszej części książki.

Iteracja i optymalizacja

Jeśli pętla `for` z poprzedniego podrozdziału przypomina nam omówione wcześniej wyrażenia z listami składanymi, tak właśnie powinno być — oba są ogólnymi narzędziami iteracyjnymi. Tak naprawdę oba rozwiązania będą działały na każdym obiekcie zgodnym z *protoolem iteracji* — rozpowszechnioną w Pythonie koncepcją, która oznacza fizycznie przechowywaną sekwencję w pamięci czy obiekt generujący jeden element na raz w kontekście operacji iteracji. Obiekt mieści się w tej drugiej kategorii, jeśli odpowiada na wbudowane wywołanie `iter` obiektem, który posuwa się naprzód w odpowiedzi na `next`. Wyrażenie składane *generatora*, które widzieliśmy wcześniej, jest takim właśnie typem obiektu.

Na temat protokołu iteracji będę miał więcej do powiedzenia w dalszej części książki. Na razie należy pamiętać, że każde narzędzie Pythona, które przegląda obiekt od lewej do prawej strony, wykorzystuje protokół iteracji. Dlatego właśnie wywołanie `sorted` wykorzystane wyżej działa bezpośrednio na słowniku. Nie musimy wywoływać metody `keys` w celu otrzymania sekwencji, ponieważ słowniki poddają się iteracji, a `next` zwraca w ich przypadku kolejne klucze.

Oznacza to także, że każde wyrażenie list składanych — jak poniższe, obliczające kwadrat z listy liczb:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]  
>>> squares  
[1, 4, 9, 16, 25]
```

zawsze może zostać zapisane w kodzie jako odpowiednik pętli `for` tworzącej listę ręcznie — poprzez dodanie do niej kolejnych przechodzonych elementów:

```
>>> squares = []  
>>> for x in [1, 2, 3, 4, 5]:  
    squares.append(x ** 2)                      # To robi listę składaną  
                                                # Wewnętrznie wykonują protokół iteracji  
  
>>> squares  
[1, 4, 9, 16, 25]
```

Lista składana, a także powiązane z nią narzędzia programowania funkcjonalnego, takie jak `map` oraz `filter`, zazwyczaj będą obecnie działały szybciej od pętli `for` (być może nawet dwa razy szybciej) — ta cecha może mieć znaczenie w przypadku większych zbiorów danych. Mimo to warto podkreślić, że pomiar wydajności może być w Pythonie dość podchwytliwy, ponieważ język ten tak dużo optymalizuje, a wydajność zmienia się z wydania na wydanie.

Najważniejszą regułą jest w Pythonie kodowanie w sposób prosty oraz czytelny i martwienie się o wydajność dopiero później, kiedy sam program już działa i przekonamy się, że wydajność jest dla nas rzeczywistym problemem. Najczęściej kod będzie wystarczająco szybki. Jeśli musimy w nim trochę pomajstrować pod kątem wydajności, Python zawiera narzędzia, które mogą nam w tym pomóc, w tym moduły `time`, `timeit` i `profile`. Więcej informacji na ten temat znajduje się w dalszej części książki, a także w dokumentacji Pythona.

Brakujące klucze — testowanie za pomocą `if`

Zanim przejdziemy dalej, warto odnotować jeszcze jedną kwestię dotyczącą słowników. Choć możemy przypisać wartość do nowego klucza w celu rozszerzenia słownika, próba pobrania nieistniejącego klucza jest błędem.

```
>>> D  
{'a': 1, 'c': 3, 'b': 2}  
  
>>> D['e'] = 99                         # Przypisanie nowego klucza rozszerza słownik  
>>> D  
{'a': 1, 'c': 3, 'b': 2, 'e': 99}  
  
>>> D['f']                                # Referencja do nieistniejącego klucza jest błędem  
...pominieto tekst błędu...  
KeyError: 'f'
```

Tego właśnie oczekujemy — zazwyczaj pobieranie czegoś, co nie istnieje, jest błędem programistycznym. Jednak w pewnych ogólnych programach nie zawsze będziemy w momencie pisania kodu wiedzieć, jakie klucze będą obecne. W jaki sposób można sobie poradzić w takim przypadku i uniknąć błędów? Jednym z rozwiązań jest sprawdzenie tego zawczasu. Wyrażenie

słownika `in` pozwala na sprawdzenie istnienia klucza i odpowiednie zachowanie w zależności od wyniku tego sprawdzenia — dzięki instrukcji `if`. Tak jak w przypadku `for`, należy pamiętać o dwukrotnym naciśnięciu przycisku *Enter* w celu wykonania instrukcji `if` w sesji interaktywnej.

```
>>> 'f' in D
False

>>> if not 'f' in D:
    print('nie ma')

nie ma
```

Na temat instrukcji `if` i ogólnej składni instrukcji powiemy więcej nieco później, jednak forma wykorzystywana w kodzie powyżej jest dość prosta. Instrukcja składa się ze słowa `if`, po nim następuje wyrażenie, którego wynik interpretowany jest jako prawdziwy lub fałszywy, a następnie blok kodu do wykonania, jeśli wyrażenie będzie prawdziwe. W pełnej formie instrukcja `if` może również zawierać instrukcję `else` z przypadkiem domyślnym, a także jedną lub większą liczbę instrukcji `elif` (od „`else if`”) sprawdzających inne testy. Jest to podstawowe narzędzie wyboru w Pythonie, a także sposób kodowania logiki w skryptach.

Istnieją inne sposoby tworzenia słowników, które pozwalają na uniknięcie prób uzyskania dostępu do nieistniejących kluczy, w tym metoda `get` (warunkowe indeksowanie z wartością domyślną), w Pythonie 2.X metoda `has_key` (w wersji 3.0 już niedostępna), instrukcja `try` (narzędzie, z którym spotkamy się w rozdziale 10. przy okazji przechwytywania wyjątków i radzenia sobie z nimi), a także wyrażenie `if/else` (instrukcja `if` zmieszczona w jednym wierszu). Poniżej znajduje się kilka przykładów:

```
>>> value = D.get('x', 0)                      # Indeks, ale z wartością domyślną
>>> value
0
>>> value = D['x'] if 'x' in D else 0          # Forma wyrażenia if/else
>>> value
0
```

Szczegółowe informacje na temat tych alternatywnych rozwiązań odłożymy jednak do jednego z kolejnych rozdziałów. Teraz czas przejść do omawiania krotek.

Krotki

Obiekt *krotki* (ang. *tuple*) jest w przybliżeniu listą, której nie można zmodyfikować. Krotki są sekwencjami, podobnie do list, jednak są też niezmienne — tak jak łańcuchy znaków. Z punktu widzenia składni kodowane są w zwykłych nawiasach, a nie w nawiasach kwadratowych. Krotki obsługują dowolne typy danych, zagnieżdżanie i zwykłe operacje na sekwencjach.

```
>>> T = (1, 2, 3, 4)                          # Krotka z 4 elementami
>>> len(T)                                     # Długość krotki
4

>> T + (5, 6)                                  # Konkatenacja
(1, 2, 3, 4, 5, 6)

>>> T[0]                                       # Indeksowanie i wycinki
1
```

Krotki mają w Pythonie 3.0 dwie metody wywoływalne specyficzne dla tego typu — nie jest ich zatem tak wiele jak w przypadku list.

```
>>> T.index(4)                                # Metody krotek — 4 znajduje się na wartości przesunięcia 3
3
>>> T.count(4)                                 # 4 pojawia się raz
1
```

Podstawową wyróżniającą cechą krotek jest to, że po utworzeniu nie można ich zmodyfikować. Oznacza to, że są sekwencjami niezmiennymi.

```
>>> T[0] = 2                                  # Krotki są niezmienne
...pominieto tekst błędu...
TypeError: 'tuple' object does not support item assignment
```

Tak jak listy i słowniki, krotki obsługują obiekty o mieszanych typach i zagnieżdżanie, jednak nie rosną i nie kurczą się, ponieważ są niezmienne.

```
>>> T = ('mielonka', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

Czemu służą krotki?

Po co nam zatem typ podobny do listy, który obsługuje mniejszą liczbę operacji? Szczerze mówiąc, w praktyce krotki nie są używane tak często jak listy, jednak ich niezmienność jest ich zaletą. Kiedy w programie przekazujemy zbiór obiektów w postaci listy, obiekty te mogą się w dowolnym momencie zmienić. W przypadku użycia krotki zmienić się nie mogą. Krotki nakładają pewne ograniczenia w zakresie integralności, które przydadzą się w programach większych od pisanych tutaj. O krotkach powiemy jeszcze w dalszej części książki. Najpierw jednak omówmy nasz ostatni główny typ podstawowy, czyli pliki.

Pliki

Obiekty plików są w Pythonie głównym interfejsem do plików zewnętrznych znajdujących się na komputerze. Są typem podstawowym, jednak nieco innym od pozostałych. Nie istnieje żadna składnia literala służąca do ich tworzenia. Zamiast tego w celu utworzenia obiektu wywołuje się wbudowaną funkcję `open`, przekazując nazwę pliku zewnętrznego jako łańcuch znaków wraz z łańcuchem określającym tryb przetwarzania. By na przykład utworzyć plik wyjściowy, należy przekazać jego nazwę wraz z łańcuchem znaków '`w`' określającym tryb przetwarzania umożliwiający zapis danych:

```
>>> f = open('data.txt', 'w')                  # Utworzenie nowego pliku w trybie do zapisu
>>> f.write('Witaj,\n')                         # Zapisanie do niego łańcuchów bajtów
6
>>> f.write('Brian\n')                           # Zwraca liczbę zapisanych bajtów w Pythonie 3.0
6
>>> f.close()                                   # Zamknięcie pliku i wyczyszczenie bufora wyjściowego na dysku
```

Powyższy kod tworzy plik w katalogu bieżącym i zapisuje do niego tekst (nazwa pliku może być pełną ścieżką do katalogu, jeśli potrzebny jest nam dostęp do pliku znajdującego się w innym miejscu komputera). By wczytać z powrotem to, co napisaliśmy, należy ponownie

otworzyć plik, tym razem w trybie przetwarzania 'r', w celu odczytania danych tekstowych (jest to wartość domyślna, jeśli pominiemy tryb w wywołaniu). Następnie należy wczytać zawartość pliku do łańcucha znaków i wyświetlić go. Zawartość pliku jest zawsze dla naszego skryptu łańcuchem znaków, bez względu na typ danych umieszczonych w pliku.

```
>>> f = open('data.txt')                                # 'r' jest domyślnym trybem przetwarzania
>>> text = f.read()                                    # Wczytanie całego pliku do łańcucha znaków
>>> text
'Witaj, \nBrian\n'
>>> print(text)                                       # Instrukcja print interpretuje znaki sterujące
Witaj,
Brian
>>> text.split()                                      # Zawartość pliku jest zawsze łańcuchem znaków
['Witaj,', 'Brian']
```

Inne metody plików obsługują dodatkowe właściwości, których nie mamy teraz czasu omawiać. Obiekty plików udostępniają na przykład większą liczbę sposobów odczytywania i zapisywania (read przyjmuje opcjonalny rozmiar w bajtach, readline wczytuje po jednym wierszu naraz), a także inne narzędzia (seek przechodzi do nowej pozycji pliku). Jak jednak zobaczymy później, najlepszym obecnie sposobem na wczytanie pliku jest *niewczytywanie go* — pliki udostępniają *iterator*, który automatycznie wczytuje wiersz po wierszu w pętlach for oraz innych kontekstach.

Z pełnym zbiorem metod plików spotkamy się w dalszej części książki, a osoby już teraz chcące przyjrzeć się tym metodom mogą wywołać funkcję dir dla dowolnego otwartego pliku oraz funkcję help dla dowolnej ze zwróconych nazw metod:

```
>>> dir(f)
[...pominieto wiele nazw...
'buffer', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
↪'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',
↪'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',
↪'writelines']
```



```
>>> help(f.seek)
...przekonaj się sam!...
```

W dalszej części książki zobaczymy także, że pliki w Pythonie 3.0 zachowują rozróżnienie pomiędzy danymi tekstowymi a binarnymi. *Pliki tekstowe* reprezentują zawartość w postaci łańcuchów znaków i automatycznie wykonują kodowanie Unicode, natomiast *pliki binarne* reprezentują zawartość w postaci specjalnego typu łańcucha bajtowego bytes i pozwalają na dostęp do niezmienionej zawartości. Poniższy fragment kodu zakłada, że w katalogu bieżącym znajduje się już plik binarny.

```
>>> data = open('data.bin', 'rb').read()    # Otwarcie pliku binarnego
>>> data                                     #  Łańcuch bajtowy przechowuje dane binarne
b'\x00\x00\x00\x07mielonka\x00\x08'
>>> data[4:12]
b'spam'
```

Osoby, które będą miały do czynienia jedynie z tekstem w formacie ASCII, nie będą się musiały przejmować tym rozróżnieniem. Łańcuchy znaków i pliki Pythona 3.0 są jednak bardzo cenne, jeśli mamy do czynienia z aplikacjami międzynarodowymi lub danymi bajtowymi.

Inne narzędzia podobne do plików

Funkcja `open` jest koniem pociągowym dla większości czynności związanych z przetwarzaniem plików w Pythonie. Dla bardziej zaawansowanych zadań Python udostępnia jednak dodatkowe narzędzia podobne do plików: potoki, kolejki FIFO, gniazda, pliki dostępne po kluczu, trwałość obiektów, pliki oparte na deskryptorze czy interfejsy do relacyjnych i zorientowanych obiektowo baz danych. Pliki deskryptorów obsługują na przykład blokowanie pliku i inne narzędzia niskiego poziomu, natomiast gniazda udostępniają interfejs do zadań sieciowych i komunikacji międzyprocesowej. Niewiele z tych zagadnień poruszymy w niniejszej książce, jednak osobom, które zajmą się programowaniem w Pythonie na poważnie, z pewnością wiele z nich się przyda.

Inne typy podstawowe

Poza omówionymi dotychczas typami podstawowymi istnieją inne, które mogą się zaliczać do tej kategorii — w zależności od tego, jak szeroko ją zdefiniujemy. *Zbiory* są na przykład nowym dodatkiem do języka, który nie jest ani odwzorowaniem, ani sekwencją. Zbiory to raczej nieuporządkowane kolekcje unikalnych i niezmiennych obiektów. Tworzy się je, wywołując wbudowaną funkcję `set` lub za pomocą nowych literałów i wyrażeń zbiorów z Pythona 3.0. Obsługują one zwykłe operacje matematyczne na zbiorach. Wybór nowej składni `{...}` dla literałów zbiorów w wersji 3.0 ma sens, ponieważ zbiory przypominają klucze słownika bez wartości.

```
>>> X = set('mielonka')                                # Utworzenie zbioru z sekwencji w wersjach 2.6 oraz 3.0
>>> Y = {'s', 'z', 'y', 'n', 'k', 'a'}                 # Utworzenie zbioru za pomocą nowego literalu z wersji 3.0
>>> X, Y
({'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n'}, {'a', 'k', 'n', 's', 'y', 'z'})
>>> X & Y                                         # Część wspólna zbiorów
{'a', 'k', 'n'}
>>> X | Y                                         # Suma zbiorów
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n', 's', 'y', 'z'}
>>> X - Y                                         # Różnica
{'i', 'm', 'e', 'l', 'o'}
>>> {x ** 2 for x in [1, 2, 3, 4]}               # Zbiory składane z wersji 3.0
{16, 1, 4, 9}
```

Dodatkowo Python od niedawna urósł o kilka nowych typów liczbowych: liczby *dziesiętne* (liczby zmiennoprzecinkowe o stałej precyzyj) oraz liczby *ulamkowe* (liczby wymierne zawierające licznik i mianownik). Oba typy można wykorzystać do obejścia ograniczeń i niedokładności będących nierozerwalną częścią arytmetyki liczb zmiennoprzecinkowych.

```
>>> 1 / 3                                         # Liczby zmiennoprzecinkowe (w Pythonie 2.6 należy użyć .0)
0.3333333333333333
>>> (2/3) + (1/2)
1.1666666666666665

>>> import decimal                               # Liczby dziesiętne — stała precyza
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')
>>> decimal.getcontext().prec = 2
```

```

>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')

>>> from fractions import Fraction          # Ułamki — licznik i mianownik
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)

```

Python zawiera także wartości typu *Boolean* (ze zdefiniowanymi obiektami `True` i `False`, które są tak naprawdę liczbami całkowitymi 1 i 0 z własną logiką wyświetlania). Od dawna obsługuje również specjalny obiekt pojemnika o nazwie `None`, wykorzystywany najczęściej do inicjalizowania zmiennych i obiektów.

```

>>> 1 > 2, 1 < 2                      # Wartości typu Boolean
(False, True)

>>> bool('mielonka')
True

>>> X = None                          # Pojemnik None
>>> print(X)
None
>>> L = [None] * 100                  # Inicjalizacja listy stu None
>>> L
[None, None, None,
 ↵...lista stu None...]

```

Jak zepsuć elastyczność kodu

Więcej o tych typach obiektów Pythona powiemy w dalszej części książki, jednak jeden z nich zasługuje na kilka słów już teraz. Obiekt *typu*, zwracany przez funkcję wbudowaną `type`, to obiekt dający typ innego obiektu. Jego wynik różni się nieco w Pythonie 3.0 z uwagi na całkowite zlanie się typów i klas (coś, co omówimy w szóstej części książki w kontekście klas w nowym stylu). Zakładając, że `L` nadal jest listą z poprzedniego podrozdziału:

```

# W Pythonie 2.6:
>>> type(L)                           # Typy: L jest obiektem typu lista
<type 'list'>
>>> type(type(L))                   # Typy też są obiektami
<type 'type'>

# W Pythonie 3.0:
>>> type(L)                           # 3.0: typy są klasami i odwrotnie
<class 'list'>
>>> type(type(L))                   # Więcej na temat typów klas w rozdziale 31.
<class 'type'>

```

Poza umożliwieniem interaktywnego zbadania obiektów w praktyce pozwala to kodowi sprawdzać typy przetwarzanych obiektów. W skryptach Pythona można to zrobić na przynajmniej trzy sposoby.

```

>>> if type(L) == type([]):           # Sprawdzanie typów, skoro już musimy...
    print('tak')

tak
>>> if type(L) == list:              # Użycie nazwy typu

```

```

print('tak')

tak
>>> if isinstance(L, list):           # Sprawdzanie zorientowane obiektowo
    print('tak')

tak

```

Pokazałem wszystkie te sposoby sprawdzania typów, jednak moim obowiązkiem jest dodać, że korzystanie z nich jest prawie zawsze złym pomysłem w programie napisanym w Pythonie (i często jest też znakiem rozpoznawczym byłego programisty języka C, który zaczyna przygodę z Pythonem). Przyczyna takiego stanu rzeczy stanie się całkowicie zrozumiała dopiero później, kiedy zaczniemy pisać większe jednostki kodu, takie jak funkcje, jednak jest to konsepcja kluczowa dla Pythona (być może wręcz najważniejsza ze wszystkich). Sprawdzając określone typy w kodzie, efektywnie niszczymy jego elastyczność, ograniczając go do tylko jednego typu danych. Bez takiego sprawdzania kod może działać na szerokiej gamie typów danych.

Jest to powiązane ze wspomnianą wcześniej koncepcją polimorfizmu i ma swoje źródło w braku deklaracji typu w Pythonie. W Pythonie, czego nauczymy się już niedługo, koduje się pod kątem *interfejsów* obiektów (obsługiwanych operacji), a nie typów. Nieprzejmowanie się konkretnymi typami sprawia, że kod automatycznie można zastosować do wielu typów danych. Każdy obiekt ze zgodnym interfejsem będzie działał bez względu na typ. Choć sprawdzanie typów jest obsługiwane — a nawet w niektórych przypadkach wymagane — to nie jest to „pythonowy” sposób myślenia. Jak się okaże, sam polimorfizm jest jedną z kluczowych koncepcji stojących za Pythonem.

Klasy zdefiniowane przez użytkownika

W dalszej części książki będziemy bardziej szczegółowo omawiali *programowanie zorientowane obiekto*wie w Pythonie — opcjonalną, lecz mającą duże możliwości właściwość tego języka pozwalającą na skrócenie czasu programowania dzięki dostosowaniu obiektów do własnych potrzeb. Z abstrakcyjnego punktu widzenia klasy definiują nowe typy obiektów, które rozszerzają zbiór typów podstawowych, dlatego zasługują na kilka słów w tym miejscu. Powiedzmy na przykład, że potrzebny jest nam typ obiektu będący modelem pracownika. Choć taki właśnie typ nie istnieje w Pythonie, poniższa zdefiniowana przez użytkownika klasa powinna się przydać.

```

>>> class Worker:
    def __init__(self, name, pay):
        self.name = name               # Inicjalizacja przy utworzeniu
        self.pay = pay                  # self jest nowym obiektem; name to nazwisko, a pay — płaca
    def lastName(self):
        return self.name.split()[-1]   # Podział łańcucha znaków na znakach pustych
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)    # Uaktualnienie płacy w miejscu

```

Powyższa klasa definiuje nowy rodzaj obiektu, który będzie miał atrybuty `name` (nazwisko) i `pay` (płaca) — czasami nazywane *informacjami o stanie* (ang. *state information*) — a także dwa rodzaje zachowania zakodowane w postaci funkcji (normalnie nazywanych *metodami*). Wywołanie klasy w sposób podobny do funkcji generuje obiekty naszego nowego typu, a metody klasy automatycznie otrzymują obiekt przetwarzany przez dane wywołanie metody (w argumentencie `self`).

```

>>> bob = Worker('Robert Zielony', 50000)          # Utworzenie dwóch obiektów
>>> anna = Worker('Anna Czerwona', 60000)        # Każdy ma atrybut imienia i nazwiska oraz płacy
>>> bob.lastName()                                # Wywołanie metody — self to bob
'Zielony'
>>> anna.lastName()                                # Teraz self to anna
'Czerwona'
>>> anna.giveRaise(.10)                            # Uaktualnienie płacy Anny
>>> anna.pay
66000.0

```

Domniemany obiekt `self` jest przyczyną nazwania tego modelu zorientowanym obiektowo — w funkcjach klasy zawsze istnieje domniemany podmiot. W pewnym sensie typ oparty na klasie po prostu tworzy coś na bazie typów podstawowych. Zdefiniowany wyżej obiekt `Worker` jest na przykład zbiorem łańcucha znaków oraz liczby (odpowiednio `name` i `pay`) wraz z funkcjami odpowiedzialnymi za przetwarzanie tych wbudowanych obiektów.

Bardziej rozbudowane wyjaśnienie kwestii klas mówi, że to ich mechanizm dziedziczenia obsługuje hierarchię oprogramowania, która pozwala na dostosowywanie programów do własnych potrzeb poprzez rozszerzanie ich. Programy rozszerza się poprzez pisanie nowych klas, a nie modyfikowanie tego, co już działa. Warto również wiedzieć, że klasy są w Pythonie opcjonalne i w wielu przypadkach proste obiekty wbudowane, takie jak listy i słowniki, są często lepszymi narzędziami od klas zakodowanych przez użytkowników. Wszystko to jednak wykracza poza zakres wstępnego omówienia typów, dlatego należy uznać to tylko za zapowiedź tego, co znajduje się dalej. Pełne informacje na temat typów zdefiniowanych przez użytkownika i tworzonych za pomocą klas będzie można znaleźć w szóstej części książki.

I wszystko inne

Jak wspomniano wcześniej, wszystko, co możemy przetworzyć w skrypcie Pythona, jest typem obiektu, przez co nasze omówienie nie może być kompletne. Jednak mimo że wszystko w Pythonie jest obiektem, jedynie obiekty omówione wyżej stanowią zbiór obiektów podstawowych tego języka. Pozostałe typy są albo obiektami powiązanymi z omówionym później wykonywaniem programu (jak funkcje, moduły, klasy i skompilowany kod), albo są implementowane przez funkcje importowanych modułów, a nie składnię języka. Ta druga kategoria pełni zwykle role specyficzne dla określonego zastosowania — wzorców tekstowych, interfejsów do baz danych czy połączeń sieciowych.

Należy również pamiętać, że obiekty omówione wyżej są *obiektami*, ale niekoniecznie są *zorientowanymi obiektami*. Zorientowanie obiektowe zazwyczaj wymaga w Pythonie dziedziczenia i instrukcji `class`, z którą spotkamy się w dalszej części książki. Obiekty podstawowe Pythona są silną napędową prawie każdego skryptu napisanego w tym języku, z jakim można się spotkać, i zazwyczaj są też podstawą większych typów niemieszczących się w tej kategorii.

Podsumowanie rozdziału

I to by było na tyle, jeśli chodzi o naszą krótką wycieczkę po typach danych. W niniejszym rozdziale zauważałem zwięzle wprowadzenie do podstawowych typów obiektów w Pythonie wraz z omówieniem rodzajów operacji, jakie można do nich zastosować. Przyjrzelismy się uniwersalnym operjom, które działają na wielu typach obiektów (na przykład operjom na sekwencjach, takim jak indeksowanie czy wycinki), a także operjom specyficznym dla

określonego typu, dostępnym w postaci wywołania metody (na przykład dzielenie łańcuchów znaków czy dodawanie elementów do listy). Zdefiniowaliśmy również pewne kluczowe pojęcia, takie jak niezmienność, sekwencje i polimorfizm.

Po drodze widzieliśmy również, że podstawowe typy obiektów Pythona są bardziej elastyczne i mają większe możliwości od tego, co dostępne jest w językach niższego poziomu, takich jak C. Listy i słowniki w Pythonie usuwają na przykład większość pracy koniecznej do obsługiwanego kolekcji oraz wyszukiwania w językach niższego poziomu. Listy to uporządkowane kolekcje innych obiektów, natomiast słowniki to kolekcje innych obiektów indeksowane po kluczu, a nie pozycji. Zarówno słowniki, jak i listy mogą być zagnieżdżane, mogą się rozszerzać i kurczyć na życzenie i mogą zawierać obiekty dowolnego typu. Co więcej, po porzuceniu ich automatycznie odzyskiwane jest miejsce zajmowane przez nie w pamięci.

Pominąłem większość szczegółów w celu skrócenia naszej wycieczki, dlatego nie należy oczekiwać, że całość tego rozdziału będzie miała sens. W kolejnych rozdziałach zacznijemy kopać coraz głębiej i głębiej, odkrywając szczegółowe podstawowe typy obiektów Pythona, które tutaj pominęliśmy w celu lepszego zrozumienia całości. Zaczniemy już w następnym rozdziale od pogłębionego omówienia liczb w Pythonie. Najpierw jednak — kolejny quiz.

Sprawdź swoją wiedzę — quiz

Koncepcje przedstawione w niniejszym rozdziale omówimy bardziej szczegółowo w kolejnych rozdziałach, dlatego teraz czas na szerszą perspektywę.

1. Należy podać cztery podstawowe typy danych Pythona.
2. Dlaczego typy te nazywane są podstawowymi?
3. Co oznacza pojęcie „niezmienny” i które trzy z podstawowych typów danych Pythona są uznawane za niezmienne?
4. Czym jest sekwencja i które trzy typy danych należą do tej kategorii?
5. Czym jest odwzorowanie i który typ podstawowy zalicza się do tej kategorii?
6. Czym jest polimorfizm i dlaczego powinno to być dla nas ważne?

Sprawdź swoją wiedzę — odpowiedzi

1. Liczby, łańcuchy znaków, listy, słowniki, krotki, pliki i zbiory są uważane za podstawowe typy danych w Pythonie. Czasami w ten sam sposób klasyfikowane są również typy, None i wartości typu Boolean. Istnieje kilka typów liczbowych (liczby całkowite, zmiennoprzecinkowe, zespolone, ułamkowe i dziesiętne), a także różne typy łańcuchów znaków (zwykłe łańcuchy znaków, łańcuchy znaków Unicode z Pythona 2.X, łańcuchy tekstowe oraz bajtowe z Pythona 3.X).
2. Typy te nazywane są podstawowymi, ponieważ są częścią samego języka i są zawsze dostępne. By natomiast utworzyć inne obiekty, konieczne jest wywołanie funkcji z importowanych modułów. Większość typów podstawowych ma określoną składnię służącą do ich generowania — na przykład ‘mielonka’ to wyrażenie tworzące łańcuch znaków i ustalające

zbiór operacji, które mogą być do niego zastosowane. Z tego powodu typy podstawowe są ściśle połączone ze składnią Pythona. Żeby natomiast utworzyć obiekt pliku, trzeba wywołać wbudowaną funkcję open.

3. Obiekt niezmienny to taki obiekt, który nie może być zmodyfikowany po utworzeniu. W Pythonie do tej kategorii zaliczamy liczby, łańcuchy znaków i krotki. Choć nie można zmodyfikować niezmiennego obiektu w miejscu, zawsze można utworzyć nowy obiekt, wykonując odpowiednie wyrażenie.
4. Sekwencja to uporządkowana pod względem pozycji kolekcja obiektów. W Pythonie sekwencjami są łańcuchy znaków, listy oraz krotki. Dzielą one wspólne operacje na sekwencjach, takie jak indeksowanie, konkatenacja czy wycinki, jednak każdy ma również specyficzne dla danego typu metody.
5. Pojęcie „odwzorowanie” oznacza obiekt, który odwzorowuje klucze na powiązane wartości. Jedynym takim typem w zbiorze podstawowych typów obiektów Pythona jest słownik. Odwzorowania nie zachowują żadnej stałej kolejności elementów od lewej do prawej strony. Obsługują dostęp do danych po kluczu oraz metody specyficzne dla danego typu.
6. Polimorfizm oznacza, że znaczenie operacji (takiej jak +) uzależnione jest od obiektów, na których się ona odbywa. Okazuje się to kluczową koncepcją stanowiącą podstawę Pythona (być może nawet tą najważniejszą) — nieograniczenie kodu do określonych typów sprawia, że kod ten można automatycznie zastosować do wielu typów.

Typy liczbowe

Niniejszy rozdział rozpoczyna nasze pogłębione omówienie języka Python. W Pythonie dane przybierają formę *obiektów* — albo wbudowanych obiektów udostępnianych przez język, albo obiektów tworzonych za pomocą narzędzi Pythona i innych języków, takich jak C. Obiekty są tak naprawdę podstawą każdego programu, jaki pisze się w Pythonie. Ponieważ są podstawowym budulcem tego języka, będą również stanowiły nasz pierwszy obiekt zainteresowania.

W poprzednim rozdziale krótko przedstawiliśmy podstawowe typy obiektów Pythona. Choć zostały w nim wprowadzone najważniejsze pojęcia, nie zagłębialiśmy się w szczegóły. Teraz zaczniemy przyglądać się bardziej dokładnie różnym koncepcjom związanym ze strukturami danych, tak by wypełnić pominięte wcześniej detale. Zaczniemy zatem od naszej pierwszej kategorii typów danych Pythona — typów liczbowych.

Podstawy typów liczbowych Pythona

Większość typów liczbowych Pythona jest dość typowa i dla osób używających kiedyś jakiegokolwiek języka programowania powinny one wyglądać znajomo. Mogą być wykorzystywane do przechowywania stanu konta, odległości z Ziemi do Marsa, liczby odwiedzających naszą witrynę internetową i każdej innej wartości liczbowej.

W Pythonie liczby nie są tak naprawdę jednym typem obiektu; są raczej kategorią podobnych typów. Python obsługuje zwykłe typy liczbowe (liczby całkowite oraz zmiennoprzecinkowe), a także literaly służące do tworzenia liczb i wyrażenia je przetwarzające. Dodatkowo Python udostępnia bardziej zaawansowaną obsługę programowania numerycznego, a także obiekty przeznaczone do bardziej zaawansowanych działań. Pełen zasób typów liczbowych Pythona obejmuje:

- liczby całkowite i zmiennoprzecinkowe,
- liczby zespolone,
- liczby dziesiętne o ustalonej precyzji,
- ułamkowe liczby wymierne,
- zbiory,
- wartości Boolean,

- liczby całkowite o nieograniczonej precyzyji,
- różne wbudowane wartości liczbowe oraz moduły.

Niniejszy rozdział rozpoczniemy od omówienia prostych typów liczbowych i podstaw, a następnie przejdziemy do kolejnych elementów z powyższej listy. Zanim jednak zabierzemy się za pisanie kodu, kilka kolejnych podrozdziałów zajmie nam krótkie omówienie sposobu zapisywania i przetwarzania liczb w naszych skryptach.

Literały liczbowe

Wśród typów podstawowych Python udostępnia *liczby całkowite* (dodatnie i ujemne) oraz *zmiennoprzecinkowe* (z częścią ułamkową). Python pozwala na zapis liczb całkowitych za pomocą literałów szesnastkowych, ósemkowych oraz dwójkowych, oferuje dodatkowo typ liczby zespolonej i pozwala, by liczby całkowite miały nieograniczoną precyzyję (moga one urosnąć do takiej liczby cyfr, na jaką pozwoli miejsce w pamięci). W tabeli 5.1 można znaleźć przykłady różnych typów liczbowych zapisane w sposób, w jaki mogą się one pokazać w programie w postaci literałów.

Tabela 5.1. Podstawowe literały liczbowe

Literał	Interpretacja
1234, -24, 0, 9999999999999999	Liczby całkowite (ang. <i>integer</i> ; o nieograniczonej wielkości)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Liczby zmiennoprzecinkowe (ang. <i>floating-point number</i>)
0177, 0x9ff, 0b101010	Literały ósemkowe, szesnastkowe i dwójkowe w wersji 2.6
0o177, 0x9ff, 0b101010	Literały ósemkowe, szesnastkowe i dwójkowe w wersji 3.0
3+4j, 3.0+4.0j, 3J	Literały liczb zespolonych (ang. <i>complex number</i>)

Generalnie typy liczbowe Pythona są dość proste pod względem zapisu, choć warto podkreślić kilka kwestii związanych z ich kodowaniem.

Literały liczb całkowitych oraz zmiennoprzecinkowych

Liczby całkowite zapisywane są jakłańcuchy cyfr dziesiętnych. Liczby zmiennoprzecinkowe mają znak dziesiętny (w postaci kropki) lub zawierają opcjonalny wykładnik ze znakiem wprowadzony po literze e lub E, po której znajduje się opcjonalny znak. Jeśli liczbę zapiszemy ze znakiem dziesiętnym lub wykładnikiem, Python automatycznie robi z niej obiekt liczby zmiennoprzecinkowej i kiedy liczba ta użyta zostanie w wyrażeniu, wykorzysta arytmetykę liczb zmiennoprzecinkowych, a nie całkowitych. Liczby zmiennoprzecinkowe zaimplementowane są jako typ *double* z języka C, dlatego mają taką precyzyję, jaką kompilator języka C wykorzystany do zbudowania interpretera Pythona przypisza liczbom tego typu.

Liczby całkowite w Pythonie 2.6 — zwykłe i długie

W wersji 2.6 dostępne są dwa typy liczb całkowitych — zwykłe (32-bitowe) oraz długie (o nieograniczonej precyzyji). Liczba całkowita może kończyć się znakami l lub L, przez co wymuszane jest potraktowanie jej jako długiej liczby całkowitej. Ze względu na to, że liczby całkowite są automatycznie konwertowane na długie liczby całkowite, kiedy ich wartość przekroczy trzydzieści dwa bity, nie trzeba wpisywać litery L samodzielnie. Kiedy potrzebna jest dodatkowa precyzyja, Python automatycznie wykonuje konwersję na długą liczbę całkowitą.

Liczby całkowite w Pythonie 3.0 — jeden typ

W wersji 3.0 dwa typy liczb całkowitych — zwykłe oraz długie — zostały połączone. Istnieje tylko jeden typ liczby całkowitej, który automatycznie obsługuje nieograniczoną precyzję odrębnego typu długiej liczby całkowitej Pythona 2.6. Z tego powodu liczby całkowite nie mogą już być zapisywane z końcową literą l lub L, a liczby tego typu nigdy nie są również wyświetlane z tymi znakami. Poza tym ograniczeniem zmiana ta nie wpłynie na większość programów, o ile nie wykonują one sprawdzania typów pod kątem długich liczb całkowitych z wersji 2.6.

Literały szesnastkowe, ósemkowe oraz dwójkowe

Liczby całkowite mogą być w Pythonie zapisane jako dziesiętne (o podstawie dziesięć), szesnastkowe (o podstawie szesnaście), ósemkowe (o podstawie osiem) oraz dwójkowe (o podstawie dwa). Literały szesnastkowe zaczynają się od znaków 0x lub 0X, po których następuje ciąg cyfr szesnastkowych od 0 do 9 i od A do F. W literałach szesnastkowych cyfry szesnastkowe mogą być zapisane małą i wielką literą. Literały ósemkowe rozpoczynają się od znaków 0o lub 0O (cyfry zero i małej lub wielkiej litery „o”), po których następuje ciąg cyfr od 0 do 7. W Pythonie 2.6 i wcześniejszych wersjach literały ósemkowe mogły także być zapisywane z samą początkową cyfrą 0, jednak od wersji 3.0 jest to niemożliwe (początkowy format liczby ósemkowej można łatwo pomylić z liczbą dziesiętną, dla tego zastąpiono go nowym formatem 0o). Literały dwójkowe, nowość w wersjach 2.6 i 3.0, rozpoczynają się od znaków 0b lub 0B, po których następują cyfry dwójkowe (0 lub 1).

Warto zauważyć, że wszystkie powyższe literały tworzą w kodzie programu obiekty liczb całkowitych, które są tylko alternatywnymi sposobami zapisu określonych wartości. Wywołania wbudowanych funkcji hex(liczba_całkowita), oct(liczba_całkowita) oraz bin(liczba_całkowita) przekształcają liczbę całkowitą na jej reprezentację o podanej podstawie, natomiast wywołanie int(str, podstawa) przekształca wykonywany lańcuch na liczbę całkowitą zgodnie z podaną podstawą.

Liczby zespolone

Literały liczb zespolonych Pythona zapisywane są jako część_rzeczywista+część_urojona, gdzie część_urojona kończy się znakiem j lub J. Fragment część_rzeczywista jest opcjonalny, dlatego część_urojona może występować samodzielnie. Wewnętrznie liczby zespolone zaimplementowane są jako pary liczb zmiennoprzecinkowych, jednak wszystkie operacje liczbowe na liczbach zespolonych wykorzystują arytmetykę liczb zespolonych. Liczby zespolone można również tworzyć za pomocą wywołania wbudowanej funkcji complex(część_rzeczywista, część_urojona).

Inne typy liczbowe

Jak zobaczymy w dalszej części niniejszego rozdziału, istnieją inne, bardziej zaawansowane typy liczbowe, które nie zostały uwzględnione w tabeli 5.1. Część z nich tworzona jest za pomocą wywoływania funkcji z importowanych modułów (na przykład liczby dziesiętne oraz ułamki), natomiast inne posiadają własną składnię literałów (na przykład zbiory).

Wbudowane narzędzia liczbowe

Oprócz zaprezentowanych w tabeli 5.1 literałów liczbowych wbudowanych w sam język Python udostępnia również zbiór narzędzi służących do przetwarzania obiektów liczbowych:

Operatory wyrażeń

`+, -, *, /, >>, **, &` i inne

Wbudowane funkcje matematyczne

`pow, abs, round, int, hex, bin` i inne

Moduły z narzędziami

`random, math` i inne

Spotkamy się z każdym z tych narzędzi w miarę omawiania liczb.

Choć liczby przetwarzają się przede wszystkim za pomocą wyrażeń, wbudowanych funkcji oraz modułów, udostępniają one również obecnie pewną liczbę *metod* specyficznych dla poszczególnych typów, które także zostaną omówione w niniejszym rozdziale. Liczby zmiennoprzecinkowe mają na przykład metodę `as_integer_ratio` przydającą się dla typu liczby ułamkowej, a także metodę `is_integer` sprawdzającą, czy liczba ta jest całkowita. Liczby całkowite dysponują różnymi atrybutami, w tym nową metodą `bit_length` z wersji 3.1 Pythona, która podaje liczbę bitów niezbędną do odzwierciedlenia wartości obiektu. Co więcej, *zbiory*, będące po części typem kolekcji, a po części typem liczbowym, także obsługują zarówno metody, jak i wyrażenia.

Ponieważ to wyrażenia są najważniejszym narzędziem w przypadku większości typów liczbowych, to od nich rozpoczęmy nasze omówienie.

Operatory wyrażeń Pythona

Chyba najważniejszym narzędziem przetwarzającym liczby jest *wyrażenie* (ang. *expression*) — kombinacja liczb (i innych obiektów) oraz operatorów obliczająca, po wykonaniu w Pythonie, wartość. W Pythonie wyrażenia zapisywane są z wykorzystaniem zwykłej notacji matematycznej oraz symboli operatorów. Żeby na przykład dodać do siebie dwie liczby, X i Y , należy stworzyć wyrażenie $X + Y$, które nakazuje Pythonowi zastosować operator `+` do wartości o nazwach X i Y . Wynikiem tego wyrażenia jest suma liczb X oraz Y , czyli inny obiekt liczbowy.

W tabeli 5.2 wymieniono wszystkie wyrażenia z operatorami dostępne w Pythonie. Wiele z nich jest oczywistych; obsługiwane są na przykład zwykłe operatory matematyczne (jak `+`, `-`, `*` czy `/`). Kilka będzie wyglądało znajomo dla osób, które w przeszłości używały innych języków programowania — i tak `%` oblicza resztę z dzielenia, `<<` wykonuje przesunięcie bitowe w lewo, a `&` oblicza wynik dla bitowego AND. Inne są specyficzne dla Pythona i nie wszystkie są z natury numeryczne. Operator `is` sprawdza na przykład tożsamość obiektu (to znaczy adres w pamięci, scisłą formę równości), a `lambda` tworzą funkcje anonimowe.

Ponieważ w niniejszej książce omawiamy Pythona w wersjach 2.6 oraz 3.0, poniżej znajduje się kilka uwag dotyczących różnic między tymi wersjami, a także nowych dodatków odnoszących się do operatorów wymienionych w tabeli 5.2.

- W Pythonie 2.6 nierówność wartości można zapisać albo jako $X \neq Y$, albo $X <> Y$. W Pythonie 3.0 druga z tych opcji została usunięta, ponieważ jest zbędna. W obu wersjach zalecane jest używanie w testach nierówności zapisu $X \neq Y$.
- W Pythonie 2.6 wyrażenie z apostrofem odwrotnym `X` działa tak samo jak `repr(X)` i przekształca obiekty na wyświetlanełańcuchy znaków. Ze względu na niejasność wyrażenie to zostało w Pythonie 3.0 usunięte. Teraz należy korzystać z o wiele bardziej czytelnych funkcji `str` oraz `repr`, opisanych w podrozdziale „Formaty wyświetlania liczb” w niniejszym rozdziale.

Tabela 5.2. Operatory wyrażeń Pythona wraz z priorytetem

Operatory	Opis
<code>yield x</code>	Protokół send funkcji generatora
<code>lambda argumenty: wyrażenie</code>	Generowanie funkcji anonimowej
<code>x if y else z</code>	Trójargumentowe wyrażenie wyboru (<code>x</code> jest obliczane jedynie wtedy, gdy <code>y</code> jest prawdziwe)
<code>x or y</code>	Logiczne OR (<code>y</code> jest obliczane tylko wtedy, gdy <code>x</code> jest fałszywe)
<code>x and y</code>	Logiczne AND (<code>y</code> jest obliczane tylko wtedy, gdy <code>x</code> jest prawdziwe)
<code>not x</code>	Logiczna negacja
<code>x in y, x not in y</code>	Przynależność (obiekty, po których można iterować, zbiorы)
<code>x is y, x is not y</code>	Testy identyczności obiektów
<code>x < y, x <= y, x > y, x >= y</code>	Operatory porównania, podzbiory i nadzbiory zbiorów;
<code>x == y, x != y</code>	Operatory równości wartości
<code>x y</code>	Bitowe OR, suma zbiorów
<code>x ^ y</code>	Bitowe XOR, symetryczna różnica zbiorów
<code>x & y</code>	Bitowe AND, część wspólna zbiorów
<code>x << y, x >> y</code>	Przesunięcie <code>x</code> w lewo lub prawo o <code>y</code> bitów
<code>x + y</code>	Dodawanie, konkatenacja;
<code>x - y</code>	Odejmowanie, różnica zbiorów
<code>x * y</code>	Mnożenie, powtórzenie;
<code>x % y</code>	Reszta z dzielenia, format;
<code>x / y, x // y</code>	Dzielenie: prawdziwe i bez reszty
<code>-x, +x</code>	Negacja, identyczność
<code>~x</code>	Bitowe NOT
<code>x ** y</code>	Potęga
<code>x[i]</code>	Indeksowanie (sekwencje, odwzorowania, inne)
<code>x[i:j:k]</code>	Wycinki
<code>x(...)</code>	Wywołanie (funkcji, metod, klas, innych obiektów, które można wywoływać)
<code>x.atrybut,</code>	Odwołanie do atrybutu
<code>(...)</code>	Krotka, wyrażenie, wyrażenie generatora
<code>[...]</code>	Lista, lista składana
<code>{...}</code>	Słownik, zbiór, a także zbiór i słownik składany

- Operator dzielenia bez reszty (ang. *floor division*) `X // Y` zawsze odcina ułamkową resztę z dzielenia — zarówno w Pythonie 2.6, jak i 3.0. Wyrażenie `X / Y` wykonuje w wersji 3.0 prawdziwe dzielenie (zachowując resztę), natomiast w wersji 2.6 — dzielenie klasyczne (odcinanie do liczby całkowitej). Więcej informacji na ten temat znajduje się w podrozdziale „Dzielenie — klasyczne, bez reszty i prawdziwe”.
- Składnia `[...]` może reprezentować albo literal listy, albo wyrażenie list składanych. Ta druga opcja wykonuje pętlę i zbiera wynik wyrażenia do nowej listy. Przykłady takiego działania można znaleźć w rozdziałach 4., 14. oraz 20.

- Składnia (...) wykorzystywana jest w krotkach oraz wyrażeniach, a także w wyrażeniach generatorów — formie list składanych zwracającej wyniki na żądanie zamiast budowania listy wyników. Przykłady znajdują się w rozdziałach 4. oraz 20. We wszystkich trzech konstrukcjach można czasami pominąć nawiasy.
- Składnia {...} wykorzystywana jest w literałach słowników, natomiast w Pythonie 3.0 w literałach zbiorów i słownikach oraz zbiorach składanych. Przykłady znajdują się w omówieniu zbiorów w niniejszym rozdziale oraz w rozdziałach 4., 8., 14. i 20.
- Wyrażenia yield oraz trójargumentowe wyrażenie wyboru if/else dostępne są w Pythonie w wersji 2.5 i nowszych. Pierwsze z nich zwraca argumenty send(...) w generatorach. Drugie jest skrótem dla wielowierszowej instrukcji if. Wyrażenie yield wymaga użycia nawiasów, jeśli nie znajduje się po prawej stronie instrukcji przypisania jako jedyny element.
- Operatory porównania można ze sobą łączyć. Kod X < Y < Z zwraca ten sam wynik co X < Y i Y < Z. Więcej informacji na ten temat znajduje się w podrozdziale „Porównania — zwykłe i łączone” w niniejszym rozdziale.
- W nowszych wersjach Pythona wyrażenie wycinka X[I:J:K] jest równoważne z indeksowaniem z obiektem wycinka X[slice(I, J, K)].
- W wersjach 2.X dozwolone jest porównywanie wielkości obiektów o mieszanych typach — przekształcanie liczb na wspólny typ i układanie innych typów mieszanych zgodnie z nazwą typu. W Pythonie 3.0 porównywanie wielkości obiektów nieliczbowych o mieszanych typach nie jest dozwolone i powoduje zgłoszanie wyjątków. Obejmuje to także sortowanie za pomocą metody pośredniczącej.
- W Pythonie 3.0 nie są już również obsługiwane porównania wielkości słowników (choć testy równości są). Jedynym dostępnym rozwiązaniem pozostaje porównanie sorted ↴(dict.items()).

Większość operatorów z tabeli 5.2 zobaczymy w akcji nieco później. Najpierw musimy jednak przyjrzeć się sposobom łączenia tych operatorów w wyrażeniach.

Połączone operatory stosują się do priorytetów

Tak jak w większości języków programowania, również w Pythonie bardziej skomplikowane wyrażenia koduje się, łącząc ze sobą wyrażenia z operatorami z tabeli 5.2. Sumę dwóch mnożeń można na przykład zapisać jak poniższe połączenie zmiennych i operatorów.

A * B + C * D

Skąd zatem Python wie, jakie działania ma wykonać jako pierwsze? Odpowiedź na to pytanie leży w *priorytecie operatorów* (ang. *operator precedence*). Kiedy piszemy wyrażenie zawierające więcej niż jeden operator, Python grupuje jego części zgodnie z tak zwanymi *regułami precedencji*. To grupowanie ustala kolejność obliczania poszczególnych części wyrażenia. Tabela 5.2 została posortowana zgodnie z priorytetem operatorów:

- Operatory znajdujące się niżej w tabeli mają wyższy priorytet, dlatego wiążą bardziej ściśle w wyrażeniach mieszanych.
- Operatory znajdujące się w tym samym wierszu tabeli 5.2 zazwyczaj po połączeniu grupują od lewej do prawej strony (z wyjątkiem potęgowania, które grupuje od prawej do lewej strony, i porównań, łączących w łańcuchy od lewej do prawej strony).

Jeśli na przykład zapiszemy wyrażenie $X + Y * Z$, Python najpierw wykona mnożenie ($Y * Z$), którego wynik doda następnie do X , ponieważ $*$ ma wyższy priorytet (jest niżej w tabeli) od $+$. W podobny sposób w pierwszym przykładzie najpierw wykonane zostaną oba działania mnożenia ($A * B$ oraz $C * D$), a dopiero później ich wyniki zostaną do siebie dodane.

Podwyrażenia grupowane są w nawiasach

O priorytecie operatorów można zapomnieć, jeśli będzie się uważnie grupowało części wyrażeń w nawiasy. Kiedy podwyrażenia umieści się w nawiasach, będą one ważniejsze od reguł priorytetów Pythona. Python zawsze najpierw wykonuje wyrażenia znajdujące się w nawiasach, a dopiero później ich wyniki wykorzystuje w zawierających je wyrażeniach.

Zamiast na przykład pisać $X + Y * Z$, można zapisać to wyrażenie na dwa sposoby, zmuszając jednocześnie Pythona do obliczenia go w pożąданej kolejności.

```
(X + Y) * Z  
X + (Y * Z)
```

W pierwszym przypadku najpierw dodawane są do siebie elementy X i Y , ponieważ to wyrażenie umieszczone jest w nawiasach. W drugim przypadku najpierw wykonywane jest mnożenie (dokładnie tak samo by było, gdyby nawiasy zostały pominięte). Dodawanie nawiasów w większych wyrażeniach jest dobrym pomysłem. Nie tylko wymusza to odpowiednią kolejność wykonywania działań, ale dodatkowo poprawia również czytelność.

Pomieszane typy poddawane są konwersji

Poza mieszaniem operatorów w wyrażeniach można również łączyć ze sobą typy liczbowe. Można na przykład dodać liczbę całkowitą do liczby zmiennoprzecinkowej.

```
40 + 3.14
```

Prowadzi to jednak do kolejnego pytania: jakiego typu będzie wynik takiego działania — będzie liczbą całkowitą czy zmiennoprzecinkową? Odpowiedź jest prosta, szczególnie dla osób, które używały już kiedyś jakiegoś języka programowania. W wyrażeniach z mieszanymi typami obiektów liczbowych Python najpierw dokonuje konwersji argumentów *w górę*, do typu, jakim jest najbardziej skomplikowany argument, a dopiero później wykonuje obliczenia dla argumentów tego samego typu. Osoby korzystające kiedyś z języka C pamiętają zapewne, że podobne konwersje typów zastosowano także w tym języku.

Python mierzy stopień skomplikowania typów liczbowych w następującej kolejności: liczby całkowite są prostsze od liczb zmiennoprzecinkowych, które są prostsze od liczb zespolonych. Kiedy zatem liczba całkowita zostanie pomieszana ze zmiennoprzecinkową, jak w przykładzie wyżej, zostanie ona najpierw przekonwertowana na wartość zmiennoprzecinkową i w tym samym formacie zwrócony zostanie wynik działania. W podobny sposób dowolne wyrażenie z różnymi typami liczbowymi, z których jeden będzie liczbą zespoloną, zwróci wynik będący liczbą zespoloną. W Pythonie 2.6 zwykłe liczby całkowite przekształcane są także na długie liczby całkowite w każdej sytuacji, kiedy ich wartość jest zbyt duża, by zmieścić się w zwykłej liczbie całkowitej. W wersji 3.0 liczby całkowite wchodząły długie liczby całkowite.

Takie zachowanie można również wymusić, ręcznie wywołując wbudowane funkcje służące do konwersji typów.

```
>>> int(3.1415)                                # Odcina liczbę zmiennoprzecinkową do całkowitej  
3  
>>> float(3)                                 # Przekształca liczbę całkowitą na zmiennoprzecinkową  
3.0
```

Zazwyczaj jednak nie będzie trzeba tego robić. Ponieważ Python automatycznie konwertuje typy w górę do najbardziej skomplikowanego typu z danego wyrażenia, wynik zazwyczaj będzie w odpowiedniej postaci.

Należy również pamiętać, że wszystkie konwersje mieszanych typów mają miejsce tylko przy łączeniu typów *liczbowych* (na przykład liczby całkowitej i zmiennoprzecinkowej) w wyrażeniach, w tym wyrażeniach wykorzystujących operatory liczbowe i porównania. Generalnie Python nie dokonuje automatycznej konwersji w przypadku typów innego rodzaju. Przykładowo dodanie łańcucha znaków do liczby całkowitej spowoduje wyświetlenie błędu, o ile ręcznie nie przekształcimy jednego z nich. Przykład takiego działania będzie można znaleźć przy okazji omawiania łańcuchów znaków w rozdziale 7.



W Pythonie 2.6 mieszane typy nieliczbowe można porównywać, jednak nie są wykonywane żadne przekształcenia (typy mieszane porównywane są zgodnie ze stałą, dość przypadkową regułą). W wersji 3.0 porównania mieszanych typów nieliczbowych nie są dozwolone i powodują zgłoszenie wyjątków.

Przeciążanie operatorów — zapowiedź

Choć obecnie skupiamy się na wbudowanych typach liczbowych, należy pamiętać, że w Pythonie wszystkie operatory mogą być przeciążane (to znaczy implementowane) przez klasy Pythona oraz typy z rozszerzeń języka C, tak by działały na tworzonych obiektach. Później zobaczymy na przykład, że obiekty utworzone za pomocą klas można dodawać i poddawać konkatenacji za pomocą wyrażeń z operatorem + czy indeksować za pomocą [i].

Co więcej, Python sam automatycznie przeciąża niektóre operatory, tak by wykonywały one różne zadania w zależności od typu przetwarzanego obiektu wbudowanego. Operator + wykonuje na przykład operację dodawania, kiedy zastosuje się go na liczbach, ale w połączeniu z łańcuchami znaków oraz listami odpowiedzialny jest za konkatenację. Operator + może oznaczać dowolne działania po zastosowaniu do obiektów zdefiniowanych za pomocą klas.

Jak widzieliśmy w poprzednim rozdziale, właściwość ta nazywana jest najczęściej *polimorfizmem*. Termin ten oznacza, że znaczenie operacji uzależnione jest od typu obiektów, na których się tę operację wykonuje. Koncepcję tą zajmiemy się ponownie przy okazji omawiania funkcji w rozdziale 16., ponieważ w tym kontekście stanie się ona o wiele bardziej oczywista.

Liczby w akcji

Czas przejść do kodu! Chyba najlepszym sposobem zrozumienia obiektów liczbowych i związanych z nimi wyrażeń jest zobaczenie ich w akcji. Należy zatem uruchomić interaktywny wiersz poleceń i przetestować kilka podstawowych operacji (wskaźówki dotyczące uruchamiania sesji interaktywnej znaleźć można w rozdziale 3.).

Zmienne i podstawowe wyrażenia

Zacznijmy od podstawowej matematyki. W poniższej sesji interaktywnej najpierw przypiszemy dwie *zmienne* (a oraz b) do liczb całkowitych, tak by móc z nich później skorzystać w większym wyrażeniu. Zmienne to po prostu nazwy utworzone przez Pythona, które wykorzystywane są do śledzenia informacji w programie. Więcej na ten temat powiemy w kolejnym rozdziale, jednak na razie warto podkreślić, że w Pythonie:

- Zmienne tworzone są, kiedy pierwszy raz przypisze się do nich wartości.
- Zmienne zastępowane są wartościami, kiedy wykorzystywane są w wyrażeniach.
- Zmienne muszą być przypisane przed użyciem w wyrażeniach.
- Zmienne odnoszą się do obiektów i nigdy nie są wcześniej deklarowane.

Innymi słowy, przypisania te sprawią, że zmienne a i b zaistnieją automatycznie.

```
% python
>>> a = 3
>>> b = 4
# Utworzono zmenną (nazwę)
```

W kodzie powyżej wykorzystano również *komentarz*. Warto przypomnieć, że w kodzie napisanym w Pythonie tekst od znaku # aż do końca wiersza jest uznawany za komentarz i tym samym ignorowany. Komentarze to metoda zapisu w kodzie czytelnej dla człowieka dokumentacji. Ponieważ kod wpisywany interaktywnie jest tymczasowy, zazwyczaj w tym kontekście nie używa się komentarzy, jednak ja dodalem je w niektórych przykładach z książki w celu wyjaśnienia kodu.¹ W dalszej części książki spotkamy się z powiązaną właściwością — łańcuchami znaków dokumentacji, które dodają tekstu komentarz do obiektów.

Teraz wykorzystamy nasze nowe obiekty liczb całkowitych w wyrażeniach. W tym momencie wartości a i b będą równe 3 i 4. Zmienne tego typu są zastępowane wartościami za każdym razem, gdy znajdują się w wyrażeniu. W czasie pracy w sesji interaktywnej wyniki wyrażenia są od razu zwracane.

```
>>> a + 1, a - 1
(4, 2)
# Dodawanie (3 + 1), odejmowanie (3 - 1)

>>> b * 3, b / 2
(12, 2)
# Mnożenie (4 * 3), dzielenie (4 / 2)

>>> a % 2, b ** 2
(1, 16)
# Modulo (reszta z dzielenia), potęga (4 ** 2)

>>> 2 + 4.0, 2.0 ** b
(6.0, 16.0)
# Konwersja typów mieszanych
```

Z technicznego punktu widzenia zwarcane wyniki są *krotkami* składającymi się z dwóch wartości, ponieważ wiersze wpisywane w sesji interaktywnej zawierają po dwa wyrażenia rozdzielone przecinkiem — dlatego właśnie wyniki wyświetlane są w nawiasach (więcej o krotkach w kolejnych rozdziałach). Warto zauważyć, że wyrażenia działają, ponieważ znajdujące się w nich zmienne a i b mają przypisane wartości. Gdybyśmy użyły innej zmiennej, do której nigdy nie przypisano wartości, Python zwróciłby błąd i nie wstawiłby do zmiennej jakiejś wartości domyślnej.

¹ Osoby pracujące z kodem nie muszą wpisywać komentarzy od znaków # do końca wiersza do sesji interaktywnej. Komentarze są przez Pythona ignorowane i nie są wymaganą częścią wykonywanych instrukcji.

```
>>> c * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'c' is not defined
```

Zmiennych nie trzeba w Pythonie wcześniej deklarować, jednak przed wykorzystaniem trzeba im przynajmniej raz przypisać jakąś wartość. W praktyce oznacza to, że liczniki trzeba inicjalizować z wartością zero, zanim będzie można coś do nich dodawać; w podobny sposób przed dodaniem czegoś do listy najpierw trzeba zainicjalizować pustą listę.

Poniżej znajdują się dwa nieco bardziej rozbudowane wyrażenia ilustrujące grupowanie operatorów i konwersje typów.

```
>>> b / 2 + a                      # To samo co ((4 / 2) + 3)
5.0
>>> print(b / (2.0 + a))          # To samo co (4 / (2.0 + 3))
0.8
```

W pierwszym wyrażeniu nie ma nawiasów, dlatego Python automatycznie grupuje argumenty zgodnie z regułami priorytetów. Ponieważ `/` jest w tabeli 5.2 niżej od `+`, ma wyższy priorytet i działanie to obliczane jest jako pierwsze. Z tego powodu całość obliczana jest w taki sam sposób, jakby wyrażenie zawierało nawiasy takie, jak w komentarzu.

Warto również zauważyc, że wszystkie liczby z pierwszego wyrażenia są liczbami całkowitymi. Z tego powodu Python 2.6 wykonuje operacje dzielenia i dodawania liczb całkowitych i zwróci wynik 5, natomiast Python 3.0 wykonuje prawdziwe dzielenie z resztą i zwraca wynik pokazany powyżej. Jeśli chcemy w wersji 3.0 uzyskać dzielenie liczb całkowitych, należy zapisać powyższe działanie w postaci `b // 2 + a` (więcej informacji na temat dzielenia wkrótce).

W drugim wyrażeniu nawiasy dodawane są wokół części z operatorem `+` w celu zmuszenia Pythona do obliczenia jej na początku (przed `/`). Jeden z argumentów staje się również liczbą zmiennoprzecinkową ze względu na dodanie do wyrażenia liczby 2.0. Ponieważ wyrażenie zawiera mieszane typy, Python konwertuje liczbę całkowitą kryjącą się pod zmienną `a` na wartość zmiennoprzecinkową (3.0), jeszcze zanim wykona dodawanie. Gdyby wszystkie liczby w tym wyrażeniu były liczbami całkowitymi, w rezultacie dzielenia liczb całkowitych (4 / 5) w Pythonie 2.6 wynik zostałby odcięty do 0, natomiast w Pythonie 3.0 wynik będzie liczbą zmiennoprzecinkową 0.8 (więcej informacji o dzieleniu wkrótce).

Formaty wyświetlania liczb

Warto zauważyc, że w ostatnim przykładzie wykorzystaliśmy operację `print`. Bez niej zobaczylibyśmy coś, co na pierwszy rzut oka może wyglądać dziwnie.

```
>>> b / (2.0 + a)                  # Dane zwracane automatycznie — więcej cyfr
0.8000000000000004
>>> print(b / (2.0 + a))          # Operacja print zaokrąglą liczbę
0.8
```

Pełne wyjaśnienie tego dziwnego wyniku wiąże się z ograniczeniami procesorów do obliczeń zmiennoprzecinkowych i ich niemożnością dokładnej reprezentacji pewnych wartości w ograniczonej liczbie bitów. Ponieważ jednak architektura komputerów znacznie wykracza poza tematykę niniejszej książki, wystarczy powiedzieć, że wszystkie cyfry z pierwszego przykładu naprawdę tam są — w urządzeniach zmiennoprzecinkowych komputera, tylko my sami nie

jestesmy przyzwyczajeni do ich oglądania. Przykład ten tak naprawdę jest kwestią wyświetlanego — automatyczne wyświetlanie wyników w sesji interaktywnej pokazuje więcej cyfr niż `print`. Jeśli nie chcemy widzieć tych wszystkich cyfr, musimy skorzystać z `print`. Zgodnie z informacjami z ramki „Formaty wyświetlania str i repr” z niniejszego rozdziału tak można uzyskać sposób wyświetlania przyjazny dla użytkownika.

Warto jednak zauważyc, że nie wszystkie wartości zawierają tyle cyfr do wyświetlenia.

```
>>> 1 / 2.0  
0.5
```

Istnieje więcej sposobów wyświetlania bitów liczby wewnętrz komputera, nie tylko korzystanie z `print` i wyświetlanie automatyczne z sesji interaktywnej:

```
>>> num = 1 / 3.0  
>>> num  
0.3333333333333333 # Wyświetlenie wyniku  
>>> print(num)  
0.333333333333 # Zaokrąglenie liczb za pomocą print  
  
>>> '%e' % num  
'3.333333e-001'  
>>> '%.2f' % num  
'0.33'  
>>> '{0:.4.2f}'.format(num)  
'0.33'  
# Formatowanie za pomocą łańcuchów znaków  
# Alternatywne formatowanie liczb zmiennoprzecinkowych  
# Metoda formatowania za pomocą łańcuchów znaków (Python 2.6 i 3.0)
```

Ostatnie trzy wyrażenia wykorzystują *formatowanie za pomocą łańcucha znaków* — narzędzie pozwalające na elastyczność formatu, które omówimy w rozdziale 7. poświęconym łańcuchom znaków. Jego wynikiem są łańcuchy znaków, które są zazwyczaj wyświetlane lub przekazywane do raportów.

Formaty wyświetlania str i repr

Z technicznego punktu widzenia różnica między automatycznym wyświetlaniem z sesji interaktywnej a `print` odpowiada różnicy między wbudowanymi funkcjami `str` i `repr`.

```
>>> num = 1 / 3  
>>> repr(num)  
'0.3333333333333331'  
>>> str(num)  
'0.333333333333'  
# Wyświetlanie w postaci jak w kodzie (sesja interaktywna)  
# Wyświetlanie w formie przyjaznej dla użytkownika (print)
```

Obie funkcje konwertują dowolne obiekty na ich reprezentacje łańcuchowe. Funkcja `repr` (i domyślne wyświetlanie z sesji interaktywnej) zwraca wyniki wyglądające tak, jakby były one kodem. Funkcja `str` (oraz operacja `print`) dokonuje konwersji na format bardziej przyjazny dla użytkownika, jeśli jest on dostępny. Niektóre obiekty mają obie postacie — `str` do użytku ogólnego i `repr` z dodatkowymi szczegółami. Koncepcje te pojawią się ponownie przy okazji omawiania łańcuchów znaków oraz przeciążania operatorów w klasach, a więcej informacji na temat tych funkcji wbudowanych znajduje się w dalszej części książki.

Oprócz udostępniania wyświetlanych łańcuchów znaków dla dowolnych obiektów wbudowana funkcja `str` jest także nazwą typu danych łańcucha znaków i można ją wywołać z nazwą kodowania w celu zdekodowania łańcucha znaków Unicode z łańcucha bajtowego. Tę ostatnią, bardziej zaawansowaną rolę omówimy w rozdziale 36. książki.

Porównania — zwykłe i łączone

Dotychczas zajmowaliśmy się standardowymi działaniami na liczbach (dodawaniem czy mnożeniem), jednak liczby można także porównywać. Zwykłe porównania działają w przypadku liczb tak, jak można by się tego spodziewać — porównują względną wielkość argumentów i zwracają wynik w postaci wartości Boolean (który w większej instrukcji normalnie byłby sprawdzany):

```
>>> 1 < 2                                # Mniejszy od
True
>>> 2.0 >= 1                             # Większy od bądź równy — mieszane typy, więc 1 przekształcone zostaje na 1.0
True
>>> 2.0 == 2.0                            # Równa wartość
True
>>> 2.0 != 2.0                            # Nie mają równej wartości
False
```

Warto raz jeszcze zwrócić uwagę na to, że w wyrażeniach liczbowych (i tylko tam) można używać typów mieszanych. W drugim teście Python porównuje wartości jako obiekty typu o większej złożoności, czyli liczby zmiennoprzecinkowe.

Co ciekawe, Python pozwala także łączyć ze sobą większą liczbę porównań w celu wykonywania testów przedziałów. Połączone porównania są w pewnym sensie skrótem dla większych wyrażeń Boolean. W skrócie, Python pozwala połączyć ze sobą kilka testów porównania wielkości w celu utworzenia kodu porównań łączonych, takich jak testy przedziałów. Wyrażenie ($A < B < C$) sprawdza na przykład, czy B znajduje się pomiędzy A i C , i jest równoznaczne z testem Boolean ($A < B$ and $B < C$), jednak przyjemniejsze dla oka (i klawiatury). Założymy, że mamy poniższe instrukcje przypisania:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

Dwa zaprezentowane niżej wyrażenia dają identyczny efekt, jednak pierwsze z nich jest krótsze i może działać nieco szybciej, ponieważ Python musi obliczyć Y tylko raz:

```
>>> X < Y < Z                                # Połączone porównania — testy przedziału
True
>>> X < Y and Y < Z
True
```

Tak samo równoznaczne są testy zwracające `False`. Dozwolone są również łańcuchy porównań o dowolnej długości:

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False

>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

W połączonych testach można także skorzystać z innych porównań, jednak niektóre wyrażenia zapisane w ten sposób mogą być nieintuicyjne, o ile nie obliczamy ich w ten sam sposób co Python. Poniższe wyrażenie zwraca na przykład `False` tylko dlatego, że 1 nie jest równe 2 :

```
>>> 1 == 2 < 3  
False  
# To samo co: 1 == 2 i 2 < 3  
# Nie to samo co: False < 3 (co oznacza 0 < 3 i jest prawdą)
```

Python nie porównuje wyniku `False` z `1 == 2` do `3` — to akurat oznaczałoby to samo co `0 < 3`, czyli powinno zwracać `True` (jak zobaczymy w dalszej części niniejszego rozdziału, `True` i `False` są po prostu zapisanymi w innym formacie liczbami `1` oraz `0`).

Dzielenie — klasyczne, bez reszty i prawdziwe

Widzieliśmy już w poprzednich podrozdziałach, jak działa dzielenie, zatem powinniśmy również wiedzieć, że działanie to zachowuje się nieco inaczej w wersjach 3.0 i 2.6. Tak naprawdę istnieją jednak trzy typy dzielenia i dwa różne operatory dzielenia (z czego jeden zmienił się w Pythonie 3.0).

`X / Y`

Dzielenie *klasyczne* i *prawdziwe*. W wersjach Pythona 2.6 i wcześniejszych operator ten wykonuje dzielenie *klasyczne*, odcinając resztę dla liczb całkowitych i utrzymując resztę dla liczb zmiennoprzecinkowych. W Pythonie 3.0 operator ten wykonuje dzielenie *prawdziwe*, które zawsze zachowuje resztę, bez względu na typ obiektu.

`X // Y`

Dzielenie *bez reszty*. Dodane w Pythonie 2.2 i dostępne w wersjach 2.6 oraz 3.0. Operator ten zawsze odcina ułamkową resztę, bez względu na typ obiektu.

Prawdziwe dzielenie zostało dodane w celu zniwelowania tego, że wyniki dzielenia klasycznego uzależnione są od typu argumentu, przez co mogą być trudne do przewidzenia w języku z typami dynamicznymi, jakim jest Python. Dzielenie klasyczne zostało usunięte w wersji 3.0 z uwagi na to właśnie ograniczenie — operatory `/` oraz `//` implementują w tej wersji Pythona dzielenie prawdziwe oraz dzielenie bez reszty.

Podsumowując:

- W wersji 3.0 operator `/` zawsze wykonuje dzielenie *prawdziwe*, zwracając wynik będący liczbą zmiennoprzecinkową uwzględniającą resztę, bez względu na typ argumentów. Operator `//` wykonuje dzielenie *bez reszty*, odcinające resztę, i zwraca liczbę całkowitą dla argumentów będących liczbami całkowitymi lub liczbą zmiennoprzecinkową, jeśli któryś z argumentów jest liczbą tego typu.
- W wersji 2.6 operator `/` wykonuje dzielenie *klasyczne*, odcinając resztę w przypadku, gdy oba argumenty są liczbami całkowitymi, i zachowując resztę w pozostałych przypadkach. Operator `//` wykonuje dzielenie *bez reszty* i działa tak samo jak w wersji 3.0, odcinając resztę w przypadku liczb całkowitych i wykonując dzielenie bez reszty w przypadku liczb zmiennoprzecinkowych.

Oto jak działają oba operatory w wersjach 3.0 oraz 2.6:

```
C:\misc> C:\Python30\python  
>>>  
>>> 10 / 4  
2.5  
# Inaczej niż w 3.0 — zachowuje resztę  
>>> 10 // 4  
2  
# Tak jak w 3.0 — odcina resztę  
>>> 10 / 4.0  
2.5  
# Tak jak w 3.0 — zachowuje resztę
```

```

>>> 10 // 4.0                                # Tak jak w 3.0 — odcina resztę
2.0

C:\misc> C:\Python26\python
>>>
>>> 10 / 4
2
>>> 10 // 4
2
>>> 10 / 4.0
2.5
>>> 10 // 4.0
2.0

```

Warto zauważyc, że w wersji 3.0 typ danych wyniku dla operatora `//` w dalszym ciągu uzależniony jest od typu argumentów. Jeśli któryś z nich jest liczbą zmiennoprzecinkową, wynikiem jest liczba zmiennoprzecinkowa. W odwrotnej sytuacji wynik jest liczbą całkowitą. Choć może się to wydawać podobne do zachowania uzależnionego od typu dla operatora `/` w wersjach 2.X, który był motywacją dla zmian z wersji 3.0, typ zwracanej wartości jest o wiele mniej istotny od samej zwracanej wartości. Co więcej, ponieważ operator `//` został udostępniony po części jako narzędzie zapewniające zgodność ze starszymi wersjami w przypadku programów, które opierają się na dzieleniu liczb całkowitych z odcinaniem reszty (co jest o wiele częściej spotykane, niż można by się spodziewać), dla liczb całkowitych musi zwracać liczby całkowite.

Obsługa różnych wersji Pythona

Choć sposób działania operatora `/` w wersjach 2.6 i 3.0 różni się od siebie, nadal można w kodzie obsługiwać obie wersje Pythona. Jeśli programy opierają się na dzieleniu liczb całkowitych z odcinaniem reszty, zarówno w 2.6, jak i 3.0 należy użyć operatora `//`. Jeśli programy wymagają wyników będących liczbami zmiennoprzecinkowymi z resztą dla liczb całkowitych, należy użyć `float` w celu zagwarantowania, że po wykonaniu programu w wersji 2.6 jeden z argumentów wokół operatora `/` będzie liczbą zmiennoprzecinkową:

```

X = Y // Z                                # Zawsze odcina resztę, wynik zawsze jest liczbą całkowitą dla liczb całkowitych w 2.6 i 3.0
X = Y / float(Z)                          # Gwarantuje dzielenie liczb zmiennoprzecinkowych z resztą w 2.6 i 3.0

```

Alternatywnie można włączyć dzielenie z operatorem `/` z wersji 3.0 w Pythonie 2.6 za pomocą instrukcji `__future__ import division`, zamiast wymuszać je za pomocą konwersji z użyciem `float`:

```

C:\misc> C:\Python26\python
>>> from __future__ import division          # Włączenie obsługi "/" z wersji 3.0
>>> 10 / 4
2.5
>>> 10 // 4
2

```

Dzielenie bez reszty a odcinanie

Drobny szczegół: operator `//` znany jest najczęściej jako dzielenie z *odcinaniem*, jednak bardziej poprawne byłoby nazywanie go dzieleniem *bez reszty* — odcina on resztę, pozostawiając podstawkę będącą najbliższą liczbą całkowitą poniżej prawdziwego wyniku. W rezultacie zaokrąglamy wynik w dół, a nie odcinamy, co ma znaczenie w przypadku liczb ujemnych. Różnicę tę można zobaczyć na własne oczy, używając modułu Pythona o nazwie `math` (przed użyciem zawartości modułu trzeba go najpierw zimportować, o czym będziemy mówić nieco później):

```
>>> import math
>>> math.floor(2.5)
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5)
2
>>> math.trunc(-2.5)
-2
```

Przy użyciu operatorów dzielenia tak naprawdę odcinamy resztę jedynie dla dodatnich wyników, ponieważ w takim przypadku odcięcie da nam samą podstawę. W przypadku liczb ujemnych uzyskujemy w rzeczywistości dzielenie bez reszty (tak naprawdę oba działania są dzieleniem bez reszty, a w przypadku liczb dodatnich dzielenie bez reszty jest równoznaczne z odcinaniem). Oto przykłady z wersji 3.0:

```
C:\misc> c:\python30\python
>>> 5 / 2, 5 / 2
(2.5, 2.5)

>>> 5 // 2, 5 // 2          # Odcina resztę do podstawy — zaokrąglą do pierwszej mniejszej liczby całkowitej
(2, 3)                      # 2.5 staje się 2, -2.5 staje się -3

>>> 5 / 2.0, 5 / 2.0
(2.5, 2.5)

>>> 5 // 2.0, 5 // 2.0      # Tak samo w przypadku liczb zmiennoprzecinkowych, choć tu wynik jest także
                            # liczbą zmiennoprzecinkową
(2.0, 3.0)
```

W przypadku wersji 2.6 jest podobnie, jednak wyniki dla operatora / znów będą różne:

```
C:\misc> c:\python26\python
>>> 5 / 2, 5 / 2          # Inaczej niż w 3.0
(2, 3)

>>> 5 // 2, 5 // 2         # Tu i poniżej wyniki w 2.6 i 3.0 są takie same
(2, 3)

>>> 5 / 2.0, 5 / 2.0
(2.5, 2.5)

>>> 5 // 2.0, 5 // 2.0
(2.0, 3.0)
```

Jeśli naprawdę potrzebne jest nam odcinanie, bez względu na znak, w każdej wersji Pythona możemy zawsze przekazać wynik dzielenia liczb zmiennoprzecinkowych do funkcji `math.trunc` (powiązaną funkcjonalność można znaleźć we wbudowanej funkcji `round`):

```
C:\misc> c:\python30\python
>>> import math
>>> 5 / 2                  # Zachowanie reszty
2.5
>>> 5 // 2                 # Zaokrąglenie wyniku w dół
-3
>>> math.trunc(5 / 2)       # Odcięcie reszty zamiast zaokrąglenia w dół
2

C:\misc> c:\python26\python
>>> import math
>>> 5 / float(2)           # Reszta w wersji 2.6
2.5
```

```
>>> 5 / 2, 5 // 2 # Zaokrąglenie w dół w wersji 2.6
( 3, 3)
>>> math.trunc(5 / float( 2))
2 # Odcięcie reszty w wersji 2.6
```

Dlaczego odcinanie ma znaczenie?

Jeśli korzystamy z Pythona 3.0, oto krótkie podsumowanie sposobu działania operatorów dzielenia:

```
>>> (5 / 2), (5 / 2.0), (5 / 2.0), (5 / 2)           # Prawdziwe dzielenie w 3.0
(2.5, 2.5, 2.5, 2.5)

>>> (5 // 2), (5 // 2.0), (5 // 2.0), (5 // 2)       # Dzielenie bez reszty w 3.0
(2, 2.0, 3.0, 3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)        # Oba
(3.0, 3.0, 3, 3.0)
```

U osób korzystających z wersji 2.6 dzielenie działa następująco:

```
>>> (5 / 2), (5 / 2.0), (5 / 2.0), (5 / 2)           # Dzielenie klasyczne w 2.6
(2, 2.5, 2.5, 3)

>>> (5 // 2), (5 // 2.0), (5 // 2.0), (5 // 2)       # Dzielenie bez reszty w 2.6 (to samo)
(2, 2.0, 3.0, 3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)        # Oba
(3, 3.0, 3, 3.0)
```

Choć o wynikach dopiero się przekonamy, możliwe, że działanie operatora / bez odcinania reszty w Pythonie 3.0 spowoduje problemy w funkcjonowaniu sporej liczby programów. Być może ze względu na korzenie w składni języka C wielu programistów polega na dzieleniu z odcinaniem reszty dla liczb całkowitych. Będą oni musieli się nauczyć stosować w tym kontekście operator //. Przykład kodu, na jaki powyższa zmiana operatora / może mieć wpływ, znajduje się w zastosowaniu pętli while na liczbach pierwszych z rozdziału 13. oraz odpowiadającym mu ćwiczeniu umieszczonym na końcu części czwartej książki. Dodatkowo więcej informacji na temat przedstawionej wyżej specjalnej formy instrukcji import będzie można znaleźć w rozdziale 24.

Precyza liczb całkowitych

Operacja dzielenia może działać różnie w poszczególnych wersjach Pythona, jednak jest elementem dość standardowym. A teraz coś bardziej egzotycznego. Jak wspomniano wcześniej, Python 3.0 obsługuje liczby całkowite o nieograniczonej wielkości:

Python 2.6 ma osobny typ dla długich liczb całkowitych, jednak wszystkie liczby, które są zbyt duże, by można je było przechować w zwykłą liczbę całkowitej, przekształca na ten typ automatycznie. Tym samym by korzystać z długich liczb całkowitych, nie trzeba używać żadnej specjalnej składni — jedyną oznaką tego, że w Pythonie 2.6 korzystamy z długiej liczby całkowitej, jest wyświetlenie jej ze znakiem `L` na końcu:

Liczby całkowite o nieograniczonej precyzyji są bardzo przydatnym narzędziem wbudowanym. Można ich na przykład użyć przy obliczeniu deficytu narodowego w groszach bezpośrednio w Pythonie (oczywiście jeśli ktoś ma na to ochotę, a także wystarczającą ilość pamięci w komputerze). Umożliwiły one również podniesienie liczby 2 do tak wysokiej potęgi w jednym z przykładów z rozdziału 3. Oto przykłady działania z wersji 2.6 oraz 3.0 Pythona:

```
>>> 2 ** 200  
1606938044258990275541962092341162602522202993782792835301376
```

```
>>> 2 ** 200  
1606938044258990275541962092341162602522202993782792835301376L
```

Ponieważ Python musi wykonać dodatkową pracę, by obsługiwać rozszerzoną precyzyję takich liczb, arytmetyka liczb całkowitych jest znaczco wolniejsza, kiedy liczby robią się coraz większe. Jeśli jednak potrzebujemy dodatkowej precyzyji, to, że taka możliwość jest od razu wbudowana w język, z pewnością przeważy nad argumentem o jej niższej wydajności.

Liczby zespolone

Choć wykorzystywane są nieco rzadziej niż typy omawiane dotychczas, liczby zespolone są w Pythonie osobnym podstawowym typem obiektów. Osoby zaznajomione z nimi wiedzą również, dlaczego liczby te są przydatne. Osoby niemające tej wiedzy mogą potraktować ten fragment tekstu jako opcjonalny.

Liczby zespolone przedstawiane są jako dwie liczby zmiennoprzecinkowe — część rzeczywista i część urojona. Są one kodowane poprzez dodanie liter j lub J do części urojonej. Liczby zespolone z niezerową częścią rzeczywistą można również zapisać, dodając dwie części za pomocą znaku +. Poniżej znajdują się przykłady arytmetyki liczb zespolonych.

```
>>> 1j * 1j  
(-1+0j)  
>>> 2 + 1j * 3  
(2+3j)  
>>> (2 + 1j) * 3  
(6+3j)
```

Można również dokonać ekstrakcji części liczb zespolonych jako atrybutów. Liczby te obsługują wszystkie zwykłe wyrażenia matematyczne i można je przetwarzać za pomocą narzędzi ze standardowego modułu `cmath` (wersji standardowego modułu `math` przeznaczonej dla liczb zespolonych). Liczby zespolone zazwyczaj sprawdzają się w programach zorientowanych na inżynierów. Ponieważ są narzędziami zaawansowanymi, więcej informacji na ten temat należy szukać w dokumentacji Pythona.

Notacja szesnastkowa, ósemkowa i dwójkowa

Jak wspomniano na początku rozdziału, liczby całkowite można w Pythonie zapisać w notacji szesnastkowej, ósemkowej i dwójkowej, poza zwykłym zapisem w notacji dziesiętnej. Reguły zapisu tych liczb zostały przedstawione na początku rozdziału. Teraz zajmijmy się przykładami z życia.

Należy pamiętać, że te literaly są tylko alternatywną składnią służącą do określania wartości obiektu liczby całkowitej. Poniższe literaly zapisane w Pythonie 3.0 oraz 2.6 oznaczają zwykłe liczby całkowite z określonymi wartościami dla wszystkich trzech podstaw:

```
>>> 0o1, 0o20, 0o377           # Literały ósemkowe
(1, 16, 255)
>>> 0x01, 0x10, 0xFF          # Literały szesnastkowe
(1, 16, 255)
>>> 0b1, 0b100000, 0b11111111 # Literały dwójkowe
(1, 16, 255)
```

Powyzsza wartość ósemkowa 0o377, szesnastkowa 0xFF i dwójkowa 0b11111111 to dziesiętne 255. Python domyślnie wyświetla liczby w formie dziesiętnej, jednak udostępnia wbudowane funkcje pozwalające na konwersję liczb całkowitych na ich odpowiedniki zapisane jakołańcuchy o innych podstawach.

```
>>> oct(64), hex(64), bin(64)
('0o100', '0x40', '0b1000000')
```

Funkcja `oct` konwertuje liczbę dziesiętną na ósemkową, funkcja `hex` — na szesnastkową, natomiast `bin` — na dwójkową. Wbudowana funkcja `int` pozwala natomiast przekształcićłańcuch cyfr na liczbę całkowitą, a opcjonalny drugi argument określa podstawę.

```
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)

>>> int('0x40', 16), int('0b1000000', 2)      # Literały też są w porządku
(64, 64)
```

Funkcja `eval`, z którą spotkamy się w dalszej części książki, traktujełańcuchy znaków tak, jakby były one kodem Pythona. Z tego powodu ma podobny efekt, choć zazwyczaj działa wolniej, ponieważ kompiluje oraz wykonujełańcuch jako część programu i zakłada, że możemy ufaćźródłu wykonywanegołańcucha — sprytny użytkownik byłby w stanie wysłaćłańcuch znaków kasujący wszystkie pliki z komputera!

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

Można również przekształcić liczby całkowite nałańcuchy szesnastkowe i ósemkowe za pomocą metody `z formatującymłańcuchem znaków`.

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64)
'100, 40, 1000000'

>>> '%o, %x, %X' % (64, 255, 255)
'100, ff, FF'
```

Ten sposób formatowania omówiony zostanie bardziej szczegółowo w rozdziale 7.

Dwa ostrzeżenia przed przejściem dalej. Po pierwsze, użytkownicy Pythona 2.6 powinni pamiętać, że liczby ósemkowe można zapisywać za pomocą umieszczenia zera na początku, w oryginalnym formacie ósemkowym Pythona:

```
>>> 0o1, 0o20, 0o377           # Nowy format ósemkowy w 2.6 (ten sam co w 3.0)
(1, 16, 255)

>>> 01, 020, 0377             # Stare literały ósemkowe w 2.6 (i wcześniejszych wersjach)
(1, 16, 255)
```

W wersji 3.0 składnia z drugiego przykładu generuje błąd. Choć w wersji 2.6 nie jest to błąd, należy uważać, by nie rozpoczynaćłańcucha cyfr zerem, o ile nie chce się utworzyć wartości ósemkowej. Python 2.6 potraktuje takiłańcuch jak liczbę o podstawie 8, co może nie działać tak, jak byśmy tego oczekiwali. 010 to w wersji 2.6 zawsze dziesiętne 8, a nie 10, jak można by się

spodziewać. Z tego właśnie powodu (a także z uwagi na symetrię formatu ósemkowego, szesnastkowego i dwójkowego) format ósemkowy został w wersji 3.0 zmieniony. W Pythonie 3.0 należy użyć `0o010` i najlepiej zrobić tak samo również w 2.6.

Po drugie, warto zauważyc, że te literały mogą tworzyć stosunkowo długie liczby całkowite. Poniższy kod tworzy na przykład liczbę całkowitą w notacji szesnastkowej, a następnie wyświetla ją najpierw w notacji dziesiętnej, a później ósemkowej oraz dwójkowej (z użyciem konwerterek). Kod ten wykonywany jest w Pythonie 2.6 w celu ukazania dłuższej precyzji:

A skoro już mowa o liczbach dwójkowych, w kolejnym podrozdziale zaprezentowane zostaną narzędzia służące do przetwarzania pojedynczych bitów.

Operacje poziomu bitowego

Poza normalnymi operacjami liczbowymi (dodawaniem, odejmowaniem i tak dalej) Python obsługuje również większość wyrażeń numerycznych dostępnych w języku C. Obejmuje to także operatory traktujące liczby całkowite jako łańcuchy bitów binarnych. Poniżej znajduje się przykład wykonania przesunięcia bitowego oraz operacji typu Boolean.

```
>>> x = 1                                # 0001
>>> x << 2                                # Przesunięcie o 2 bity w lewo: 0100
4
>>> x | 2                                  # Bitowe OR: 0011
3
>>> x & 1                                  # Bitowe AND: 0001
1
```

W pierwszym wyrażeniu binarne 1 (w systemie dwójkowym — 0001) przesuwane jest w lewo o dwa miejsca, tak by utworzyć binarne 4 (0100). Ostatnie dwie operacje wykonują binarne OR ($0001 \mid 0010 = 0011$) oraz binarne AND ($0001 \& 0001 = 0001$). Takie operacje masek bitowych pozwalają na kodowanie kilku opcji (flag) i innych wartości wewnętrz jednej liczby całkowitej.

Jest to jeden z obszarów, w których obsługa liczb dwójkowych i szesnastkowych w Pythonie 2.6 oraz 3.0 staje się szczególnie przydatna. Pozwala nam ona mianowicie zapisywać i badać liczby wedługłańcuchów bitów:

```
>>> X = 0b0001          # Literaly dwójkowe
>>> X << 2            # Przesunięcie w lewo
4
>>> bin(X << 2)      # Łancuch cyfr dwójkowych
'0b100'

>>> bin(X | 0b010)     # Bitowe OR
'0b11'
>>> bin(X & 0b1)       # Bitowe AND
'0b1'

>>> X = 0xFF           # Literaly szesnastkowe
>>> bin(X)
```

```

'0b11111111'
>>> X ^ 0b10101010          # Bitowe XOR
85
>>> bin(X ^ 0b10101010)
'0b1010101'
>>> int('1010101', 2)       # Łąćuch znaków na liczbę całkowitą zgodnie z podstawą
85
>>> hex(85)                # Łąćuch cyfr szesnastkowych
'0x55'

```

Nie będziemy za bardzo zagłębiać się w operacje poziomu bitowego. Są one w Pythonie obsługiwane i dostępne; przydają się, kiedy musimy sobie radzić z pakietami sieciowymi czy spakowanymi danymi binarnymi utworzonymi przez program napisany w języku C. Należy jednak pamiętać, że operacje poziomu bitowego nie są w języku wysokiego poziomu, takim jak Python, tak ważne jak w języku niskiego poziomu, jak C. Generalnie jeśli przyjdzie nam ochota zająć się w Pythonie bitami, należy się zastanowić, w jakim języku tak naprawdę piszemy. W Pythonie istnieją lepsze sposoby zapisywania informacji od łańcuchów bitowych.



W zbliżającym się wydaniu 3.1 Pythona metoda liczb całkowitych `bit_length` pozwala także na sprawdzenie liczby bitów wymaganych do reprezentowania wartości liczby w systemie dwójkowym. Ten sam rezultat można osiągnąć, odejmując 2 od długości łańcucha `bin` za pomocą wbudowanej funkcji `len` widzianej już w rozdziale 4., choć rozwiązanie to może być mniej wydajne:

```

>>> X = 99
>>> bin(X), X.bit_length()
('0b1100011', 7)
>>> bin(256), (256).bit_length()
('0b100000000', 9)
>>> len(bin(256)) - 2
9

```

Inne wbudowane narzędzia liczbowe

Poza typami podstawowymi Python udostępnia wbudowane funkcje i moduły biblioteki standardowej służące do przetwarzania liczb. Wbudowane funkcje `pow` i `abs` służą na przykład do potęgowania i obliczania wartości bezwzględnej liczb. Poniżej znajdują się przykłady operacji dostępnych we wbudowanym module `math` (zawierającym większość narzędzi z biblioteki matematycznej języka C), a także kilka funkcji wbudowanych.

```

>>> import math
>>> math.pi, math.e           # Często używane stałe
(3.1415926535897931, 2.7182818284590451)

>>> math.sin(2 * math.pi / 180)      # Sinus, tangens, cosinus
0.034899496702500969

>>> math.sqrt(144), math.sqrt(2)    # Pierwiastek kwadratowy
(12.0, 1.4142135623730951)

>>> pow(2, 4), 2 ** 4            # Potęgowanie
(16, 16)

>>> abs(-42.0), sum((1, 2, 3, 4))  # Wartość bezwzględna, suma
(42.0, 10)

>>> min(3, 1, 2, 4), max(3, 1, 2, 4)  # Minimum, maksimum
(1, 4)

```

Pokazana w przykładzie funkcja `sum` działa na sekwencji liczb, natomiast `min` i `max` przyjmują albo sekwencję, albo pojedyncze argumenty. Istnieje kilka sposobów na opuszczenie części ułamkowej z liczb zmiennoprzecinkowych. Wcześniej spotkaliśmy się z odcinaniem i zaokrąglaniem do najbliższej mniejszej liczby całkowitej. Możemy również zaokrągać liczby, zarówno liczbowo, jak i na potrzeby wyświetlania:

```
>>> math.floor(2.567), math.floor(-2.567)      # Zaokrąglenie do najbliższej mniejszej liczby całkowitej
(2, -3)

>>> math.trunc(2.567), math.trunc( 2.567)       # Odcięcie (pominięcie części ułamkowej)
(2, 2)

>>> int(2.567), int( 2.567)                     # Odcięcie (konwersja na liczbę całkowitą)
(2, 2)

>>> round(2.567), round(2.467), round(2.567, 2) # Zaokrąglenie (Python 3.0)
(3, 2, 2.5699999999999998)

>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)      # Zaokrąglenie na potrzeby wyświetlania (rozdział 7.)
('2.6', '2.57')
```

Jak widzieliśmy wcześniej, ostatni przykład zwracała łańcuchy znaków, które normalnie byśmy wyświetlili, a także obsługuje różne opcje formatowania. Jak już wspomnieliśmy, przedostatni przykład zwróciłby (3, 2, 2.57), gdybyśmy umieścili go w wywoaniu `print` w celu uzyskania zapisu nieco bardziej przyjaznego dla użytkownika. Mimo to te dwa przykłady różnią się od siebie — `round` zaokrąga liczbę zmiennoprzecinkową, jednak przechowuje ją w pamięci, natomiast formatowanie z użyciem łańcucha znaków zwracała łańcuch znaków i nie przechowuje zmodyfikowanej liczby:

```
>>> (1 / 3), round(1 / 3, 2), ('%.2f' % (1 / 3))
(0.333333333333331, 0.3300000000000002, '0.33')
```

Co ciekawe, w Pythonie istnieją trzy sposoby obliczenia pierwiastka kwadratowego — za pomocą funkcji modułu, wyrażenia lub funkcji wbudowanej (osoby zainteresowane wydajnością tych rozwiązań odsyłam do ćwiczenia i jego rozwiązania na końcu czwartej części książki, gdzie sprawdzimy, które z nich działa szybciej).

```
>>> import math
>>> math.sqrt(144)                                # Moduł
12.0
>>> 144 ** .5                                    # Wyrażenie
12.0
>>> pow(144, .5)                                 # Funkcja wbudowana
12.0

>>> math.sqrt(1234567890)                         # Większe liczby
35136.418286444619
>>> 1234567890 ** .5
35136.418286444619
>>> pow(1234567890, .5)
35136.418286444619
```

Warto zauważyć, że moduły biblioteki standardowej, takie jak `math`, trzeba importować, jednak funkcje wbudowane, takie jak `abs` czy `round`, dostępne są zawsze bez importowania. Innymi słowy, moduły to komponenty zewnętrzne, natomiast funkcje wbudowane znajdują się w domniemanej przestrzeni nazw, którą Python automatycznie przeszukuje w celu odnalezienia nazw użytych w programie. Ta przestrzeń nazw odpowiada modułowi o nazwie `builtins` w Pythonie 3.0 (`__builtin__` w wersji 2.6).Więcej informacji na temat rozwijania nazw można znaleźć w częściach książki poświęconych funkcjom i modułom. Kiedy jednak słyszmy słowo „moduł”, od razu powinno nam przyjść do głowy słowo „importowanie”.

Importować trzeba również moduł `random` z biblioteki standardowej. Moduł ten udostępnia narzędzia służące na przykład do wybierania losowej liczby zmiennoprzecinkowej o wartości między 0 a 1, wybierania losowej liczby całkowitej o wartości między dwoma podanymi liczbami, a także do losowego wybierania elementu z sekwencji.

```
>>> import random
>>> random.random()
0.44694718823781876
>>> random.random()
0.28970426439292829

>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
4

>>> random.choice(['Żywot Briana', 'Święty Graal', 'Sens życia'])
'Żywot Briana'
>>> random.choice(['Żywot Briana', 'Święty Graal', 'Sens życia'])
'Święty Graal'
```

Moduł `random` może się przydać do tasowania kart w grze, losowego wybierania zdjęć w pokazie slajdów, wykonywania symulacji statystycznych i wielu innych zastosowań. Więcej informacji na jego temat można znaleźć w dokumentacji biblioteki Pythona.

Inne typy liczbowe

W tym rozdziale korzystaliśmy na razie z podstawowych typów liczbowych Pythona — liczb całkowitych, liczb zmiennoprzecinkowych oraz liczb zespolonych. Wystarczą one w zupełności w większości zastosowań, z jakimi spotka się większość programistów. Python udostępnia jednak kilka bardziej egzotycznych typów liczbowych, które zasługują na przyjrzenie się im.

Typ liczby dziesiętnej

W wersji 2.4 Pythona wprowadzono nowy podstawowy typ liczbowy — liczbę dziesiętną, formalnie znaną jako `Decimal`. Z punktu widzenia składni liczby te tworzy się, wywołując funkcję z zainportowanego modułu, a nie wyrażenie z literałem. Z funkcjonalnego punktu widzenia liczby dziesiętne przypominają liczby zmiennoprzecinkowe, jednak mają stałą liczbę miejsc dziesiętnych. Tym samym liczba dziesiętna jest wartością zmiennoprzecinkową o stałej precyzyji.

Liczba dziesiętna pozwala na przykład na utworzenie wartości zmiennoprzecinkowej, która zawsze zachowuje dwie pozycje dziesiętne. Co więcej, możemy również określić, w jaki sposób należy zaokrągać lub odcinać dodatkowe pozycje dziesiętne. Choć liczby dziesiętne są zazwyczaj nieco mniej wydajne od zwykłych liczb zmiennoprzecinkowych, dobrze nadają się do przedstawiania ilości o stałej precyzyji, na przykład pieniędzy, a także mogą pomóc uzykać lepszą dokładność.

Podstawy

Ostatnie stwierdzenie zasługuje na wyjaśnienie. Arytmetyka liczb zmiennoprzecinkowych jest mało dokładna z uwagi na ograniczenie miejsca wykorzystywanego do przechowywania wartości. Poniższe wyrażenie powinno na przykład zwrócić zero, jednak tak się nie dzieje. Rezultat jest bliski零, jednak nie ma wystarczającej liczby bitów do bycia dokładnym.

```
>>> 0.1 + 0.1 + 0.1 - 0.3  
5.5511151231257827e-17
```

Wyświetlenie wyniku za pomocą print, w formacie przyjaznym dla użytkownika, nie pomoże, ponieważ urządzenia odpowiedzialne za arytmetykę liczb zmiennoprzecinkowych są ograniczone w zakresie dokładności.

```
>>> print(0.1 + 0.1 + 0.1 - 0.3)  
5.55111512313e-17
```

W przypadku liczb dziesiętnych wynik może być dokładny.

```
>>> from decimal import Decimal  
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')  
Decimal('0.0')
```

Jak widać powyżej, obiekty liczb dziesiętnych tworzy się, wywołując funkcję konstruktora o nazwie Decimal znajdująca się w module decimal i przekazując jej łańcuchy znaków o pożądanej liczbie pozycji dziesiętnych, jaką będzie miał również wynik. Jeśli jest to niezbędne, możemy wykorzystać funkcję str do przekształcenia wartości zmiennoprzecinkowych na łańcuchy znaków. Kiedy w wyrażeniach łączy się ze sobą liczby dziesiętne o różnej precyzji, Python automatycznie konwertuje je w górę do największej liczby pozycji dziesiętnych.

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')  
Decimal('0.00')
```



W Pythonie 3.1 (który został wydany już po publikacji niniejszej książki) można także utworzyć obiekt liczby dziesiętnej z obiektu liczby zmiennoprzecinkowej za pomocą wywołania w postaci decimal.Decimal.from_float(1.25). Konwersja jest dokładna, jednak czasami może zwracać znaczną liczbę cyfr.

Globalne ustawienie precyzji

Inne narzędzia z modułu decimal można wykorzystać na przykład do ustawienia precyzji wszystkich liczb dziesiętnych, a także by skonfigurować obsługę błędów. Obiekt kontekstowy z tego modułu pozwala na przykład na określanie precyzji (liczby miejsc dziesiętnych) i trybu zaokrąglania (w dół, w górę). Ustawienie precyzji stosowane jest globalnie dla wszystkich liczb dziesiętnych utworzonych w wątku wywołującym.

```
>>> import decimal  
>>> decimal.Decimal(1) / decimal.Decimal(7)  
Decimal('0.1428571428571428571429')  
  
>>> decimal.getcontext().prec = 4  
>>> decimal.Decimal(1) / decimal.Decimal(7)  
Decimal('0.1429')
```

Jest to szczególnie przydatne w aplikacjach finansowych, w których grosze reprezentowane są jako dwa miejsca ułamkowe. Liczby dziesiętne są właściwie w tym kontekście alternatywą dla ręcznego zaokrąglania i formatowania za pomocą łańcuchów znaków.

```
>>> 1999 + 1.33  
2000.329999999999  
>>>  
>>> decimal.getcontext().prec = 2  
>>> pay = decimal.Decimal(str(1999 + 1.33))  
>>> pay  
Decimal('2000.33')
```

Dziesiętne menedżery kontekstu

W Pythonie 2.6 i 3.0 (oraz kolejnych wersjach) można również tymczasowo przywrócić ustalenia precyzyji za pomocą instrukcji menedżera kontekstu `with`. Precyza przywracana jest do oryginalnej wartości na wyjściu z instrukcją:

```
C:\misc> C:\Python30\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33333333333333333333333333333333')
```

Choć jest to przydatne, instrukcja ta wymaga o wiele większej wiedzy niż uzyskana w tym miejscu książki. Omówienie instrukcji `with` znajduje się w rozdziale 33.

Ponieważ korzystanie z typu liczby dziesiętnej nadal jest w praktyce rzadkością, osoby zainteresowane odsyłam do dokumentacji biblioteki standardowej Pythona, a także interaktywnego modułu z pomocą. A ponieważ liczby dziesiętne rozwiązują często te same problemy z dokładnością liczb zmienoprzecinkowych co liczby ułamkowe, przejdźmy teraz do kolejnego podrozdziału w celu przekonania się, jak wypada porównanie tych dwóch typów.

Typ liczby ułamkowej

W Pythonie 2.6 oraz 3.0 zadebiutował nowy typ liczbowy — `Fraction` (ułamek), implementujący obiekt *liczby wymiernej*. W sposób jawnym zachowuje on zarówno licznik, jak i mianownik, tak by uniknąć części niedokładności i ograniczeń arytmetyki liczb zmienoprzecinkowych.

Podstawy

Typ `Fraction` jest w pewnym sensie krewnym istniejącego typu `Decimal` o ustalonej precyzyji, opisanego w poprzednim podrozdziale, ponieważ oba mogą być wykorzystywane do kontrolowania dokładności liczb poprzez ustalenie stałej liczby miejsc dziesiętnych, a także polityki zaokrąglania i odcinania. Typ ten jest także wykorzystywany w podobny sposób — tak jak `Decimal`, `Fraction` znajduje się w module. W celu uzyskania obiektu tego typu należy zaimportować konstruktor i przekazać mu licznik oraz mianownik. Poniższy zapis sesji interaktywnej pokazuje, jak można tego dokonać:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)                                     # Licznik, mianownik
>>> y = Fraction(4, 6)                                     # Uproszczony do 2, 3 przez gcd

>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

Po utworzeniu obiekty `Fraction` można wykorzystywać w wyrażeniach arytmetycznych w zwykły sposób:

```
>>> x + y
Fraction(1, 1)
>>> x - y
Fraction(-1, 3)                                         # Wyniki są dokładne: licznik, mianownik
>>> x * y
Fraction(2, 9)
```

Obiekty `Fraction` mogą również być tworzone złańcuchów znaków liczb zmiennoprzecinkowych, podobnie jak liczby dziesiętne:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

Dokładność liczb

Warto zwrócić uwagę na różnice w porównaniu z arytmetyką liczb zmiennoprzecinkowych, ograniczoną przez możliwości procesorów do obliczeń zmiennoprzecinkowych. W celach porównawczych poniżej zaprezentowano te same działania wykonane za pomocą obiektów liczb zmiennoprzecinkowych wraz z uwagami dotyczącymi ich ograniczonej dokładności.

```
>>> a = 1 / 3.0                                         # Dokładność taka, jak procesora do obliczeń zmiennoprzecinkowych
>>> b = 4 / 6.0                                         # Może w miarę obliczeń tracić precyzję
>>> a
0.3333333333333333
>>> b
0.6666666666666666

>>> a + b
1.0
>>> a - b
-0.3333333333333331
>>> a * b
0.2222222222222221
```

Ograniczenia liczb zmiennoprzecinkowych są szczególnie widoczne w przypadku wartości, których nie da się reprezentować dokładnie z uwagi na ograniczoną liczbę bitów w pamięci. Typy `Fraction` i `Decimal` mogą dawać dokładne wyniki, jednak kosztem szybkości. W poniższym przykładzie (powtórzonym z poprzedniego podrozdziału) liczby zmiennoprzecinkowe nie dają oczekiwanej wyniku równego zero, natomiast dwa pozostałe typy robią to bez problemu:

```
>>> 0.1 + 0.1 + 0.1 - 0.3                            # To powinno wynosić zero (blisko, ale nie dokładnie)
5.5511151231257827e-17

>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)

>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Co więcej, liczby wymierne oraz dziesiętne mogą dawać bardziej intuicyjne i dokładne wyniki, niż czasami robią to liczby zmiennoprzecinkowe, choć w inny sposób (za pomocą użycia reprezentacji wymiernej i ograniczenia precyzji).

```
>>> 1 / 3                                # W Pythonie 2.6 należy użyć "3.0" w celu uzyskania prawdziwego dzielenia
0.3333333333333333
```

```
>>> Fraction(1, 3)                      # Dokładność liczb
Fraction(1, 3)
```

```
>>> import decimal
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal(1) / decimal.Decimal(3)
Decimal('0.33')
```

Tak naprawdę liczby ułamkowe zarówno zachowują dokładność, jak i automatycznie skracają wyniki. Kontynuując poprzednią sesję interaktywną:

```
>>> (1 / 3) + (6 / 12)                  # W Pythonie 2.6 należy użyć ".0" w celu uzyskania prawdziwego dzielenia
0.8333333333333332
```

```
>>> Fraction(6, 12)                     # Automatyczne skrócenie ułamka
Fraction(1, 2)
```

```
>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)
```

```
>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')
```

```
>>> 1000.0 / 1234567890
8.100000073710001e-07
```

```
>>> Fraction(1000, 1234567890)
Fraction(100, 123456789)
```

Konwersje i typy mieszane

By móc obsługiwać konwersje na ułamki, obiekty zmiennoprzecinkowe udostępniają teraz metodę zwracającą czynnik ich licznika i mianownika, ułamki mają metodę `from_float`, natomiast metoda `float` przyjmuje obiekt `Fraction` jako argument. By przekonać się, jak to działa, wystarczy prześledzić poniższą sesję interaktywną (znak * w drugim teście to składnia specjalna rozszerzająca krotkę na pojedyncze argumenty; więcej informacji na ten temat znajduje się w omówieniu przekazywania argumentów do funkcji w rozdziale 18.).

```
>>> (2.5).as_integer_ratio()            # Metoda obiektu liczby zmiennoprzecinkowej
(5, 2)
```

```
>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio()) # Konwersja z liczby zmiennoprzecinkowej na ułamek:
                                         # 2 argumenty
                                         # To samo co Fraction(5, 2)
>>> z
Fraction(5, 2)
```

```
>>> x
                                         # x z poprzedniej interakcji
Fraction(1, 3)
>>> x + z
Fraction(17, 6)                         # 5/2 + 1/3 = 15/6 + 2/6
```

```

>>> float(x)                                # Konwersja z ułamka na liczbę zmiennoprzecinkową
0.3333333333333333
>>> float(z)
2.5
>>> float(x + z)                           # Konwersja z liczby zmiennoprzecinkowej na ułamek: inny sposób
2.8333333333333335
>>> 17 / 6
2.8333333333333335

>>> Fraction.from_float(1.75)               # Konwersja z liczby zmiennoprzecinkowej na ułamek: inny sposób
Fraction(7, 4)
>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)

```

Mieszanie typów jest w pewnym stopniu dozwolone w wyrażeniach, jednak czasami trzeba ręcznie rozszerzyć ułamek w celu zachowania dokładności. By przekonać się, jak to działa, warto przyjrzeć się poniższym przykładom.

```

>>> x
Fraction(1, 3)
>>> x + 2                                  # Ułamek + liczba całkowita -> ułamek
Fraction(7, 3)
>>> x + 2.0                                # Ułamek + liczba zmiennoprzecinkowa -> liczba zmiennoprzecinkowa
2.3333333333333335
>>> x + (1./3)                            # Ułamek + liczba zmiennoprzecinkowa -> liczba zmiennoprzecinkowa
0.6666666666666666

>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3)                      # Ułamek + ułamek -> ułamek
Fraction(5, 3)

```

Uwaga! Choć można dokonać konwersji z liczby zmiennoprzecinkowej na ułamek, w niektórych przypadkach nie do uniknięcia jest pewna utrata precyzji, ponieważ liczba w pierwotnej postaci zmiennoprzecinkowej jest niedokładna. Kiedy jest to niezbędne, można uprościć takie wyniki, ograniczając maksymalną wartość mianownika.

```

>>> 4.0 / 3
1.333333333333333
>>> (4.0 / 3).as_integer_ratio()           # Utrata precyzji z liczby zmiennoprzecinkowej
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488. # 5 / 3 (lub niedaleko!)
1.6666666666666667

>>> a.limit_denominator(10)                  # Uproszczenie do najbliższego ułamka
Fraction(5, 3)

```

W celu uzyskania większej ilości informacji na temat typu `Fraction` warto samodzielnie poeksperymentować lub odwołać się do dokumentacji biblioteki Pythona 2.6 i 3.0 bądź innych materiałów.

Zbiory

Python 2.4 wprowadził nowy typ kolekcji — *zbiór* (ang. *set*). Zbiór to nieuporządkowana kolekcja unikalnych i niezmiennych obiektów, obsługująca działania odpowiadające matematycznej teorii zbiorów. Zgodnie z definicją element może się pojawić w zbiorze tylko raz, bez względu na to, ile razy zostanie do niego dodany. Z tego powodu zbiory mają wiele zastosowań, w szczególności w działaniach skupiających się na liczbach oraz bazach danych.

Ponieważ zbiory są kolekcjami innych obiektów, współdzielą z obiektami takimi, jak listy i słowniki pewne zachowania, które pozostają poza zakresem niniejszego rozdziału. Na zbiorach można na przykład wykonywać iterację, mogą one rosnąć i kurczyć się na żądanie, a także zawierać obiekty różnych typów. Jak zobaczymy, zbiór zachowuje się dość podobnie do kluczy słownika bez wartości, jednak obsługuje przy tym dodatkowe operacje.

Ponieważ zbiory są nieuporządkowane i nie odwzorowują kluczy na wartości, nie są ani typem sekwencji, ani odwzorowania — przynależą raczej do własnej kategorii. Co więcej, ponieważ z natury są ściśle związane z matematyką (i wielu Czytelnikom mogą się wydawać bardziej akademickie, a także o wiele rzadziej wykorzystywane niż bardziej powszechnie obiekty, takie jak słowniki), podstawowe zastosowania obiektów zbiorów Pythona omówimy właśnie w tym rozdziale.

Podstawy zbiorów w Pythonie 2.6

Obecnie istnieje kilka sposobów tworzenia zbiorów — w zależności od tego, czy korzysta się z Pythona 2.6, czy 3.0. Ponieważ niniejsza książka omawia obie wersje, zacznijmy od sposobu z Pythona 2.6, który jest również dostępny (i w pewnym stopniu nadal wymagany) w wersji 3.0. Sposób ten rozszerzymy za chwilę za pomocą opcji dostępnych w Pythonie 3.0. By utworzyć obiekt zbioru, należy przekazać sekwencję lub inny obiekt, po którym można wykonywać iterację, do wbudowanej funkcji `set`.

```
>>> x = set('abcde')
>>> y = set('bdxyz')
```

Otrzymujemy w ten sposób obiekt zbioru zawierający wszystkie elementy z przekazanego obiektu (warto zauważyć, że zbory nie są uporządkowane w zakresie pozycji, dlatego nie są sekwencjami).

```
>>> x
set(['a', 'c', 'b', 'e', 'd']) # Format wyświetlania z wersji 2.6
```

Utworzone w ten sposób zbory obsługują najważniejsze działania matematyczne na zbiorach za pomocą *wyrażeń* z operatorami. Warto zauważyć, że nie można tych działań wykonywać na zwykłych sekwencjach — trzeba z nich najpierw utworzyć zbory.

```
>>> 'e' in x # Istnienie w zbiorze
True

>>> x - y # Różnica zbiorów
set(['a', 'c'])

>>> x | y # Suma zbiorów
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y # Część wspólna zbiorów
set(['b', 'd'])
```

```
>>> x ^ y                                # Różnica symetryczna (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])
>>> x > y, x < y                         # Nadzbiór, podzbiór
(False, False)
```

Oprócz wyrażeń obiekt zbioru udostępnia również *metody* odpowiadające tym i innym działaniom, obsługujące zmiany zbiorów. Metoda `add` wstawia jeden element, `update` to suma zbiorów w miejscu, a `remove` usuwa element o podanej wartości (by zobaczyć wszystkie dostępne metody, należy wywołać `dir` na dowolnej instancji zbioru lub nazwie typu `set`). Zakładając, że `x` i `y` mają wartości jak w poprzedniej sesji interaktywnej:

```
>>> z = x.intersection(y)                  # To samo co x & y
>>> z
set(['b', 'd'])
>>> z.add('MIELONKA')                   # Wstawienie jednego elementu
>>> z
set(['b', 'd', 'MIELONKA'])
>>> z.update(set(['X', 'Y']))            # Połączenie: suma w miejscu
>>> z
set(['Y', 'X', 'b', 'd', 'MIELONKA'])
>>> z.remove('b')                        # Usunięcie jednego elementu
>>> z
set(['Y', 'X', 'd', 'MIELONKA'])
```

Ponieważ zbiory są pojemnikami, na których można wykonywać iterację, można je także wykorzystywać w operacjach takich, jak `len`, pętle `for` czy listy składane. Ponieważ jednak są nieuporządkowane, nie obsługują operacji na sekwencjach, takich jak indeksowanie czy wycinki.

```
>>> for item in set('abc'): print(item * 3)
...
aaa
ccc
bbb
```

Wreszcie, choć zaprezentowane powyżej wyrażenia zbiorów zazwyczaj wymagają podania dwóch zbiorów, odpowiadające im metody działają także na dowolnych innych typach, na których można wykonywać iterację:

```
>>> S = set([1, 2, 3])
>>> S | set([3, 4])                      # Wyrażenie wymaga, by oba były zbiorami
set([1, 2, 3, 4])
>>> S | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> S.union([3, 4])                      # Metody zezwalają na typy, po których można iterować
set([1, 2, 3, 4])
>>> S.intersection([1, 3, 5])
set([1, 3])
>>> S.issubset(range(-5, 5))
True
```

Więcej informacji na temat działań na zbiorach można znaleźć w dokumentacji biblioteki Pythona lub książkach. Choć działania na zbiorach można w Pythonie kodować ręcznie za pomocą innych typów, takich jak listy i słowniki (i często tak właśnie kiedyś się robiło), wbudowane zbiory Pythona wykorzystują wydajne algorytmy i techniki implementacyjne, które ułatwiają szybkie wykonywanie standardowych operacji.

Literały zbiorów w Pythonie 3.0

Dla osób, które uważają zbiorzy za świetny wynalazek, mam dobrą wiadomość: teraz stały się one jeszcze bardziej interesujące. W Pythonie 3.0 nadal możemy tworzyć obiekty zbiorów za pomocą wbudowanego konstruktora `set`, jednak dodano również nową formę literalu zbioru wykorzystującą nawiasy klamrowe zarezerwowane wcześniej dla słowników. W wersji 3.0 poniższe dwa wiersze kodu są równoważne:

```
set([1, 2, 3, 4])                                # Wywołanie wbudowanego konstruktora  
{1, 2, 3, 4}                                     # Literały zbiorów z wersji 3.0
```

Powyzsza składnia ma sens, biorąc pod uwagę to, że zbiorzy są właściwie jak *słowniki bez wartości* — ponieważ elementy zbioru są nieuporządkowane, unikalne i niezmienne, zachowują się jak klucze słowników. To podobieństwo w działaniu staje się jeszcze bardziej uderzające, gdy weźmiemy pod uwagę, że w wersji 3.0 listy kluczów słowników są obiektami *widoku*, obsługującymi działania podobne do zbiorów, takie jak część wspólna czy suma (więcej informacji o obiektach widoku słowników znajduje się w rozdziale 8.).

Tak naprawdę bez względu na sposób konstruowania Python 3.0 wyświetla zbiór za pomocą nowego formatu literalu. Wbudowany konstruktor `set` jest nadal wymagany w wersji 3.0 do tworzenia pustych zbiorów i budowania zbiorów z istniejących obiektów, na których można wykonywać iterację (oprócz wykorzystywania zbiorów składanych omówionych w dalszej części rozdziału), jednak nowy literał przydaje się do inicjalizowania zbiorów o znanej strukturze.

```
C:\Misc> c:\python30\python  
>>> set([1, 2, 3, 4])                                # Wbudowany konstruktor: tak samo jak w 2.6  
{1, 2, 3, 4}  
>>> set('mielonka')                               # Dodanie wszystkich elementów obiektu, po którym można iterować  
{'a', 'e', 'i', 'k', 'l', 'o', 'n'}  
>>> {1, 2, 3, 4}                                    # Literały zbiorów: nowość w 3.0  
{1, 2, 3, 4}  
>>> S = {'m', 'i', 'e', 'l', 'o', 'n', 'k', 'a'}  
>>> S.add('konserwowa')  
>>> S  
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n', 'konserwowa'}
```

Wszystkie omówione w poprzednim podrozdziale operacje przetwarzające zbiorzy działają w Pythonie 3.0 tak samo, lecz wynikowe zbiorzy wyświetlane są w odmienny sposób:

```
>>> S1 = {1, 2, 3, 4}  
>>> S1 & {1, 3}                                     # Część wspólna  
{1, 3}  
>>> {1, 5, 3, 6} | S1                            # Suma  
{1, 2, 3, 4, 5, 6}  
>>> S1 - {1, 3, 4}                                # Różnica  
{2}  
>>> S1 > {1, 3}                                   # Nadzbiór  
True
```

Warto pamiętać, że {} jest w Pythonie nadal słownikiem. *Puste* zbiorzy muszą być tworzone za pomocą wbudowanego konstruktora `set` i w ten sam sposób wyświetlane.

```
>>> S1 - {1, 2, 3, 4}                                # Puste zbiorzy wyświetlane są inaczej  
set()  
>>> type({})                                       # Ponieważ {} jest pustym słownikiem  
<class 'dict'>  
  
>>> S = set()                                         # Inicjalizacja pustego zbioru  
>>> S.add(1.23)  
>>> S  
{1.23}
```

Tak jak w Pythonie 2.6, zbiory utworzone za pomocą literałów z wersji 3.0 obsługują te same metody. Część z nich — w przeciwnieństwie do wyrażeń — pozwala na wykorzystywanie ogólnych argumentów typów, na których można wykonywać iterację.

```
>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}

>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True
```

Ograniczenia niezmienności i zamrożone zbiory

Zbiory są elastycznymi obiektami o dużych możliwościach, jednak mają w Pythonie 2.6 i 3.0 jedno ograniczenie, o którym należy pamiętać. W dużej mierze z powodu sposobu ich implementacji zbiory mogą zawierać jedynie obiekty niezmienne. Tym samym w zbiorach nie można osadzać list i słowników, choć można to zrobić w przypadku krotek, jeśli niezbędne jest przechowanie wartości złożonych. Przy wykorzystywaniu w działańach na zbiorach krotki porównywane są pod względem pełnej wartości.

```
>>> S
{1, 23}
>>> S.add([1, 2, 3])                                     # W zbiorze działają jedynie obiekty niezmienne
TypeError: unhashable type: 'list'
>>> S.add({'a':1})                                     # W zbiorze działają jedynie obiekty niezmienne
TypeError: unhashable type: 'dict'
>>> S.add((1, 2, 3))                                    # Ani listy, ani słowniki; krotki są OK
>>> S
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}                      # Suma: to samo co S.union(...)
{1.23, (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S                                     # Istnienie w zbiorze: porównanie po pełnej wartości
True
>>> (1, 4, 3) in S
False
```

Krotki można w zbiorach wykorzystać do reprezentowania dat, rekordów, adresów IP czy adresów pocztowych (więcej informacji o krotkach znajduje się w dalszej części książki). Same zbiory są zmienne, dlatego nie mogą być zagnieżdżane w innych zbiorach w sposób bezpośredni. Jeśli jednak przechowanie zbioru wewnętrz innego zbioru jest konieczne, wywołanie wbudowanego konstruktora frozenset działa tak samo jak set, choć tworzy niezmienny zbiór, którego nie można modyfikować i który tym samym da się osadzać w innych zbiorach.

Zbiory składane w Pythonie 3.0

Obok literałów w wersji 3.0 wprowadzono także konstrukcję zbioru składanego (ang. *set comprehension*). Jest ona podobna do list składanych omówionych pokrótko w rozdziale 4., jednak zapisywana jest w nawiasach klamrowych zamiast kwadratowych i wykonywana w celu

utworzenia zbioru, a nie listy. Zbiory składane wykonują pętlę i zbierają wynik wyrażenia z każdą iteracją. Zmienna pętli daje dostęp do aktualnej wartości iteracji, z której można skorzystać w wyrażeniu zbierającym. Wynikiem jest nowy zbiór utworzony dzięki wykonaniu kodu, z normalnymi właściwościami zbiorów.

```
>>> {x ** 2 for x in [1, 2, 3, 4]} # Zbiory składane z wersji 3.0
{16, 1, 4, 9}
```

W powyższym wyrażeniu pętla zapisywana jest po prawej stronie, natomiast wyrażenie zbierające po lewej ($x^{**} 2$). Tak jak w przypadku list składanych, otrzymujemy właściwie to, co mówi wyrażenie: „Daj mi nowy zbiór zawierający x podniesione do kwadratu dla każdego x z listy”. Zbiory składane mogą również wykonywać iterację po innych typach obiektów, takich jak łańcuchy znaków (pierwszy z poniższych przykładów ilustruje oparty na zbiorach składanych sposób utworzenia zbioru z istniejącego elementu, na którym można wykonywać iterację).

```
>>> {x for x in 'mielonka'} # To samo co: set('mielonka')
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n'}
>>> {c * 4 for c in 'mielonka'} # Zbiór zebranych wyników wyrażenia
{'iiii', 'eeee', 'oooo', 'nnnn', 'mmmm', 'aaaa', 'llll', 'kkkk'}
>>> {c * 4 for c in 'szynkajajka'}
{'yyyy', 'jjjj', 'nnnn', 'zzzz', 'ssss', 'aaaa', 'kkkk'}
>>> S = {c * 4 for c in 'mielonka'}
>>> S | {'mmmm', 'xxxx'}
{'mmmm', 'eeee', 'xxxx', 'oooo', 'nnnn', 'iiii', 'aaaa', 'llll', 'kkkk'}
>>> S & {'mmmm', 'xxxx'}
{'mmmm'}
```

Ponieważ inne właściwości zbiorów składanych oparte są na koncepcjach, których jeszcze nie omawialiśmy, odłożymy ich przedstawienie do późniejszej części książki. W rozdziale 8. spotkamy pierwszego kremnego zbiorów składanych — słowniki składane. O wiele więcej na temat wszystkich typów obiektów składanych (list, zbiorów, słowników i generatorów) będę miał do powiedzenia później, zwłaszcza w rozdziałach 14. oraz 20. Jak się przekonamy, wszystkie obiekty składane, w tym zbior, obsługują dodatkową składnię wykraczającą poza pokazaną tutaj, w tym zagnieżdżone pętle i testy `if`, co może być trudne do zrozumienia, dopóki nie omówi się bardziej złożonych instrukcji.

Dlaczego zbior?

Działania na zbiorach mają wiele powszechnie wykorzystywanych zastosowań; niektóre z nich są bardziej praktyczne niż matematyczne. Przykładowo ponieważ elementy przechowywane są w zbiorze jedynie raz, zbiory można wykorzystywać do odfiltrowywania duplikatów z kolekcji innego typu. Wystarczy przekształcić kolekcję na zbiór, a następnie przekształcić ją z powrotem (ponieważ na zbiorach można wykonywać iterację, działając one w wywołaniu konstruktora `list`):

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L)) # Usunięcie duplikatów
>>> L
[1, 2, 3, 4, 5]
```

Zbiory można także wykorzystać do śledzenia, gdzie się już było, przy przechodzeniu wykresu czy innej struktury cyklicznej. Przykłady przechodniego ładowania modułów czy wyświetla-

nia drzева dziedziczenia, odpowiednio z rozdziałów 24. i 30., muszą śledzić odwiedzone elementy w celu uniknięcia pętli. Choć zapisywanie odwiedzonego stanu w postaci kluczy słownika jest wydajne, zbiory są alternatywą, która jest właściwie równoważna (i może być bardziej lub mniej intuicyjna, w zależności od tego, kogo o to zapytać).

Wreszcie zbiory przydają się w przypadku radzenia sobie z większymi zbiorami danych (na przykład wynikami zapytań do bazy danych). Część wspólna dwóch zbiorów zawiera obiekty powtarzające się w obu, natomiast suma zbiorów to wszystkie elementy znajdujące się w którymkolwiek z nich. Poniżej znajduje się nieco bardziej realistyczny przykład działań na zbiorach, zastosowany do list osób z hipotetycznej firmy i wykorzystujący literaly zbiorów z wersji 3.0 (w wersji 2.6 należy użyć konstruktora `set`).

```
>>> engineers = {'robert', 'amadeusz', 'anna', 'aleksander'}
>>> managers = {'edward', 'amadeusz'}

>>> 'robert' in engineers                                # Czy Robert jest inżynierem?
True

>>> engineers & managers                                # Kto jest inżynierem i menedżerem?
{'amadeusz'}

>>> engineers | managers                                # Wszystkie osoby z dowolnej kategorii
{'edward', 'amadeusz', 'anna', 'robert', 'aleksander'}

>>> engineers - managers                                # Inżynierowie niebędący menedżerami
{'robert', 'aleksander', 'anna'}

>>> managers - engineers                                # Menedżerowie niebędący inżynierami
{'edward'}

>>> engineers > managers                                # Czy wszyscy menedżerowie są inżynierami? (nadzbiór)
False

>>> {'robert', 'amadeusz'} < engineers                  # Czy obaj są inżynierami? (podzbiór)
True

>>> (managers | engineers) > managers                  # Nadzbiorem menedżerów są wszyscy ludzie
True

>>> managers ^ engineers                                # Kto jest w jednym zbiorze, ale nie obu?
{'robert', 'edward', 'aleksander', 'anna'}

>>> (managers | engineers) - (managers ^ engineers)    # Część wspólna!
{'amadeusz'}
```

Więcej informacji na temat działań na zbiorach można znaleźć w dokumentacji biblioteki Pythona, a także tekstach teoretycznych poświęconych matematyce i relacyjnym bazom danych. W rozdziale 8. powrócimy do niektórych omówionych tutaj operacji na zbiorach w kontekście obiektów widoku słowników z Pythona 3.0.

Wartości Boolean

Niektóre osoby uważają, że typ wartości Boolean Pythona (`bool`) jest z natury liczbowy, ponieważ jego dwie wartości — `True` i `False` — są tylko innymi wersjami liczb całkowitych 1 i 0, które jedynie zapisuje się w odmienny sposób. Choć większość programistów taka wiedza wystarczy, zajmijmy się tym typem przez chwilę.

Formalnie Python ma jawnego typ danych Boolean o nazwie `bool`, z wartościami `True` i `False` dostępnymi jako nowe wbudowane nazwy. Wewnętrznie te nazwy są instancjami `bool`; `bool` z kolei jest tylko podklassą (w znaczeniu zorientowania obiektowego) wbudowanego typu liczby całkowitej `int`. `True` i `False` zachowują się dokładnie tak samo jak liczby całkowite 1 i 0, jednak mają własną logikę wyświetlania — wyświetlane są jako słowa `True` i `False` w miejscu cyfr 1 i 0. Tak naprawdę `bool` redefiniuje formaty `str` i `repr` na potrzeby swoich dwóch obiektów.

Ze względu na tę zmianę dane wyjściowe z wyrażeń Boolean wpisanych w sesji interaktywnej wyświetlane są jako słowa `True` i `False` zamiast starszej, mniej oczywistej wersji 1 i 0. Dodatkowo typ ten sprawia, że wartości Boolean są bardziej oczywiste. Pętlę można na przykład teraz zapisać jako `while True`: zamiast mniej intuicyjnego `while 1`. W podobny, jaśniejszy sposób można także inicjalizować flagi za pomocą `flag = False`. Instrukcje tego typu omówimy jeszcze w trzeciej części książki.

Dla wszystkich innych celów praktycznych można traktować `True` i `False` tak, jakby były zdefiniowanymi zmiennymi o wartości 1 i 0. Większość programistów i tak przypisywała `True` i `False` do 1 i 0, więc nowy typ jedynie standaryzuje tę technikę. Jej implementacja może jednak prowadzić do dziwnych wyników — ponieważ `True` jest po prostu liczbą całkowitą 1 z innym formatem wyświetlania, `True + 4` zwraca w Pythonie wynik 5!

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1
True
>>> # Ta sama wartość
>>> True is 1
False
>>> # Ale inny obiekt — patrz kolejny rozdział
>>> True or False
True
>>> # To samo co: 1 or 0
>>> True + 4
5
>>> # (Hmmm)
```

Ponieważ mało prawdopodobne jest spotkanie w prawdziwym kodzie napisanym w Pythonie z wyrażeniem takiego jak ostatnie, można bezpiecznie zignorować jego głębokie implikacje metafizyczne...

Wartościami Boolean zajmiemy się ponownie w rozdziale 9. (przy okazji definiowania pojęcia prawdy w Pythonie), a także w rozdziale 12. (by sprawdzić, jak działają operatory Boolean `and` i `or`).

Dodatkowe rozszerzenia numeryczne

Oprócz własnych typów liczbowych Pythona istnieje również wiele różnych dodatków na licencji open source, odpowiadających na bardziej wyspecjalizowane potrzeby. Ponieważ programowanie numeryczne jest popularną dziedziną zastosowania Pythona, dostępnych jest wiele różnych zaawansowanych narzędzi.

Jeśli na przykład potrzebne nam jest wykonywanie poważniejszych obliczeń, opcjonalne rozszerzenie Pythona o nazwie *NumPy* (Numeric Python) udostępnia zaawansowane narzędzia programowania numerycznego, takie jak typ danych macierzy, przetwarzanie wektorów i wyszukiwanie biblioteki obliczeniowe. Poważne zespoły naukowe programistów w miejscowościach

takich, jak Los Alamos czy NASA wykorzystują Pythona w połączeniu z NumPy w implementacji zadań, które wcześniej wykonywali w językach C++, FORTRAN czy programie Matlab. Połączenie Pythona i NumPy jest często określane mianem darmowej i bardziej elastycznej wersji Matlaba — otrzymujemy wydajność NumPy i język Python wraz z jego bibliotekami.

Ponieważ NumPy jest rozszerzeniem tak bardzo zaawansowanym, nie będziemy go już omawiać w niniejszej książce. Dodatkowe wsparcie dla zaawansowanego programowania numerycznego w Pythonie, w tym narzędzia do tworzenia wykresów, biblioteki statystyczne, a także popularny pakiet *SciPy*, można znaleźć na stronie PyPI Pythona lub przeszukując Internet. Warto również podkreślić, że NumPy jest obecnie rozszerzeniem opcjonalnym. Nie jest częścią Pythona i trzeba je instalować osobno.

Podsumowanie rozdziału

Niniejszy rozdział zawierał przegląd typów obiektów liczbowych Pythona, a także operacji, jakie można do nich zastosować. Poznaliśmy standardowe typy liczb całkowitych i zmiennoprzecinkowych, a także typy bardziej egzotyczne i rzadziej używane, jak liczby zespolone, ułamki i zbiorы. Omówiliśmy również składnię wyrażeń Pythona, konwersję typów, operacje poziomu bitowego i różne literały służące do kodowania liczb w programach.

W dalszej części książki uzupełnimy szczegóły dotyczące kolejnego typu danych, czyli łańcucha znaków. W kolejnym rozdziale bardziej szczegółowo zajmiemy się jednak przypisywaniem zmiennych. Okazuje się to w Pythonie jedną z najbardziej podstawowych koncepcji, dlatego lektura kolejnego rozdziału będzie niezbędna do tego, by przejść dalej. Najpierw jednak pora na zwyczajowy quiz.

Sprawdź swoją wiedzę — quiz

1. Jaka w Pythonie jest wartość wyrażenia $2 * (3 + 4)$?
2. Jaka w Pythonie jest wartość wyrażenia $2 * 3 + 4$?
3. Jaka w Pythonie jest wartość wyrażenia $2 + 3 * 4$?
4. Jakich narzędzi można użyć do obliczenia pierwiastka kwadratowego liczby, a także jej kwadratu?
5. Jakiego typu będzie wynik wyrażenia $1 + 2.0 + 3$?
6. W jaki sposób można odciąć i zaokrąglić liczbę zmiennoprzecinkową?
7. W jaki sposób można przekonwertować liczbę całkowitą na liczbę zmiennoprzecinkową?
8. W jaki sposób można wyświetlić liczbę całkowitą w notacji szesnastkowej, ósemkowej czy dwójkowej?
9. W jaki sposób można przekonwertować łańcuch ósemkowy, szesnastkowy lub dwójkowy na zwykłą liczbę całkowitą?

Sprawdź swoją wiedzę — odpowiedzi

1. Wynikiem będzie 14, czyli $2 * 7$, gdyż nawiasy wymuszają wykonanie dodawania przed mnożeniem.
2. Teraz wynikiem będzie 10, czyli $6 + 4$. Gdy nie ma nawiasów, stosowane są reguły priorytetu operatorów Pythona, a mnożenie ma wyższy priorytet od dodawania (czyli wykonywane jest wcześniej) — zgodnie z tabelą 5.2.
3. Wyrażenie to zwraca 14, czyli $2 + 12$, zgodnie z tymi samymi regułami priorytetu co w poprzednim pytaniu.
4. Funkcje służące do uzyskania pierwiastka kwadratowego, a także na przykład `pi` czy `tangensa`, można znaleźć w importowanym module `math`. By obliczyć pierwiastek kwadratowy liczby, należy zaimportować moduł `math` i wywołać funkcję `math.sqrt(N)`. By otrzymać kwadrat liczby, należy albo wykorzystać wyrażenie potęgowania `X ** 2`, albo wbudowaną funkcję `pow(X, 2)`. Oba ostatnie rozwiązania mogą także obliczyć pierwiastek kwadratowy po podaniu potęgi 0.5 (na przykład `X ** .5`).
5. Wynik będzie liczbą zmiennoprzecinkową — liczby całkowite przekształcane są w góre do liczb zmiennoprzecinkowych, najbardziej skomplikowanego typu tego wyrażenia. Do obliczenia wykorzystana zostanie arytmetyka liczb zmiennoprzecinkowych.
6. Funkcje `int(N)` oraz `math.trunc(N)` odcinają część ułamkową, natomiast `round(N, liczba_cyfr)` zaokrąglą ją. Możemy także obliczyć zaokrąglenie do najbliższej mniejszej liczby całkowitej za pomocą `math.floor(N)` i zaokrąglić liczbę na cele jej wyświetlania za pomocą operacji formatowania łańcucha znaków.
7. Funkcja `float(I)` konwertuje liczbę całkowitą na zmiennoprzecinkową. Podobna konwersja będzie wynikiem połączenia w jednym wyrażeniu liczb obu tych typów. W pewnym sensie dzielenie / z Pythonem 3.0 także wykonuje konwersję — zwraca liczbę zmiennoprzecinkową obejmującą resztę, nawet jeśli oba argumenty były liczbami całkowitymi.
8. Wbudowane funkcje `oct(I)` i `hex(I)` zwracają łańcuch ósemkowy i szesnastkowy dla podanej liczby całkowitej. Wbudowana funkcja `bin(I)` zwraca z kolei łańcuch cyfr dwójkowych w Pythonie 2.6 oraz 3.0. To samo można osiągnąć za pomocą wyrażenia z formatującym łańcuchem znaków % oraz metody łańcuchów znaków `format`.
9. Funkcję `int(S, podstawa)` można wykorzystać do przekształcenia łańcucha ósemkowego lub szesnastkowego na normalną liczbę całkowitą (jako podstawę należy podać 8, 16 lub 2). Do tego samego celu można również wykorzystać funkcję `eval(S)`, jednak jej wykonanie jest bardziej kosztowne i może się wiązać z problemami z bezpieczeństwem. Warto zauważyc, że liczby całkowite są w pamięci komputera zawsze przechowywane w postaci binarnej; pozostałe formy to tylko konwersje formatu wyświetlania.

Wprowadzenie do typów dynamicznych

W poprzednim rozdziale rozpoczęliśmy pogłębione omawianie podstawowych typów obiektów Pythona, zaczynając od liczb. Omówienie to będziemy kontynuować w kolejnym rozdziale, jednak zanim przejdziemy dalej, ważne jest, by zrozumieć coś, co może być najważniejszą koncepcją w programowaniu w Pythonie, a z pewnością jest podstawą zarówno zwiążności, jak i elastyczności tego języka — typów dynamicznych i związanego z nimi polimorfizmu.

Jak zobaczymy zarówno tutaj, jak i w dalszej części książki, w Pythonie nie deklaruje się typu obiektów wykorzystywanych przez skrypty. Programy nie powinny nigdy przejmować się określonymi typami. W zamian za to mogą one być stosowane w większej liczbie kontekstów, niż czasami sami planujemy. Ponieważ podstawą tej elastyczności są typy dynamiczne, przyjrzymy się bliżej temu modelowi.

Sprawa brakujących deklaracji typu

Osoby znające języki komplilowane lub z typami statycznymi, takie jak C, C++ czy Java, mogą w tym miejscu książki czuć się nieco zagubione. Dotychczas wykorzystywaliśmy zmienne bez deklarowania istnienia ich czy typów i w jakiś sposób wszystko działało. Kiedy w sesji interaktywnej czy pliku programu wpiszemy na przykład `a = 3`, w jaki sposób Python będzie wiedział, że `a` to liczba całkowita? A skoro już przy tym jesteśmy, skąd Python w ogóle wie, czym jest `a`?

Kiedy zaczynamy sobie zadawać takie pytania, wkraczamy do modelu *typów dynamicznych* Pythona. W Pythonie typy ustalone są automatycznie w momencie wykonywania, a nie w odpowiedzi na deklarację w kodzie. Oznacza to, że nigdy nie deklarujemy zmiennych wcześniej (konsepcję tę łatwiej można pojąć, kiedy się pamięta, że wszystko sprowadza się do zmiennych, obiektów i połączeń między nimi).

Zmienne, obiekty i referencje

Jak widzieliśmy w wielu przykładach przedstawionych w książce, kiedy w Pythonie wykonyuje się instrukcję przypisania, na przykład `a = 3`, działa ona mimo tego, że nigdy Pythonowi nie nakazaliśmy użyć `a` jako zmiennej ani też nie określiliśmy, że `a` ma być obiektem liczby całkowitej. W języku tym wszystko to układa się w sposób bardzo naturalny.

Tworzenie zmiennej

Zmienna (czyli nazwa) — taka jak `a` — tworzona jest, kiedy kod pierwszy raz przypisuje jej wartość. Przyszłe przypisania zmieniają wartość utworzonej wcześniej zmiennej. Z technicznego punktu widzenia wygląda to tak, że Python wykrywa niektóre nazwy zmiennych przed wykonaniem kodu, ale możemy to sobie wyobrazić tak, jakby początkowe przypisanie tworzyło zmenną.

Typ zmiennej

Zmienna nigdy nie ma dołączonych żadnych informacji o typie czy ograniczeniach z nim związanych. Pojęcie typu wiąże się z obiektami, a nie nazwami. Zmienne są z natury uniwersalne. Zawsze w danym momencie odnoszą się po prostu do określonego obiektu.

Użycie zmiennej

Kiedy zmienność pojawiła się w wyrażeniu, jest natychmiast zastępowana przez obiekt, do którego się aktualnie odnosi, bez względu na to, czym by on nie był. Co więcej, wszystkie zmienne muszą mieć jawnie przypisane wartości, zanim będzie można ich użyć. Próba odwołania się do nieprzypisanej zmiennej kończy się błędem.

Podsumowując, zmienne tworzone są przy przypisaniu, mogą się odwoływać do dowolnego typu obiektu i muszą być przypisane przed wykonaniem referencji do nich. Oznacza to, że nigdy nie trzeba deklarować zmiennych wykorzystywanych w skrypcie, jednak przed ich uaktualnieniem niezbędna jest ich inicjalizacja. Przykładowo liczniki trzeba najpierw zainicjalizować z wartością 0, by móc do nich dodawać.

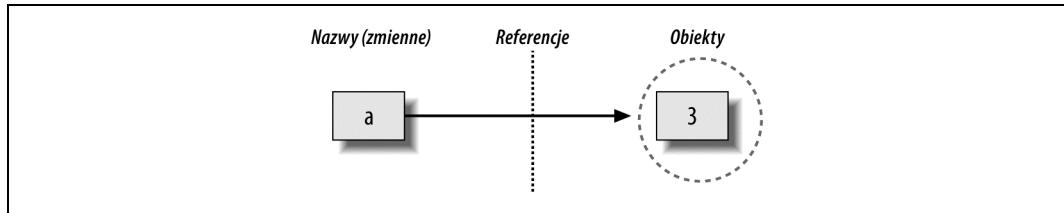
Model typów dynamicznych bardzo się różni od modelu typów w tradycyjnych językach programowania. Kiedy zaczyna się przygoda z Pythonem, najłatwiej jest zrozumieć typy dynamiczne, gdy jasno rozdziela się od siebie nazwy (zmienne) i obiekty. Kiedy na przykład napiszemy taki kod:

```
>>> a = 3
```

przynajmniej z konceptualnego punktu widzenia Python wykona trzy osobne kroki, by wykonać to żądanie. Kroki te odzwierciedlają operację przypisania w tym języku:

1. Utworzenie obiektu reprezentującego wartość 3.
2. Utworzenie zmiennej `a`, o ile jeszcze nie istnieje.
3. Połączenie zmiennej `a` z nowym obiektem 3.

Rezultatem będzie struktura wewnętrz Pythona przypominająca rysunek 6.1. Jak widać na szkicu, zmienne i obiekty przechowywane są w różnych częściach pamięci i powiązane ze sobą połączeniami (takie połączenie widoczne jest na rysunku jako strzałka). Zmienne zawsze łączą się z obiektami i nigdy z innymi zmiennymi, jednak większe obiekty mogą łączyć się z innymi obiektami (jak na przykład obiekt listy łączący się z obiektami, które zawiera).



Rysunek 6.1. Zmienne i obiekty po wykonaniu przypisania `a = 3`. Zmienna `a` staje się referencją do obiektu `3`. Wewnętrznie zmienne są tak naprawdę wskaźnikiem do miejsca w pamięci zajmowanego przez obiekt i utworzonego przez wykonanie wyrażenia literatu `3`

Te połączenia pomiędzy zmiennymi a obiektami nazywane są w Pythonie *referencjami*. Referencja to rodzaj powiązania zaimplementowanego jako wskaźnik w pamięci.¹ Za każdym razem, gdy zmienne są używane (to znaczy korzysta się z referencji), Python automatycznie podąży połączeniem między zmienną a obiektem. Wszystko to jest o wiele prostsze, niż może wynikać z terminologii. A mówiąc konkretnie:

- *Zmienne* są wpisami w tabeli systemowej z miejscem na łącze do obiektów.
- *Obiekty* to fragmenty przydzielonej pamięci z ilością miejsca wystarczającą, by zmieścić wartości, które reprezentują.
- *Referencje* to wskaźniki między zmiennymi a obiektami, którymi automatycznie podąża Python.

Przynajmniej z koncepcjonalnego punktu widzenia za każdym razem, gdy w skrypcie generowana jest nowa wartość poprzez wykonanie jakiegoś wyrażenia, Python tworzy nowy obiekt (czyli fragment pamięci) reprezentujący tę wartość. Wewnętrznie, ze względu na optymalizacje, Python umieszcza pewne rodzaje niezmiennych obiektów w pamięci podręcznej i używa ich ponownie; jest tak na przykład w przypadku małych liczb całkowitych orazłańcuchów znaków (nie każde `0` jest tak naprawdę nowym fragmentem pamięci — więcej na ten temat później). Jednak z logicznego punktu widzenia działa to tak, jakby wartość wynikowa każdego wyrażenia była osobnym obiektem, a każdy obiekt był osobną częścią pamięci.

Z technicznego punktu widzenia obiekty mają nieco bardziej rozbudowaną strukturę niż tylko „tyle miejsca, by zmieścić swoją wartość”. Każdy obiekt ma również dwa standardowe pola nagłówków — *desygnator typu* wykorzystywany jest do oznaczenia typu obiektu, natomiast *licznik referencji* służy do ustalenia, kiedy obiekt można uwolnić. By zrozumieć, jakie znaczenie mają te dwa dodatkowe pola nagłówków, należy przejść dalej.

Typy powiązane są z obiektami, a nie ze zmiennymi

By zobaczyć, w którym miejscu pojawiają się typy, warto prześledzić, co się stanie, kiedy zmienną przypiszemy kilka razy.

¹ Osoby znające język C mogą skojarzyć referencje w Pythonie ze wskaźnikami (adresami w pamięci) języka C. Tak naprawdę referencje zaimplementowane są jako wskaźniki i często pełnią te same role, w szczególności w przypadku obiektów, które mogą być modyfikowane w miejscu (więcej na ten temat później). Ponieważ jednak referencje są zawsze automatycznie usuwane po użyciu, nie można z nimi samymi zrobić nic przydatnego — jest to cecha, która eliminuje dużą kategorię błędów występujących w języku C. Referencje w Pythonie można sobie wyobrazić jako wskaźniki typu „void **” z języka C, które przy użyciu są automatycznie śledzone.

```
>>> a = 3                                # Jest liczbą całkowitą  
>>> a = 'mielonka'                         # Teraz jestłańcuchem znaków  
>>> a = 1.23                               # A teraz liczbą zmiennoprzecinkową
```

Nie jest to kod typowy dla Pythona, ale działa — a na początku jest liczbą całkowitą, później staje się łańcuchem znaków, a na koniec staje się liczbą zmiennoprzecinkową. Ten przykład może wyglądać szczególnie dziwacznie dla programistów języka C, ponieważ wygląda na to, jakby *typ zmiennej a zmieniał się z liczby całkowitej na łańcuch znaków, kiedy wykona się instrukcję a = 'mielonka'*.

Tak naprawdę dzieje się jednak coś innego. W Pythonie wszystko jest jeszcze prostsze. Nazwy (zmienne) nie mają typów; jak wspomniano wcześniej, typy powiązane są z obiektami, a nie z nazwami. W poprzednim listingu zmodyfikowaliśmy tylko zmienną a, tak by odnosiła się ona do innych obiektów. Ponieważ zmienne nie mają typu, nie zmodyfikowaliśmy tak naprawdę typu zmiennej a — zmieniona ta odnosi się po prostu teraz do innego typu obiektu. Tak naprawdę jedynie, co można powiedzieć o zmiennych w Pythonie, to to, że odnoszą się do określonego obiektu w określonym momencie.

Obiekty wiedzą z kolei, jakiego są typu, ponieważ każdy obiekt zawiera pole nagłówka oznaaczające obiekt jego typem. Obiekt liczby całkowitej 3 zawiera na przykład wartość 3 i oprócz tego desygnator typu informujący Pythona, że obiekt ten jest liczbą całkowitą (a tak naprawdę wskaźnik do obiektu `int`, nazwy typu liczby całkowitej). Desygnator typu obiektu łańcucha znaków '`mielonka`' wskazuje z kolei na typ łańcucha znaków o nazwie `str`. Ponieważ obiekty znajdują swoje typy, zmienne nie muszą ich znać.

Powtarzając raz jeszcze: typy są w Pythonie powiązane z obiektami, a nie ze zmiennymi. W typowym kodzie dana zmienna będzie zazwyczaj zawierała referencję do jednego rodzaju obiektu. Ponieważ nie jest to jednak wymagane, wkrótce okaże się, że kod napisany w Pythonie zazwyczaj jest o wiele bardziej elastyczny od tego, do czego możemy być przyzwyczajeni. Jeśli będziemy dobrze wykorzystywać Pythona, kod przez nas tworzony może automatycznie działać na wielu typach obiektów.

Wspomniałem wyżej, że obiekty zawierają dwa pola nagłówków — desygnator typu i licznik referencji. By zrozumieć znaczenie tego drugiego pola, musimy przyjrzeć się temu, co dzieje się na końcu życia obiektu.

Obiekty są uwalniane

W kodzie wyżej przypisaliśmy zmienną a do trzech różnych typów obiektów. Co się jednak po kolejnym przypisaniu dzieje z wartością, do której odnosił się obiekt? Co na przykład stanie się z obiektem 3 po wykonaniu poniższej instrukcji?

```
>>> a = 3  
>>> a = 'mielonka'
```

Odpowiedź będzie następująca: w Pythonie za każdym razem, gdy zmienna przypisywana jest do nowego obiektu, miejsce zajmowane przez poprzedni obiekt jest uwalniane (o ile nie odnosi się do niego żadna inna nazwa lub obiekt). Takie automatyczne zwalnianie przestrzeni obiektu nazywane jest *czyszczeniem pamięci* (ang. *garbage collection*).

By zilustrować ten mechanizm, rozważmy poniższy przykład, który przyporządkowuje zmiennej x inny obiekt przy każdej instrukcji przypisania.

```
>>> x = 42
>>> x = ' żywopłot'
>>> x = 3.1415
>>> x = [1, 2, 3]
# Uwołnienie liczby 42 (o ile nie ma innych referencji)
# Uwołnienie łańcucha znaków ' żywopłot'
# Uwołnienie liczby 3.1415
```

Po pierwsze, należy zauważyc, że zmienna `x` za każdym razem odnosi się do innego typu obiektu. I znowu, choć tak naprawdę wcale tak nie jest, można odnieść wrażenie, jakby typ zmiennej `x` z czasem się zmieniał. Należy pamiętać, że w Pythonie typy powiązane są jednak z obiektami, a nie z nazwami. Ponieważ zmienne są uniwersalnymi referencjami do obiektów, taki kod działa w naturalny sposób.

Po drugie, warto zwrócić uwagę na to, że referencje do obiektów są po drodze porzucane. Za każdym razem, gdy do zmiennej `x` przypisujemy nowy obiekt, Python zwalnia miejsce zajmowane przez poprzedni. Kiedy na przykład do `x` przypisujemy łańcuchów znaków ' `żywopłot`', obiekt 42 jest natychmiast zwalniany (zakładając, że nie istnieje żadna inna referencja do niego). Tym samym miejsce zajmowane przez ten obiekt jest automatycznie wrzucane z powrotem do puli wolnego miejsca, które może być użyte przez kolejny obiekt.

Wewnętrznie Python wykonuje to wszystko, przechowując w każdym obiekcie licznik, który śledzi liczbę referencji wskazujących na ten obiekt. W momencie gdy licznik spadnie do zera, miejsce zajmowane przez obiekt w pamięci jest automatycznie zwalniane. W poprzednim liście zakładamy, że za każdym razem, gdy do zmiennej `x` przypisujemy nowy obiekt, licznik referencji poprzedniego obiektu spada do zera, co sprawia, że jest on zwalniany.

Najbardziej zauważalną zaletą czyszczenia pamięci jest to, że można swobodnie korzystać z obiektów bez konieczności troszczenia się o zwalnianie miejsca w pamięci w skrypcie. Python czyści nieużywaną przestrzeń za nas w miarę wykonywania programu. W praktyce eliminuje to dużą ilość kodu śledzącego i utrzymującego w porównaniu z językami niższego poziomu, takimi jak C czy C++.



Z technicznego punktu widzenia mechanizm czyszczenia pamięci w Pythonie oparty jest przede wszystkim na *licznikach referencji*, zgodnie z powyższym opisem. Zawiera jednak również komponent wykrywający i zwalniający obiekty z *referencjami cyklicznymi*. Komponent ten można wyłączyć, jeśli jesteśmy pewni, że nasz kod nie tworzy cykli, jednak domyślnie jest on włączony.

Ponieważ referencje zaimplementowane są w postaci wskaźników, obiekt może odwoływać się do samego siebie bądź też do obiektu, który zawiera do niego referencję. Ćwiczenie 3. znajdujące się na końcu pierwszej części książki oraz jego rozwiązanie z dodatku B pokazują na przykład, jak można utworzyć cykl, zagnieżdżając referencję do listy wewnętrznej niej samej. To samo zjawisko może wystąpić w przypadku przypisów do atrybutów obiektów tworzonych z klas definiowanych przez użytkownika. Choć jest to niezwykle rzadkie, sytuacje takie muszą być traktowane w specjalny sposób, ponieważ liczniki referencji dla takich obiektów nigdy nie spadną do zera.

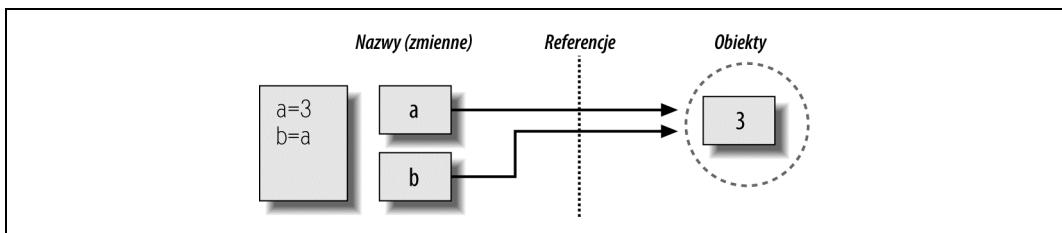
Szczegółowe informacje na temat wykrywacza cykli w Pythonie można znaleźć w dokumentacji modułu `gc` dostępnej w podręczniku biblioteki standardowej Pythona. Warto także zauważyc, że niniejszy opis mechanizmu czyszczenia pamięci w Pythonie odnosi się jedynie do standardowej implementacji CPython. Jython i IronPython mogą korzystać z innych mechanizmów, choć rezultat ich działania będzie podobny — niewykorzystywane miejsce jest automatycznie zwalniane.

Referencje współdzielone

Dotychczas widzieliśmy, co dzieje się, kiedy do jednej zmiennej przypisujemy referencje do obiektów. Teraz wprowadzimy do naszego kodu kolejną zmienną i sprawdzimy, co stanie się z nazwami i obiektemi.

```
>>> a = 3  
>>> b = a
```

Wpisanie tych dwóch instrukcji generuje sytuację ujętą na rysunku 6.2. Drugi wiersz sprawia, że Python tworzy zmienną `b`. Zmienna `a` jest tutaj użyta, ale nie przypisana, dlatego zostaje zastąpiona obiektem, do którego się odnosi (3), natomiast `b` zawiera odniesienie do tego obiektu. Rezultat jest taki, że zmienne `a` i `b` odnoszą się do tego samego obiektu (czyli wskazują na ten sam fragment pamięci). Jest to w Pythonie znane pod nazwą *referencji współdzielonych* (ang. *shared references*) i oznacza, że wiele zmiennych odnosi się do tego samego obiektu.



Rysunek 6.2. Nazwy (zmienne) i obiekty po przypisaniu `b = a`. Zmienna `b` staje się referencją do obiektu 3. Wewnętrznie zmienna jest tak naprawdę wskaźnikiem do miejsca w pamięci zajmowanego przez obiekt i utworzonego przez wykonanie wyrażenia literatu 3

Następnie założymy, że rozszerzymy ten kod interaktywny o kolejną instrukcję.

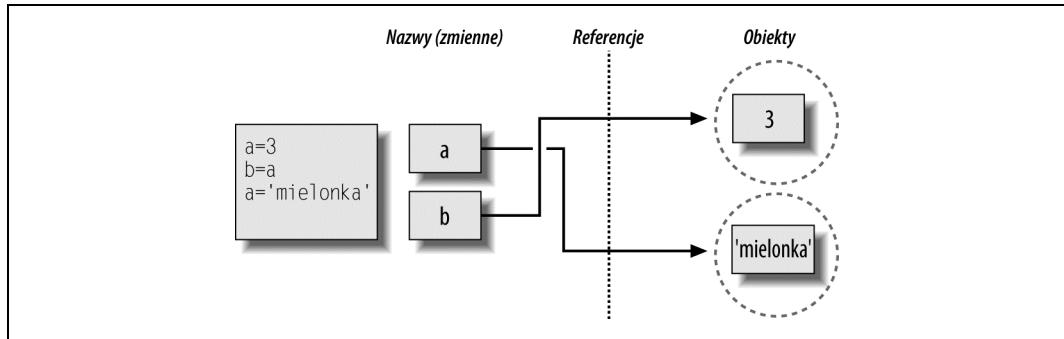
```
>>> a = 3  
>>> b = a  
>>> a = 'mielonka'
```

Tak jak w przypadku wszystkich przypisań w Pythonie, instrukcja ta tworzy nowy obiekt reprezentujący wartość 'mielonka' i ustawia zmienną `a` w taki sposób, by odnosiła się do tego nowego obiektu. Nie zmienia jednak wartości zmiennej `b` — nadal odnosi się ona do oryginalnego obiektu, czyli liczby całkowitej 3. Struktura referencji będzie teraz zatem taka, jak na rysunku 6.3.

To samo stałoby się, gdybyśmy zamiast tego zmienili `b` na 'mielonka' — przypisanie zmieniłoby jedynie zmienną `b`, a nie `a`. Takie zachowanie pojawia się również wtedy, gdy nie ma żadnej różnicy w zakresie typów. Rozważmy na przykład trzy poniższe instrukcje.

```
>>> a = 3  
>>> b = a  
>>> a = a + 2
```

W tej sekwencji przypisań dzieje się to samo. Python przypisuje do zmiennej `a` obiekt 3, następnie `b` staje się referencją do tego samego obiektu co `a` (jak na rysunku 6.2). Jak wcześniej, ostatnie przypisanie zmienia `a` na zupełnie inny obiekt (w tym przypadku liczbę całkowitą 5, będącą wynikiem wyrażenia z operatorem `+`). Nie modyfikuje to jednak zmiennej `b`. Tak naprawdę nie da się nadpisać wartości obiektu 3 — tak jak pisaliśmy w rozdziale 4., liczby całkowite są niezmienne i tym samym nie można ich modyfikować w miejscu.



Rysunek 6.3. Nazwy (zmiennne) i obiekty po przypisaniu $a = \text{'mielonka'}$. Po wykonaniu wyrażenia z literałem 'mielonka' zmienna a staje się referencją do nowego obiektu (miejscu w pamięci), natomiast zmienna b nadal odwołuje się do oryginalnego obiektu — liczby 3. Ponieważ to przypisanie nie jest modyfikacją obiektu 3 w miejscu, modyfikuje ono jedynie zmienną a , a nie b

Można to sobie wyobrazić w taki sposób, że w przeciwieństwie do niektórych języków programowania w Pythonie zmienne zawsze są wskaźnikami do obiektów, a nie podpisami zmieniających się miejsc w pamięci. Nadanie zmiennej nowej wartości nie modyfikuje oryginalnych obiektów, ale raczej sprawia, że zmienna odnosi się do całkowicie innego obiektu. Rezultat jest taki, że przypisanie do zmiennej może wpływać jedynie na tę jedną zmienną, do której coś się przypisuje. Kiedy do równania dodamy również obiekty zmienne i zmiany w miejscu, obraz ten nieco się zmieni. Żeby to zobaczyć, przejdźmy dalej.

Referencje współdzielone a modyfikacje w miejscu

Jak zobaczymy w kolejnych rozdziałach tej części książki, w Pythonie istnieją obiekty i operacje modyfikujące obiekt w miejscu. Przypisanie do pozycji przesunięcia w liście zmienia na przykład listę w miejscu, a nie generuje nowego obiektu listy. W przypadku obiektów obsługujących zmiany tego rodzaju należy bardziej uważać na referencje współdzielone, ponieważ zmiana dokonana w jednym miejscu może wpływać na inne obiekty.

By zilustrować tę kwestię, przyjrzyjmy się raz jeszcze obiektom listy wprowadzonym w rozdziale 4. Warto przypomnieć, że listy obsługujące przypisanie w miejscu do określonych pozycji są po prostu kolekcjami innych obiektów, zakodowanymi w nawiasach kwadratowych.

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

L_1 to lista zawierająca obiekty 2, 3 i 4. Dostęp do obiektów znajdujących się wewnętrznej tej listy można uzyskać za pomocą ich pozycji, dzięki czemu $L1[0]$ odnosi się do obiektu 2, pierwszego elementu listy L_1 . Listy są oczywiście obiektem same w sobie, podobnie jak liczby całkowite i łańcuchy znaków. Po wykonaniu dwóch powyższych instrukcji przypisania L_1 i L_2 odnoszą się do tego samego obiektu, podobnie jak a i b z poprzedniego przykładu (rysunek 6.2). Powiedzmy, że teraz znów, tak jak poprzednio, rozszerzymy ten kod o kolejne przypisanie:

```
>>> L1 = 24
```

Przypisanie to sprawia, że L_1 będzie teraz referencją do innego obiektu, natomiast L_2 nadal jest referencją do oryginalnej listy. Gdybyśmy nieco zmienili składnię tej instrukcji, miałaby ona zupełnie inny efekt.

```

>>> L1 = [2, 3, 4]          # Zmienny obiekt
>>> L2 = L1                # Referencja do tego samego obiektu
>>> L1[0] = 24              # Zmiana w miejscu

>>> L1                      # Lista L1 zmienia się
[24, 3, 4]
>>> L2                      # Tak samo lista L2!
[24, 3, 4]

```

Tym razem nie zmodyfikowaliśmy tak naprawdę samej listy `L1`, a jedynie komponent *obiektu*, do którego odnosi się `L1`. Ten rodzaj zmiany powoduje nadpisanie części obiektu listy w miejscu. Ponieważ obiekt listy jest współdzielony przez różne zmienne (które się do niego odnoszą), modyfikacja tego typu nie ma zastosowania jedynie do `L1`. Trzeba pamiętać, że zmiany tego typu mogą mieć wpływ na inne części naszego programu. W powyższym przykładzie widać to w `L2`, gdyż zmienna ta odnosi się do tego samego obiektu co `L1`. I choć nie modyfikowaliśmy także `L2`, wartość tej zmiennej jest już inna, ponieważ została ona nadpisana.

Takie zachowanie jest zazwyczaj czymś pożądanym, choć należy być go świadomym, tak by zmiany można było przewidzieć. Jest to zachowanie domyślne. Jeśli jednak nie jest przez nas pożądane, można założyć, by Python *kopiował* obiekty, zamiast robić do nich referencje. Istnieje kilka sposobów skopiowania listy, w tym wykorzystanie wbudowanej funkcji `list`, a także modułu `copy` z biblioteki standardowej. Chyba najczęściej stosowaną techniką jest sporządzenie wycinka z całej listy (więcej informacji na temat wycinków znajduje się w rozdziałach 4. oraz 7.).

```

>>> L1 = [2, 3, 4]
>>> L2 = L1[:]                  # Sporządzenie kopii L1
>>> L1[0] = 24

>>> L1
[24, 3, 4]
>>> L2                      # L2 się nie zmienia
[2, 3, 4]

```

W powyższym kodzie zmiany wprowadzone do `L1` nie są przenoszone do `L2`, ponieważ zmienna `L2` odnosi się do kopii obiektu, do którego referencją jest `L1`. Oznacza to, że obie zmienne odnoszą się do dwóch różnych fragmentów pamięci.

Warto podkreślić, że technika z wycinkami nie będzie działała na innych zmiennych typach podstawowych, czyli słownikach oraz zbiorach, ponieważ nie są one sekwencją. W celu skopiowania słownika lub zbioru należy użyć wywołania ich metody `X.copy()`. Moduł `copy` z biblioteki standardowej zawiera wywołanie służące do uniwersalnego skopiowania obiektu dowolnego typu, a także wywołanie kopiujące zagnieżdzoną strukturę obiektów (na przykład słownik z zagnieżdzonymi listami).

```

import copy
X = copy.copy(Y)                      # Wykonanie "pływkiej" kopii dowolnego obiektu Y
X = copy.deepcopy(Y)                  # Wykonanie "głębokiej" kopii dowolnego obiektu Y
                                         # — skopiowanie wszystkich elementów zagnieżdzonych

```

Listy i słowniki, a także koncepcję współdzielonych referencji oraz kopii omówimy bardziej szczegółowo w rozdziałach 8. i 9. Na razie należy pamiętać, że obiekty zmienne (takie, które można modyfikować w miejscu) są podatne na tego rodzaju efekty. W Pythonie dotyczy to list, słowników i pewnych obiektów definiowanych za pomocą instrukcji `class`. Jeśli nie jest to pożądane zachowanie, można obiekt w miarę potrzeby skopiować.

Referencje współdzielone a równość

By wszystko było jasne, warto wspomnieć, że czyszczenie pamięci opisane wcześniej może dla niektórych typów obiektów być bardziej koncepcyjne niż dosłowne. Rozważmy poniższe instrukcje:

```
>>> x = 42  
>>> x = 'żywopłot'  
# Uwolnić teraz obiekt 42?
```

Ponieważ Python umieszcza małe liczby całkowite oraz łańcuchy znaków w pamięci podręcznej, by użyć ich ponownie, obiekt 42 prawdopodobnie nie zostanie faktycznie uwolniony. Zamiast tego pozostałe raczej w tabeli systemowej, tak by można go było wykorzystać następnym razem, gdy w kodzie wygenerujemy liczbę 42. Większość typów obiektów jest jednak uwalniana od razu po tym, jak znika ostatnia referencja do nich. W przypadku tych, które nie są uwalniane, mechanizm umieszczania w pamięci podręcznej nie ma znaczenia dla naszego kodu.

Przykładowo ze względu na model referencji Pythona istnieją dwa różne sposoby sprawdzania równości w programach napisanych w tym języku. By to zademonstrować, utworzymy wspólną dzieloną referencję.

```
>>> L = [1, 2, 3]  
>>> M = L  
>>> L == M  
True  
>>> L is M  
True  
# M i L odnoszą się do jednego obiektu  
# Ta sama wartość  
# Ten sam obiekt
```

Pierwsza z technik zaprezentowanych powyżej, operator `==`, sprawdza, czy dwa obiekty, do których odnoszą się zmienne, mają tę samą wartość. Metoda ta jest prawie zawsze wykorzystywana w sprawdzaniu równości w Pythonie. Druga metoda, operator `is`, sprawdza zamiast tego identyczność obiektów — zwraca `True` tylko wtedy, gdy obie zmienne odnoszą się do dokładnie tego samego obiektu, dlatego jest o wiele mocniejszą formą sprawdzania równości.

Tak naprawdę operator `is` sprawdza po prostu wskaźniki implementujące referencje, dzięki czemu jest w stanie w razie potrzeby wykrywać w kodzie współdzielone referencje. Zwraca `False`, jeśli zmienne odnoszą się do odpowiadających sobie, jednak różnych obiektów, tak jak wtedy, gdy wykonamy dla nich różne wyrażenia z literałami.

```
>>> L = [1, 2, 3]  
>>> M = [1, 2, 3]  
>>> L == M  
True  
>>> L is M  
False  
# M i L odnoszą się do różnych obiektów  
# Te same wartości  
# Różne obiekty
```

Sprawdźmy teraz, co stanie się, gdy te same operacje wykonamy na małych liczbach.

```
>>> X = 42  
>>> Y = 42  
>>> X == Y  
True  
>>> X is Y  
True  
# Powinny być dwoma różnymi obiektami  
# A jednak to jeden obiekt (pamięć podręczna działa)!
```

W powyższym kodzie `X` i `Y` powinny mieć tę samą wartość (operator `==`), jednak nie powinny być tym samym obiektem (operator `is`), ponieważ wykonaliśmy dwa osobne wyrażenia z literałami. Ponieważ jednak małe liczby i łańcuchy znaków umieszczane są w pamięci podręcznej i używane ponownie, okazuje się, że obie zmienne odnoszą się do tego samego obiektu.

Gdybyśmy naprawdę chcieli przyjrzeć się temu, jak to działa, zawsze możemy zapytać Pythona, ile jest referencji do określonego obiektu. Funkcja `getrefcount` z modułu standardowego `sys` zwraca licznik referencji obiektu. Kiedy na przykład w IDLE zapytam o obiekt liczby całkowitej 1, Python zwraca wynik 837 (choć większość referencji pochodzi z kodu systemowego IDLE, a nie mojego własnego).

```
>>> import sys  
>>> sys.getrefcount(1) # 837 wskaźników do tego miejsca w pamięci  
837
```

Umieszczanie obiektów w pamięci podręcznej i ponowne ich użycie nie ma znaczenia dla naszego kodu (o ile oczywiście nie sprawdzamy równości za pomocą `is`). Ponieważ liczb i łańcuchów znaków nie można modyfikować w miejscu, nie ma znaczenia, ile jest referencji do tego samego obiektu. Zachowanie to odzwierciedla jednak jeden z wielu sposobów optymalizacji modelu Pythona pod kątem szybkości wykonania.

Typy dynamiczne są wszędzie

By używać Pythona, nie trzeba tak naprawdę rysować diagramów z nazwami zmiennych, obiektami i strzałkami. Na początek jednak prześledzenie struktury referencji pomaga często zrozumieć niezwykłe przypadki. Jeśli na przykład zmienny obiekt zostanie zmodyfikowany w czasie przekazywania w programie, jest duża szansa, że trafiliśmy na zagadnienia omówione w niniejszym rozdziale.

Co więcej, nawet jeśli typy dynamiczne wydają się w tej chwili nieco abstrakcyjne, z pewnością wkrótce uda się nam przyswoić tę koncepcję. Ponieważ w Pythonie *wszystko* zdaje się działać przez przypisanie i referencje, podstawowe zrozumienie tego modelu przydaje się w wielu różnych kontekstach. Jak się niedługo okaże, tak samo działa to w instrukcjach przypisania, argumentach funkcji, zmiennych pętli `for`, importowaniu modułów, atrybutach klas i innych. Dobra wiadomość jest taka, że w Pythonie istnieje tylko jeden model przypisania. Kiedy zrozumiemy typy dynamiczne, zauważymy, że w całym języku działają one w ten sam sposób.

Z praktycznego punktu widzenia typy dynamiczne oznaczają mniej kodu, który musimy napisać. Co jednakowo ważne, typy dynamiczne są również podstawą *polimorfizmu* Pythona, czyli koncepcji przedstawionej w rozdziale 4., do której powrócimy w dalszej części książki. Ponieważ typy nie są w kodzie Pythona ograniczone, kod ten jest bardzo elastyczny. Jak zobaczymy niebawem, kiedy się je dobrze wykorzystuje, typy dynamiczne i polimorfizm tworzą kod, który wraz z rozwojem systemu automatycznie dostosowuje się do nowych wymagań.

Podsumowanie rozdziału

W niniejszym rozdziale szerzej przyjrzaliśmy się modelowi typów dynamicznych Pythona, czyli sposobowi, w jaki Python w automatyczny sposób przechowuje dla nas informacje o typach obiektów, zamiast wymagać od nas umieszczania w skryptach instrukcji z deklaracjami. Przy okazji dowiedzieliśmy się, w jaki sposób zmienne i obiekty są w Pythonie powiązane referencjami. Omówiliśmy również kwestię czyszczenia pamięci, dowiedzieliśmy się, jak współdzielone referencje do obiektów mogą wpływać na wiele zmiennych, a także jaki wpływ mają na zagadnienie równości w Pythonie.

Ponieważ w Pythonie istnieje tylko jeden model przypisania, a także ponieważ przypisania pojawiają się w tym języku wszędzie, zrozumienie tego modelu przed przejściem do dalszego omawiania języka jest bardzo istotne. Quiz znajdujący się na końcu rozdziału pomoże usystematyzować niektóre z przedstawionych tutaj koncepcji. Potem, w kolejnym rozdziale, powrócimy do omawiania kolejnych typów obiektów — tym razem łańcuchów znaków.

Sprawdź swoją wiedzę — quiz

1. Rozważmy trzy poniższe instrukcje. Czy zmieniają one wartość A?

```
A = "mielonka"  
B = A  
B = " żywopłot"
```

2. Rozważmy trzy poniższe instrukcje. Czy zmieniają one wartość A?

```
A = ["mielonka"]  
B = A  
B[0] = " żywopłot"
```

3. Czy tym razem A się zmieni?

```
A = ["mielonka"]  
B = A[:]  
B[0] = " żywopłot"
```

Sprawdź swoją wiedzę — odpowiedzi

1. Nie. A nadal ma wartość "mielonka". Kiedy B przypiszemy do łańcucha znaków " żywopłot ", jedynie, co się dzieje, to modyfikacja zmiennej B, tak by wskazywała ona teraz na nowy obiekt łańcucha znaków. A i B początkowo współdzielą (to znaczy odnoszą się, wskazują) ten sam obiekt łańcucha znaków "mielonka", jednak obie zmienne nigdy nie są ze sobą w Pythonie połączone. Ustawienie B na inny obiekt nie ma zatem wpływu na A. Tak samo byłoby, gdyby ostatnią instrukcją było `B = B + ' żywopłot '` — konkatenacja utworzyłaby z wyniku nowy obiekt, który zostałby następnie przypisany jedynie do B. Nigdy nie możemy nadpisać łańcucha znaków (albo liczby czy krotki) w miejscu, ponieważ typy te są niezmienne.

2. Tak. A ma teraz wartość [" żywopłot "]. Z technicznego punktu widzenia nie zmieniliśmy ani A, ani B. Zamiast tego zmodyfikowaliśmy część obiektu, do którego odnoszą się obie zmienne, nadpisując ten obiekt w miejscu poprzez zmienną B. Ponieważ A odnosi się do tego samego obiektu co B, zmiana ta odzwierciedlana jest również w A.

3. Nie. A nadal ma wartość ["mielonka"]. Przypisanie w miejscu za pomocą B nie ma tym razem żadnego wpływu na A, ponieważ wyrażenie z wycinkiem sporządziło kopię obiektu listy przed przypisaniem go do B. Po drugiej instrukcji przypisania istnieją dwa różne obiekty list o tej samej wartości (w Pythonie mówi się, że są one `==`, ale nie `is`). Trzecia instrukcja zmienia wartość obiektu listy, na który wskazuje zmienna B, jednak już nie tego, na który wskazuje A.

Łańcuchy znaków

Kolejnym podstawowym typem, jakim zajmiemy się przy okazji omawiania obiektów wbudowanych, jest *łańcuch znaków* (ang. *string*) — uporządkowana kolekcja znaków wykorzystywana do przechowywania i reprezentowania informacji tekstowych. Z łańcuchami znaków spotkaliśmy się już w rozdziale 4. Teraz omówimy je bardziej szczegółowo, uzupełniając pewne informacje, które wcześniej pominęliśmy.

Z funkcjonalnego punktu widzenia łańcuchy znaków można wykorzystać do reprezentowania wszystkiego, co można zakodować w postaci tekstowej — symboli i słów (na przykład naszego imienia), zawartości załadowanych do pamięci plików tekstowych, adresów internetowych czy programów napisanych w Pythonie. Można ich również użyć do przechowywania binarnych wartości oraz wielobajtowych tekstów Unicode.

Łańcuchy znaków występują również w innych językach programowania. W Pythonie pełnią one tę samą rolę co tablice znaków z języków takich, jak C, jednak w pewnym sensie są narzędziami wyższego poziomu. W przeciwieństwie do języka takiego, jak C, Python nie ma specjalnego typu dla pojedynczego znaku (jak `char` z C). Zamiast tego tworzy się łańcuchy jednoznakowe.

Ścisłe mówiąc, łańcuchy znaków Pythona zaliczane są do niezmiennych sekwencji, co oznacza, że znaki przez nie zawierane ustawione są w określonym porządku od lewej strony do prawej, a samych łańcuchów nie można modyfikować w miejscu. Tak naprawdę łańcuchy znaków są pierwszym reprezentantem większej klasy obiektów zwanych *sekwencjami*. Warto zwrócić szczególną uwagę na omawiane w niniejszym rozdziale operacje na sekwencjach, ponieważ będą one działały w ten sam sposób na innych typach sekwencji, którymi zajmiemy się później — na przykład na listach czy krotkach.

W tabeli 7.1 znajduje się przegląd popularnych literałów i operacji na łańcuchach znaków, które zostaną przedstawione w niniejszym rozdziale. Puste łańcuchy znaków zapisuje się jako parę cudzysłówów lub apostrofów bez żadnych znaków między nimi. Istnieje wiele sposobów kodowania łańcuchów znaków. Jeśli chodzi o przetwarzanie, łańcuchy znaków obsługują *wyrażenia*, takie jak konkatenacja (łączenie łańcuchów), wycinki (ekstrakcja części łańcuchów), indeksowanie (pobieranie elementów zgodnie z wartością przesunięcia) i wiele innych. Poza wyrażeniami Python udostępnia również zbiór *metod* łańcuchów znaków, które implementują często spotykane zadania związane z tym typem danych, a także *moduły* służące do bardziej zaawansowanego przetwarzania tekstu, w tym na przykład dopasowywania wzorców. Wszystkie te operacje omówimy w dalszej części rozdziału.

Tabela 7.1. Popularne literały i operacje na łańcuchach znaków

Operacja	Interpretacja
S = ''	Pusty łańcuch znaków
S = "mielonka o nazwie 'Mielonka'"	Cudzysłowy
S = 'm\ni\te\x00lonka'	Sekwencje znaków specjalnych
S = """..."""	Blok w potrójnych cudzysłach
S = r"\temp\mielonka"	Surowy łańcuch znaków
S = b'mielonka'	Bitowe łańcuchy znaków w 3.0 (rozdział 36.)
S = u'mielonka'	łańcuch znaków Unicode tylko w 2.6 (rozdział 36.)
S1 + S2	Konkatenacja, powtórzenie
S * 3	
S[i]	Indeksowanie, wycinek, długość
S[i:j]	
len(S)	
"mała %s papuga" % kind	Wyrażenie formatujące łańcuch znaków
"a {0} papuga".format(kind)	Formatowanie łańcuchów znaków w 2.6 i 3.0
S.find('ie')	Wywołania metod łańcuchów znaków — wyszukiwania,
S.rstrip()	usuwania białych znaków,
S.replace('ie', 'xx')	zastępowania,
S.split(',')	dzielenia w miejscu wystąpienia ogranicznika,
S.isdigit()	sprawdzania zawartości,
S.lower()	konwersji wielkości liter,
S.endswith('mielonka')	sprawdzenia końcówki łańcucha znaków,
'mielonka'.join(strlist)	złączenia z użyciem separatora,
S.encode('latin-1')	kodowania Unicode
for x in S: print(x)	Iteracja, przynależność
'mielonka' in S	
[c * 2 for c in S]	
map(ord, S)	

Poza podstawowym zestawem narzędzi przeznaczonych do pracy z łańcuchami znaków (tabela 7.1) Python obsługuje również bardziej zaawansowane przetwarzanie łańcuchów znaków w oparciu o dopasowanie wzorców dzięki wprowadzonemu w rozdziale 4. modułowi `re` (od ang. *regular expressions* — wyrażenia regularne) z biblioteki standardowej, a nawet wysoko poziomowe narzędzia, jak parsery XML, omówione pokróć w rozdziale 36. W tej książce zajmiemy się jednak zagadnieniami podstawowymi, wymienionymi w tabeli 7.1.

Na początku rozdziału znajduje się omówienie podstaw, czyli przegląd form literałów i wyrażeń tekstowych, następnie przejdziemy do bardziej zaawansowanych narzędzi, jak metody łańcuchów znaków i formatowanie. Python oferuje wiele narzędzi służących do pracy z łańcuchami znaków. Nie będziemy tu omawiać wszystkich z nich, kompletną dokumentację można znaleźć w podręczniku biblioteki standardowej Pythona. Naszym celem jest omówienie najpowszechniej stosowanych narzędzi i pokazanie reprezentatywnych przykładów. Metody, których nie zademonstrujemy, mają w większości zastosowanie analogiczne do tych omówionych.



Uwaga na temat treści: Z technicznego punktu widzenia niniejszy rozdział prezentuje jedynie część informacji o łańcuchach znaków w Pythonie — część, którą powinna znać większość programistów. Omawiamy podstawowy typ `str`, obsługujący tekst w kodowaniu ASCII i działający tak samo niezależnie od zastosowanej wersji Pythona. Innymi słowy, niniejszy rozdział celowo ogranicza zakres informacji do tego, który jest wykorzystywany w większości programów w Pythonie.

Z formalnej perspektywy ASCII stanowi prostą formę tekstu Unicode. Python rozróżnia tekst i dane binarne, stosując do ich obsługi różne typy obiektów.

- W Pythonie 3.0 istnieją trzy typy znakowe: `str` jest stosowany do zapisu tekstów Unicode (w kodowaniu ASCII lub innym), `bytes` jest stosowany do zapisu danych binarnych (w tym tekstu zakodowanego), natomiast `bytearray` jest mutowalnym wariantem typu `bytes`.
- W Pythonie 2.6 do przetwarzania tekstów Unicode stosowany jest typ `unicode`, natomiast `str` jest stosowany do zapisu tekstów w kodowaniach 8-bitowych oraz do danych binarnych. Łańcuchy obsługują zarówno kodowanie 8-bitowe, jak i binarne.

Typ `bytearray` jest również dostępny w 2.6, ale nie w wersjach wcześniejszych i nie jest tak blisko związany z danymi binarnymi jak w 3.0. Większość programistów nie potrzebuje zagłębiać się w szczegóły związane z kodowaniami Unicode oraz formatami danych binarnych, dlatego omówienie tych zagadnień odłożyłem do rozdziału 36.

Jeśli ktoś potrzebuje wykorzystać bardziej zaawansowane narzędzia tekstowe, jak kodowania znaków czy dane i pliki binarne, zachęcam do przeczytania rozdziału 36. — po przeanalizowaniu niniejszego rozdziału. Jednak na razie skupimy się na podstawowych typach znakowych i operacjach na nich. Jak będziemy mieli okazję się przekonać, podstawy poznane w niniejszym rozdziale przekładają się bezpośrednio na bardziej zaawansowane techniki oferowane przez Pythona.

Literały łańcuchów znaków

Generalnie łańcuchy znaków są w Pythonie łatwe w użyciu. Chyba najbardziej skomplikowanym ich aspektem jest to, że istnieje tyle sposobów zapisania ich w kodzie.

- apostrofy — `'mie"lonka'`,
- cudzysłowy — `"mie'lonka"`,
- potrójne apostrofy lub cudzysłowy — `'''...mielonka...'''`, `"""...mielonka..."""`,
- sekwencje ucieczki — `"mi\tel\no\Onka"`,
- surowe łańcuchy znaków — `r"C:\nowe\test.spm"`,
- bajtowe łańcuchy znaków w 3.0 (rozdział 36.) — `b'miel\x01onka'`,
- łańcuchy znaków Unicode w 2.6 (rozdział 36.) — `u'jajka\u0020mielonka'`.

Najczęściej spotykane są bez wątpienia formy z apostrofami i cudzysłowami. Pozostałe mają wyspecjalizowane role, a omówienie ostatnich dwóch odłożymy do rozdziału 36. Przyjrzyjmy się krótko każdej z powyższych opcji.

Łańcuchy znaków w apostrofach i cudzysłowach są tym samym

W przypadku łańcuchów znaków Pythona znaki apostrofów i cudzysłowów mogą być stosowane wymiennie. Oznacza to, że literaly łańcuchowe można zapisać zarówno w dwóch cudzysłowach, jak i dwóch apostrofach — obie te formy działają w ten sam sposób i zwracają ten sam typ obiektu. Dwa poniższe łańcuchy znaków będą na przykład identyczne:

```
>>> ' żywopłot ', " żywopłot "
(' żywopłot ', ' żywopłot ')
```

Powodem używania dwóch form jest możliwość osadzenia wewnątrz łańcucha znaków znaku cudzysłowa (w łańcuchu ograniczonym apostrofami) i odwrotnie bez konieczności stosowania sekwencji ucieczki ze znakiem ukośnika lewego. Apostrof można osadzić w łańcuchu znaków w cudzysłowach, natomiast cudzysłów — w łańcuchu w apostrofach.

```
>>> 'ryce"rz', "ryce'rz"
('ryce"rz', "ryce'rz")
```

Tak się składa, że Python automatycznie dokonuje konkatenacji przylegających do siebie literałów łańcuchowych w każdym wyrażeniu, choć równie proste jest jawne wywołanie konkatenacji poprzez wstawienie między te łańcuchy operatora + (jak zobaczymy w rozdziale 12., ujęcie tego wyrażenia w nawiasy pozwala zapisać je w kilku wierszach).

```
>>> title = "Żywot " 'Briana'                      # Niejawna konkatenacja
>>> title
'Żywot Briana'
```

Warto zauważyć, że wstawienie między łańcuchy znaków przecinków zwróciłoby krotkę, a nie łańcuch znaków. Widać również, że we wszystkich danych wyjściowych Python woli wyświetlać łańcuchy znaków w apostrofach, o ile same nie zawierają one takiego znaku. Apostrofy i cudzysłowy można również umieszczać w łańcuchach znaków z sekwencjami ucieczki.

```
>>> 'ryce\'rz', "ryce\"rz"
("ryce'rz", 'ryce"rz')
```

By zrozumieć ich działanie, warto dowiedzieć się, czym w ogóle są sekwencje ucieczki.

Sekwencje ucieczki reprezentują bajty specjalne

W ostatnim przykładzie cudzysłów i apostrof osadzono w łańcuchu znaków, poprzedzając go znakiem ukośnika lewego (\). To ogólny wzorzec stosowany w łańcuchach znaków — ukośniki lewe wykorzystywane są do wprowadzenia specjalnego kodowania bajtów zwanego *sekwencją ucieczki* (ang. *escape sequence*).

Sekwencje ucieczki pozwalają osadzać w łańcuchach znaków kody bajtów, których nie da się łatwo wpisać na klawiaturze. Znak \ i jeden lub większa liczba następujących po nim znaków w literale łańcuchowym w wynikowym obiekcie łańcucha znaków zastępowane są pojedynczym znakiem o wartości binarnej odpowiadającej sekwencji ucieczki. Poniżej widać na przykład łańcuch pięcioznakowy, w którym osadzony jest nowy wiersz i tabulator.

```
>>> s = 'a\nb\tc'
```

Dwa znaki \n zastępują pojedynczy znak — bajt zawierający wartość binarną znaku nowego wiersza w określonym kodowaniu (zazwyczaj kod 10 z ASCII). W podobny sposób sekwencja

\t zastępowana jest pojedynczym znakiem tabulatora. Wygląd wyświetlonego łańcucha znaków zależy od sposobu jego wyświetlenia. W sesji interaktywnej znaki specjalne zwracane są w postaci z sekwencjami ucieczki, natomiast instrukcja print pozwala je zinterpretować.

```
>>> s
'a\nb\tc'
>>> print(s)
a
b      c
```

By upewnić się, ile bajtów naprawdę zawiera łańcuch znaków, należy skorzystać z wbudowanej funkcji len. Zwraca ona prawdziwą liczbę bajtów w łańcuchu znaków bez względu na sposób wyświetlenia tego łańcucha.

```
>>> len(s)
5
```

Łańcuch ten składa się z pięciu bajtów. Zawiera bajt ASCII a, bajt nowego wiersza, bajt ASCII b i tak dalej. Warto zauważyć, że oryginalne znaki ukośników lewych nie są przechowywane w pamięci wraz z łańcuchami znaków, jedynie sygnalizują Pythonowi, że następujące po nich znaki mają traktować w specjalny sposób. W celu zakodowania tych bajtów specjalnych Python wykorzystuje pełny zbiór sekwencji kodów ucieczki, zaprezentowany w tabeli 7.2.

Tabela 7.2. Sekwencje ucieczki łańcuchów znaków

Sekwencja ucieczki	Znaczenie
\newline	Ignorowany (kontynuacja)
\\"	Ukośnik lewy (zachowuje znak \)
\'	Apostrof (zachowuje znak ')
\"	Cudzysłów (zachowuje znak ")
\a	Sygnal dźwiękowy
\b	Znak cofania (ang. <i>backspace</i>)
\f	Wysuw strony (ang. <i>form feed</i>)
\n	Nowy wiersz (wysuw wiersza, ang. <i>line feed</i>)
\r	Powrót karetki
\t	Tabulator poziomy
\v	Tabulator pionowy
\xhh	Znak o wartości szesnastkowej hh (najwyżej dwie cyfry)
\ooo	Znak o wartości ósemkowej ooo (najwyżej trzy cyfry)
\0	Null — binarny zapis znaku o kodzie 0 (nie sygnalizuje końca łańcucha znaków)
\N{ id }	Identyfikator bazy danych Unicode
\uhhhh	Szesnastobitowa liczba określająca znak Unicode
\Uhhhhhhhh	Trzydziestodwubitowa liczba określająca znak Unicode ¹
\inny	Nie jest sekwencją ucieczki (przechowuje znak \ oraz znak następujący po nim)

¹ Sekwencja ucieczki \Uhhhh... przyjmuje dokładnie osiem cyfr szesnastkowych (h). Zarówno \u, jak i \U można używać jedynie w literałach łańcuchów znaków Unicode.

Niektóre sekwencje ucieczki pozwalają na osadzanie bezwzględnych wartości binarnych w bajtach łańcucha znaków. Poniżej znajduje się na przykład łańcuch pięcioznakowy, w którym osadzono dwa binarne bajty zerowe (zakodowane jako jednocyfrowe, ósemkowe kody znaków).

```
>>> s = 'a\0b\0c'  
>>> s  
'a\x00b\x00c'  
>>> len(s)  
5
```

W Pythonie bajt zerowy (`null`) nie kończy łańcucha znaków w taki sposób, jak dzieje się to w języku C. Zamiast tego Python zachowuje zarówno długość łańcucha znaków, jak i jego tekst w pamięci. Tak naprawdę w Pythonie żaden znak nie kończy łańcucha znaków. Poniżej znajduje się łańcuch znaków składający się z samych bezwzględnych kodów ucieczki — binarnych 1 i 2 (zakodowanych w postaci ósemkowej), po których znajduje się binarne 3 (zakodowane w postaci szesnastkowej).

```
>>> s = '\001\002\x03'  
>>> s  
\x01\x02\x03'  
>>> len(s)  
3
```

Python wyświetla niedrukowalne znaki w postaci kodów ósemkowych, niezależnie od tego, w jaki sposób zostały zadeklarowane. W literałach można mieszać kody ósemkowe z sekwencjami wykorzystującymi odwrotne ukośniki, prezentowanymi w tabeli 7.2. Poniższy listing zawiera łańcuch znaków mielonka z umieszczonymi znakami tabulacji i przejścia do nowego wiersza oraz binarnym znakiem o kodzie zero, przekazanym w formie kodu ósemkowego.

```
>>> S = "m\ti\n\x00lonka"  
>>> S  
'm\ti\n\x00lonka'  
>>> len(S)  
11  
>>> print(S)  
m           i  
e lonka
```

Ma to dużo większe znaczenie, kiedy w Pythonie przetwarza się binarne pliki z danymi. Ponieważ ich zawartość w naszych skryptach reprezentowana jest jako łańcuchy znaków, możliwe jest przetwarzanie plików binarnych zawierających dowolne rodzaje binarnych wartości bajtów (więcej informacji na temat plików znajduje się w rozdziale 9.).²

Wreszcie, zgodnie z informacjami z ostatniego wpisu z tabeli 7.2, jeśli Python nie rozpozna znaku umieszczonego po \ jako poprawnego kodu ucieczki, po prostu zachowuje znak ukośnika w wynikowym łańcuchu znaków.

² Dla osób szczególnie zainteresowanych binarnymi plikami z danymi: ich najważniejszą cechą wyróżniającą jest to, że otwiera się je w trybie binarnym (wykorzystując opcje trybu otwarcia z literą b, takie jak 'rb' czy 'wb'). W Pythonie 3.0 pliki binarne są odczytywane z użyciem typu bytes, z wykorzystaniem interfejsu przypominającego zwykłe ciągi znaków, w 2.6 tego typu dane są odczytywane z użyciem zwykłych ciągów znaków. W rozdziale 9. znajduje się omówienie modułu struct biblioteki standardowej, który może być użyty do parsowania danych binarnych wczytywanych z plików. W rozdziale 36. znajduje się rozbudowany opis obsługi plików binarnych i typu bytes.

```
>>> x = "C:\\py\\code"                                # Zachowuje dosłowny znak \
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Osoby, które nie są w stanie zapamiętać całej tabeli 7.2, nie powinny raczej polegać na tej ostatniej kwestii.³ Aby kodować dosłowne znaki ukośników lewych, tak by były one zachowywane w łańcuchach znaków, należy je podwoić (\\" jest sekwencją ucieczki dla \) lub korzystać z surowych łańcuchów znaków opisanych w kolejnym podrozdziale.

Surowe łańcuchy znaków blokują sekwencje ucieczki

Jak widzieliśmy, sekwencje ucieczki przydają się do osadzania specjalnych kodów bajtów wewnętrz łańcuchów znaków. Czasami jednak specjalne traktowanie ukośników lewych służących jako wprowadzenie do sekwencji ucieczki może powodować problemy. Zaskakująco często spotyka się na przykład osoby początkujące, które próbują otworzyć plik, podając jego nazwę jako argument w następującej postaci:

```
myfile = open('C:\\nowy\\tekst.dat', 'w')
```

i myślą, że w ten sposób uda im się otworzyć plik o nazwie *tekst.dat* umieszczony w katalogu *C:\\nowy*. Problemem jest jednak to, że \n oznacza znak nowego wiersza, a \t zastępowane jest przez tabulator. W rezultacie wywołanie to próbuje otworzyć plik o nazwie *C:(nowy wiersz)owy(tabulator)ekst.dat*, czego rezultat nie może być satysfakcyjny.

W takiej właśnie sytuacji przydają się surowe łańcuchy znaków. Jeśli litera r (mała lub wielka) pojawia się tuż przed cudzysłowem lub apostrofem otwierającym łańcuch znaków, wyłącza ona mechanizm sekwencji ucieczki. W rezultacie Python zachowuje dosłowne znaki ukośników dokładnie tak, jak je wpisaliśmy. Z tego powodu, by naprawić problem z nazwą pliku, wystarczy pamiętać o dodaniu w systemie Windows litery r.

```
myfile = open(r'C:\\nowy\\tekst.dat', 'w')
```

Alternatywnie, ponieważ dwa znaki ukośnika lewego są sekwencją ucieczki dla pojedynczego znaku tego rodzaju, ukośniki można zachować, podwajając je.

```
myfile = open('C:\\\\nowy\\\\tekst.dat', 'w')
```

Tak naprawdę sam Python czasami wykorzystuje schemat z podwajaniem, gdy wyświetla łańcuchy znaków zawierające ukośniki lewe.

```
>>> path = r'C:\\nowy\\tekst.dat'                  # Wyświetl jako kod Pythona
>>> path
'C:\\nowy\\tekst.dat'
>>> print(path)                                    # Format przyjazny dla użytkownika
C:\\nowy\\tekst.dat
>>> len(path)                                     # Długość łańcucha znaków
17
```

Tak jak w przypadku reprezentacji liczbowej, format domyślny w sesji interaktywnej wyświetla wyniki w taki sposób, jakby były one kodem, i tym samym w danych wyjściowych zawiera

³ W czasie prowadzonych przeze mnie kursów spotkałem się z osobami, które faktycznie nauczyły się całej tabeli na pamięć. Normalnie pomyślałbym, że to chore, gdyby nie fakt, że sam należę do tego grona.

sekwencje ucieczki dla ukośników lewych. Instrukcja `print` udostępnia format bardziej przyjazny dla użytkowników, pokazujący, że tak naprawdę w każdym miejscu znajduje się tylko jeden znak ukośnika. By to zweryfikować, możemy sprawdzić wynik wbudowanej funkcji `len`, która zwraca liczbę bajtów łańcucha znaków niezależnie od formatu wyświetlania. Jeśli policzymy znaki w zmiennej `path` zwracanej przez instrukcję `print`, okaże się, że każdy ukośnik jest jednym znakiem, co razem daje 17 znaków dla całego łańcucha.

Poza ścieżkami do katalogów w systemie Windows surowe łańcuchy znaków wykorzystywane są często w wyrażeniach regularnych (dopasowywaniu wzorców obsługiwany przez moduł `re` przedstawiony w rozdziale 4.). Warto również zauważyc, że skrypty Pythona mogą zazwyczaj w ścieżkach do katalogów w systemach Windows i Unix wykorzystywać także znaki ukośników prawych, ponieważ Python próbuje interpretować ścieżki w sposób przenośny (to znaczy w Windows zadziała ścieżka '`C:/new/text.dat`'). Surowe łańcuchy znaków przydają się jednak, kiedy ścieżki do kodu wykorzystują własne ukośniki lewe systemu Windows.



Nawet surowy łańcuch znaków nie może się kończyć lewym ukośnikiem, ponieważ ten pojedynczy ukośnik zostanie potraktowany jako znak specjalny zmieniający znaczenie znaku cudzysłówu domykającego dany literał, co powoduje błąd składni. Innymi słowy, łańcuch znaków `r"..."`` nie jest prawidłowym literałem. Jeśli ktoś potrzebuje zakończyć surowy łańcuch znakiem ukośnika, powinien użyć dwóch ukośników, a następnie zastosować operator wycinania, pozbywając się zbędnego znaku (`r'1\nb\tc\\'[:-1]`), albo dokleić ten znak (`r'1\nb\tc' + '\\'`), ewentualnie zrezygnować z surowych łańcuchów znaków i użyć zwykłego łańcucha znaków ze zdublowanymi znakami lewego ukośnika (`'1\\nb\\tc\\'`). Wszystkie trzy powyższe techniki dają w efekcie ten sam, ośmioznakowy łańcuch znaków zawierający trzy lewe ukośniki.

Potrójne cudzysłowy i apostrofy kodują łańcuchy znaków będące wielowierszowymi blokami

Jak na razie zapoznaliśmy się z apostofami, cudzysłowami, sekwencjami ucieczki i surowymi łańcuchami znaków. Python posiada również format literala łańcucha znaków z potrójnymi apostrofami lub cudzysłowami, czasem nazywany *blokowym łańcuchem znaków*. Jest on składowym ułatwieniem służącym do zapisu wielowierszowych danych tekstowych. Format ten rozpoczyna się od trzech apostrofów lub cudzysłówów, potem następuje dowolna liczba wierszy tekstu, a na końcu sekwencja trzech apostrofów lub cudzysłówów — w zależności od tego, jaka sekwencja otwierała łańcuch. Cudzysłowy i apostrofy osadzone w tekście łańcucha znaków mogą być zastępowane sekwencjami ucieczki, ale nie muszą — łańcuch ten nie kończy się, dopóki Python nie napotka sekwencji trzech znaków, które wykorzystane zostały do otwarcia łańcucha.

```
>>> mantra = """Zawsze patrz
...   na życie
...   z humorem."""
>>>
>>> mantra
'Zawsze patrz\n na życie\nz humorem.'
```

Ten łańcuch znaków rozciąga się na trzy wiersze (w niektórych interfejsach interaktywny znak zachęty zmienia się na `...` w wierszach będących kontynuacją; IDLE po prostu przechodzi do wiersza niżej). Python zbiera cały tekst mieszący się w potrójnych cudzysłowach lub apostro-

fach do jednego, wielowierszowego łańcucha znaków ze znakami nowego wiersza (\n) osadzonymi w miejscach, gdzie w kodzie znajduje się złamanie wiersza. Warto również zauważać, że w drugim wierszu literał na początku znajduje się spacja, jednak w trzecim wierszu już jej nie ma — otrzymujemy dokładnie to, co wpisaliśmy. Aby zobaczyć nasz łańcuch znaków z właściwie zinterpretowanymi znakami końca wierszy, należy wywołać funkcję print, zamiast zwracać go na konsolę.

```
>>> print(mantra)
Zawsze patrz
na życie
z humorem.
```

Łańcuchy znaków tego rodzaju przydają się zawsze wtedy, gdy w programie potrzebny jest nam tekst wielowierszowy — na przykład, kiedy chcemy w plikach z kodem źródłowym osadzić wielowierszowe komunikaty o błędzie lub kod HTML czy XML. Takie bloki można osadzać bezpośrednio w skryptach bez konieczności korzystania z zewnętrznych plików tekstowych czy połączenia konkatenacji i znaków nowych wierszy.

Łańcuchy znaków z potrójnymi cudzysłowami lub apostrofami są również często wykorzystywane w łańcuchach dokumentacji, czyli literałach łańcuchowych traktowanych jako komentarze, kiedy pojawiają się one w określonych punktach naszego pliku (więcej informacji na ten temat później). Nie muszą one być blokami z potrójnymi cudzysłowami czy apostrofami, ale najczęściej tak jest, by można w nich było umieszczać komentarze wielowierszowe.

Wreszcie łańcuchy znaków tego typu są często wykorzystywane jako paskudny sposób tymczasowego wyłączania wierszy kodu w trakcie programowania (no dobrze, nie tak całkiem paskudny i na dodatek będący często spotykaną praktyką). Jeśli chcemy wyłączyć kilka wierszy kodu i ponownie wykonać skrypt, wystarczy przed tym kodem i po nim wstawić trzy apostrofy lub cudzysłowy.

```
X = 1
"""
import os                               # Tymczasowo blokujemy wykonanie tego kodu
print(os.getcwd())
"""

Y = 2
```

Nazwałem ten sposób paskudnym, ponieważ Python naprawdę tworzy łańcuch znaków z wierszy wyłączonych w ten sposób; nie ma to jednak większego wpływu na wydajność programu. W przypadku większych fragmentów kodu jest to również łatwiejsze wyjście od ręcznego wstawiania znaków # na początku każdego wiersza, a później usuwania ich. Jest to szczególnie przydatne, kiedy używa się edytora tekstu, który nie obsługuje edycji kodu napisanego w Pythonie w specjalny sposób. W Pythonie praktyczność często wygrywa z estetyką.

Łańcuchy znaków w akcji

Po utworzeniu łańcuchów znaków za pomocą omówionych wyrażeń z literałami z pewnością będziemy chcieli coś z nimi zrobić. W tym podrozdziale i kolejnych dwóch zademonstrujemy wyrażenia, metody i formatowanie łańcuchów znaków — najważniejsze narzędzia służące w Pythonie do przetwarzania tekstu.

Podstawowe operacje

Rozpoczniemy od zaangażowania interpretera Pythona w sesji interaktywnej w celu zaprezentowania podstawowych operacji na łańcuchach znaków wymienionych w tabeli 7.1. Łańcuchy znaków można ze sobą łączyć za pomocą operatora konkatenacji `+`, a także powtarzać za pomocą operatora `*`.

```
% python
>>> len('abc')                                # Długość (len), czyli liczba elementów
3
>>> 'abc' + 'def'                            # Konkatenacja — nowy łańcuch znaków
'abcdef'
>>> 'Ni!' * 4                                 # Powtórzenie — jak "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Z formalnego punktu widzenia dodanie do siebie dwóch obiektów łańcuchów znaków tworzy nowy obiekt łańcucha znaków, który łączy zawartość obu argumentów. Powtórzenie przypomina dodanie łańcucha znaków do samego siebie kilka razy. W obu przypadkach Python pozwala na utworzenie łańcuchów znaków o dowolnym rozmiarze. W Pythonie nie trzeba niczego wcześniej deklarować — dotyczy to również wielkości struktur danych.⁴ Wbudowana funkcja `len` zwraca długość łańcucha znaków (i każdego innego obiektu mającego jakąś długość).

Powtórzenie może się na początku wydawać nieco dziwne, ale przydaje się w zaskakującym duży liczbie kontekstów. Żeby na przykład wyświetlić linię składającą się z osiemdziesięciu myślników, można albo odliczyć osiemdziesiąt takich znaków, albo pozwolić, by to Python je za nas policzył.

```
>>> print('----- ...więcej... ---')      # 80 myślników, gorszy sposób
>>> print('-'*80)                           # 80 myślników, lepszy sposób
```

Warto zauważyć, że działa tutaj przeciążanie operatorów — te same operatory `+` i `*` wykonują na liczbach dodawanie i mnożenie. Python wykonuje te operacje poprawnie, ponieważ zna typy dodawanych i mnożonych obiektów. Należy jednak pamiętać — reguły nie są aż tak liberalne, jak można by oczekiwać. Python nie pozwala na przykład łączyć liczb i łańcuchów znaków w wyrażeniu z operatorem `+` i wyrażenie `'abc' + 9` zwróci błąd, zamiast automatycznie przekonwertować liczbę `9` na łańcuch znaków.

Jak widać w ostatnim wierszu tabeli 7.1, można również wykonywać iterację po łańcuchach znaków za pomocą pętli (wykorzystując instrukcję `for`), a także sprawdzać przynależność znaków lub podłańcuchów za pomocą wyrażenia z operatorem `in` (co jest tak naprawdę formą wyszukiwania). W przypadku podłańcuchów instrukcja `in` działa podobnie do metody `str.find()` omówionej w dalszej części niniejszego rozdziału, ale zwraca wartość boolowską zamiast pozycji podłańcucha w źródłowym łańcuchu znaków.

⁴ W przeciwnieństwie do tablic znaków języka C, używając łańcuchów znaków Pythona, nie musimy alokować tablic przechowywania czy nimi zarządzać. Wystarczy utworzyć pożądane obiekty i pozwolić Pythonowi na zarządzanie miejscem w pamięci. Zgodnie z informacjami z rozdziału 6, Python automatycznie zwalnia miejsce zajmowane przez nieużywany obiekt w pamięci, wykorzystując do tego technikę czyszczenia pamięci opartą na licznikach referencji. Każdy obiekt przechowuje informacje o liczbie zmiennych czy struktur danych do niego się odnoszących. Kiedy licznik ten osiągnie zero, Python zwalnia miejsce zajmowane przez ten obiekt. Taki schemat oznacza, że Python nie musi się zatrzymywać i przeszukiwać całej pamięci w celu znalezienia nieużywanego miejsca do zwolnienia. Dodatkowy komponent czyszczący pamięć zbiera również obiekty cykliczne.

```

>>> myjob = "haker"
>>> for c in myjob: print(c, end=' ')      # Przejście przez elementy
...
h a k e r
>>> "k" in myjob                         # Znaleziono
True
>>> "z" in myjob                         # Nie znaleziono
False
>>> 'jajko' in 'abcdejajkoxyz'           # Zawieranie podłańcucha znaków, pozycja nie jest zwracana
True

```

Pętla `for` przypisuje zmienną do kolejnych elementów sekwencji (tutaj: łańcucha znaków) i wykonyuje dla każdego z nich instrukcję lub większą ich liczbę. W rezultacie zmienna `c` staje się tutaj kursem przechodzącym łańcuch znaków. Narzędzia iteracyjne o których wspomina tabela 7.1 omówimy bardziej szczegółowo w dalszej części książki (zwłaszcza w rozdziałach 14. i 20.).

Indeksowanie i wycinki

Ponieważ łańcuchy znaków definiowane są jako uporządkowane kolekcje znaków, dostęp do ich komponentów można uzyskać za pomocą ich pozycji. W Pythonie każdy znak z łańcucha można pobrać za pomocą *indeksowania* — podania w nawiasach kwadratowych po łańcuchu znaków liczbowej wartości przesunięcia pożądanego elementu. Otrzymamy w ten sposób łańcuch zawierający jeden znak znajdujący się na podanej pozycji.

Tak jak w języku C, wartości przesunięcia rozpoczynają się w Pythonie od 0 i kończą się na wielkości o jeden mniejszej od długości łańcucha. W przeciwieństwie do tego języka, Python pozwala jednak pobierać elementy z sekwencji również za pomocą *ujemnych* wartości przesunięcia. Z technicznego punktu widzenia ujemna wartość przesunięcia dodawana jest do długości łańcucha znaków w celu uzyskania pożądanej wartości dodatniej. Ujemne wartości przesunięcia można sobie również wyobrazić jak odliczanie od końca. Widać to w przykładzie poniżej.

```

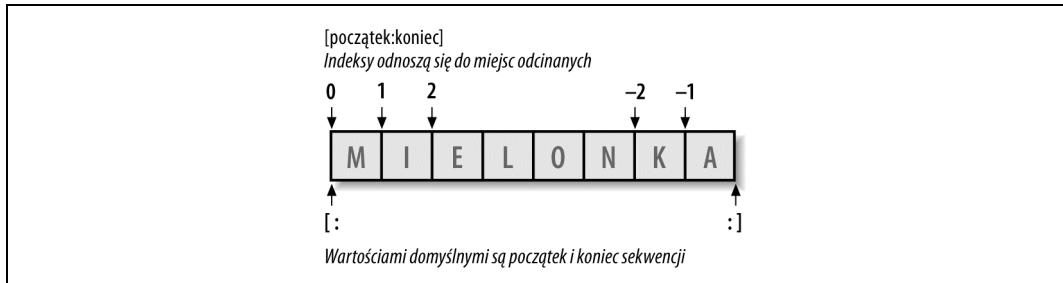
>>> S = 'mielonka'
>>> S[0], S[-2]                           # Indeksowanie od początku lub od końca
('m', 'k')
>>> S[1:3], S[1:], S[:-1]                # Wycinek — ekstrakcja części łańcucha
('ie', 'ielonka', 'mielonk')

```

Pierwszy wiersz definiuje łańcuch składający się z ośmiu znaków i przypisuje do niego zmienną `S`. W kolejnym wierszu łańcuch ten zostaje zindeksowany na dwa sposoby. `S[0]` pobiera element znajdujący się na pozycji o wartości przesunięcia równej 0 (jednoznakowy łańcuch znaków 'm'). `S[-2]` pobiera element znajdujący się na pozycji o wartości przesunięcia 2 liczonej od końca (lub, odpowiednio, 8 + -2 od początku). Wartości przesunięcia i wycinki można odwzorować na komórki, jak na rysunku 7.1.⁵

Ostatni wiersz powyższego przykładu demonstruje tworzenie *wycinków* (ang. *slice*). Stanowią one uogólnioną formę indeksowania, które nie wykorzystuje *pozycji*, ale zakres podłańcucha elementów. Tworzenie wycinków najlepiej jest sobie wyobrazić jako formę analizy składniowej (*parsowania*), w szczególności przy zastosowaniu do łańcuchów znaków. Pozwala to na ekstrakcję

⁵ Osoby o większych zdolnościach matematycznych (a także studenci moich kursów) czasami zauważają tutaj niewielką asymetrię. Element znajdujący się najbardziej na lewo ma wartość przesunięcia 0, natomiast element umieszczony najbardziej na prawo -1. I faktycznie w Pythonie nie ma czegoś takiego, jak osobna wartość -0.



Rysunek 7.1. Wartości przesunięcia i wycinki. Dodatnie wartości przesunięcia liczone są od lewej strony (pierwszy element ma wartość 0), natomiast wartości ujemne — od prawej (ostatni element ma wartość przesunięcia równą -1). W indeksowaniu i tworzeniu wycinków można skorzystać z dowolnego typu wartości przesunięcia

całej części (łańcucha znaków) za jednym razem. Wycinki można wykorzystać do ekstrakcji kolumn danych czy odcinania tekstu znajdującego się na początku i końcu. W dalszej części rozdziału zobaczymy kolejny przykład tworzenia wycinków jako metodę analizy składowej tekstu.

A oto jak działają wycinki. Kiedy indeksuje się obiekt sekwencji — taki, jak łańcuch znaków — za pomocą pary wartości przesunięcia rozdzielonej dwukropkiem, Python zwraca nowy obiekt zawierający przylegającą część identyfikowaną przez tę parę wartości. Przedział taki jest lewostronnie zamknięty i prawostronnie otwarty, co oznacza, że Python pobiera wszystkie elementy od lewej wartości przesunięcia (*włącznie z nią*) do prawej wartości przesunięcia (*ale już bez niej samej*) i zwraca nowy obiekt zawierający pobrane elementy. Jeśli któraś z wartości zostaje pominięta, jej wartością domyślną jest odpowiednio zero i długość ciętego obiektu.

W przykładzie wyżej wyrażenie `S[1:3]` powodowało ekstrakcję elementów zajmujących pozycje o wartościach przesunięcia 1 i 2. Pobrane zostały zatem drugi i trzeci element, ale już nie element czwarty, mający wartość przesunięcia równą 3. Kolejny wycinek — `S[1:]` — pobiera *wszystkie elementy poza pierwszym*. Góra granica, która nie jest podana, ma wartość domyślną równą długości łańcucha znaków. Wreszcie wyrażenie `S[:-1]` pobiera *wszystkie elementy z wyjątkiem ostatniego*, ponieważ granica dolna ma wartość domyślną równą 0, natomiast -1 odnosi się do ostatniego elementu i zarazem nie obejmuje go (przedział jest prawostronnie otwarty).

Może się to na pierwszy rzut oka wydawać mylące, jednak indeksowanie i wycinki są narzędziami prostymi w użyciu (kiedy nabierze się już wprawy) i o dużych możliwościach. Należy pamiętać, że jeśli nie jest się pewnym, co oznacza dany wycinek, zawsze należy go przetestować w sesji interaktywnej. W kolejnym rozdziale zobaczymy, że można nawet zmienić całą część określonego obiektu za jednym razem, przypisując ją do wycinka (ta technika nie działa jednak z wartościami niemutowalnymi, jak łańcuchy znaków). Poniżej znajduje się podsumowanie zagadnień związanych z indeksowaniem i wycinkami.

- *Indeksowanie (`S[i:j]`)* pobiera komponenty znajdujące się na pozycji określonej wartością przesunięcia.
 - Pierwszy element ma wartość przesunięcia równą 0.
 - Indeks ujemny oznacza odliczanie od końca (inaczej — od prawej strony).
 - `S[0]` pobiera pierwszy element.
 - `S[-2]` pobiera drugi element od końca (podobnie jak `S[len(S)-2]`).

- *Wycinek* ($S[i:j]$) dokonuje ekstrakcji przylegającej części sekwencji.
 - Górną granicę jest otwarta.
 - Pominięte granice wycinka mają wartości domyślne 0 i długość sekwencji.
 - $S[1:3]$ pobiera elementy znajdujące się na pozycjach o wartościach przesunięcia od 1 do 2 włącznie (bez 3).
 - $S[1:]$ pobiera elementy znajdujące się na pozycjach o wartościach przesunięcia od 1 do końca (długości) łańcucha.
 - $S[:3]$ pobiera elementy znajdujące się na pozycjach o wartościach przesunięcia od 0 do 2 włącznie (bez 3).
 - $S[:-1]$ pobiera elementy znajdujące się na pozycjach od wartości przesunięcia 0 do ostatniego elementu, ale bez niego.
 - $S[:]$ pobiera elementy znajdujące się na pozycjach od wartości przesunięcia 0 do końca — w rezultacie otrzymuje się kopię łańcucha S najwyższego poziomu.

Ostatni element powyższej listy okazuje się często stosowaną sztuczką. Pozwala on wykonać pełną kopię najwyższego poziomu dla obiektu sekwencji, czyli uzyskać obiekt o tej samej wartości, jednak zajmujący odrębne miejsce w pamięci (więcej informacji na temat kopii można znaleźć w rozdziale 9.). Nie jest to zbyt przydatne w przypadku obiektów niezmiennych, jakimi są łańcuchy znaków, jednak przydaje się w połączeniu z obiektami, które można modyfikować w miejscu, jak listy.

W kolejnym rozdziale zobaczymy również, że składnia wykorzystywana do indeksowania po wartości przesunięcia (nawiasy kwadratowe) używana jest również do indeksowania słowników po kluczu. Operacje te wyglądają tak samo, jednak mają różne znaczenie.

Rozszerzone wycinki — trzeci limit i obiekty wycinków

W Pythonie 2.3 i późniejszych wersjach wyrażenia z wycinkami otrzymały opcjonalny trzeci indeks wykorzystywany jako *krok* (w języku angielskim znany jako *step* lub *stride*). Krok dodawany jest do indeksu każdego pobieranego elementu. Pełną formą wycinka jest teraz $X[I:J:K]$, co oznacza, że należy dokonać ekstrakcji wszystkich elementów z X znajdujących się na pozycjach o wartościach przesunięcia od I do $J-1$, co K element. Trzeci argument, K , ma wartość domyślną równą 1, dlatego normalnie pobierane są wszystkie elementy wycinka od lewej do prawej strony. Jeśli jednak podamy inną wartość tego argumentu, trzecią wartość można wykorzystać do pomijania niektórych elementów lub odwrócenia ich kolejności.

Wyrażenie $X[1:10:2]$ spowoduje na przykład pobranie co drugiego elementu z X pomiędzy pozycjami przesunięcia o wartościach 1 do 9, co da elementy z pozycji przesunięcia 1, 3, 5, 7 oraz 9. Tak jak wcześniej, pierwszy i drugi argument mają wartości domyślne równe odpowiednio 0 i długości sekwencji, dzięki czemu wyrażenie $X[::2]$ spowoduje wybranie co drugiego elementu od początku do końca sekwencji.

```
>>> S = 'abcdefghijklmno'
>>> S[1:10:2]
'bdfhj'
>>> S[::2]
'acegikmo'
```

Krok może również być wartością ujemną. Wyrażenie "halo"[::-1] zwraca na przykład nowy łańcuch znaków "olah". Pierwsze dwie granice mają wartości domyślne 0 i długość sekwencji,

natomiast krok o wartości -1 wskazuje na to, że wycinanie elementów powinno się odbywać od prawej strony do lewej zamiast — jak zazwyczaj — od lewej do prawej. W rezultacie sekwencja zostaje *odwrócona*.

```
>>> S = 'halo'  
>>> S[::-1]  
'olah'
```

Po podaniu ujemnej wartości kroku znaczenie dwóch pierwszych granic właściwie się odwraca. Oznacza to, że wycinek `S[5:1:-1]` powoduje pobranie elementów z pozycji przesunięcia od 2 do 5 w odwrotnej kolejności (wynik zawiera elementy z pozycji przesunięcia 5, 4, 3 oraz 2).

```
>>> S = 'abcdefg'  
>>> S[5:1:-1]  
'fdec'
```

Podobne przeskakiwanie elementów i ich odwracanie to najczęstsze zastosowanie wycinków z trzema argumentami. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona; warto również spróbować na własną rękę wykonać kilka eksperymentów w sesji interaktywnej. Do wycinków tego typu powrócimy ponownie w dalszej części książki przy okazji omawiania instrukcji pętli `for`.

W dalszej części książki dowiemy się, że tworzenie wycinków jest w rzeczywistości operacją indeksowania z zastosowaniem *obiektu wycinka* zamiast liczby całkowitej określającej indeks. Jest to obserwacja kluczowa dla programistów tworzących własne klasy, które mają obsługiwać operacje indeksowania i wycinania.

```
>>> 'mielonka'[1:3]                                # Składnia wycinania  
'ie'  
>>> 'mielonka'[slice(1, 3)]                      # Obiekty wycinków  
'ie'  
>>> 'mielonka'[::-1]  
'aknoleim'  
>>> 'mielonka'[slice(None, None, -1)]  
'aknoleim'
```

Narzędzia do konwersji łańcuchów znaków

Jedną z idei przyświecających projektowi Pythona jest to, że język ten ma zniechęcać do zgadywania. Ewidentnym przykładem tej koncepcji jest niemożność dodania do siebie liczb i łańcuchów znaków, nawet kiedy łańcuch wygląda jak liczba, to znaczy składa się z samych cyfr.

```
>>> "42" + 1  
TypeError: cannot concatenate 'str' and 'int' objects
```

Jest to celowe działanie — ponieważ symbol + może oznaczać zarówno dodawanie, jak i konkatenację, wybór konwersji nie byłby jednoznaczny. Python traktuje zatem takie wyrażenie jak błąd. W języku tym z reguły stara się unikać magii, jeśli może nam ona skomplikować życie.

Co zatem można zrobić, gdy skrypt pobierze liczbę jako łańcuch tekstowy z pliku czy interfejsu użytkownika? Sztuczka polega na zastosowaniu narzędzi do konwersji przed potraktowaniem łańcucha jako liczby bądź odwrotnie.

```
>>> int("42"), str(42)                            # Konwersja z łańcucha i na łańcuch  
(42, '42')  
>>> repr(42)                                     # Konwersja na łańcuch w postaci kodu  
'42'
```

Znaczenie wycinków

W całej książce będę zamieszczał ramki z przypadkami zastosowania (jak ta tutaj), które pozwolą zobaczyć, w jaki sposób niektóre omawiane możliwości języka są zazwyczaj stosowane w prawdziwych programach. Ponieważ trudno zrozumieć rzeczywiste przypadki zastosowania bez znajomości samego Pythona, ramki te będą zazwyczaj zawierały odniesienia do zagadnień, które jeszcze nie były wprowadzone. Można je również potraktować jako zapowiedź tego, w jaki sposób pewne abstrakcyjne koncepcje z języka przydają się w często wykonywanych zadaniach programistycznych.

Przykładowo później przekonamy się, że argumenty podane w systemowym wierszu poleceń i wykorzystywane do uruchamiania programu w Pythonie są udostępniane w atrybutie `argv` wbudowanego modułu `sys`.

```
# Plik echo.py
import sys
print(sys.argv)

% python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

Zazwyczaj interesują nas tylko argumenty następujące po nazwie programu. Z tej potrzeby zrodziło się bardzo typowe zastosowanie wycinków — jedno wyrażenie z wycinkiem można wykorzystać do zwrócenia wszystkich elementów listy z wyjątkiem pierwszego. Wyrażenie `sys.argv[1:]` zwraca pożądaną listę `['-a', '-b', '-c']`. Możemy ją następnie przetworzyć bez konieczności troszczenia się o nazwę programu, która znajdowała się na początku.

Wycinki są również często wykorzystywane do oczyszczania wierszy wczytanych z plików wejściowych. Jeśli wiemy, że wiersz będzie miał na końcu znak końca wiersza (a właściwie `\n` oznaczający nowy wiersz), możemy się go pozbyć za pomocą jednego wyrażenia, takiego jak `line[:-1]`, które powoduje ekstrakcję wszystkich znaków z wiersza z wyjątkiem ostatniego (pierwsza granica ma domyślną wartość 0). W obu przypadkach wycinek spełnia rolę logiki, która w językach niższego rzędu musi być napisana w jawnym sposobie.

Warto przypomnieć, że często preferowane jest wywołanie metody `line.rstrip`, która odcina znaki nowego wiersza — to wywołanie pozostawia wiersz nietknięty, kiedy na jego końcu nie ma znaków nowego wiersza, co często ma miejsce w przypadku plików tworzonych przez narzędzia do edycji tekstu. Zastosowanie w tej sytuacji wycinka ma sens wtedy, gdy wiadomo, że wiersz jest poprawnie zakończony.

Funkcja `int` przekształca łańcuch znaków na liczbę, natomiast funkcja `str` — liczbę na łańcuch znaków (czyli to, jak wygląda ona po wyświetleniu). Funkcja `repr` (i jej starszy odpowiednik — wyrażenie z apostrofem lewym, usunięte w wersji 3.0) również konwertuje liczbę na jej reprezentację łańcuchową, jednak zwraca obiekt jako łańcuch kodu, który można wykonać w celu odtworzenia obiektu. W przypadku łańcuchów znaków wynik zawiera apostrofy, kiedy wyświetli się go za pomocą instrukcji `print`:

```
>>> print(str('mielonka'), repr('mielonka'))
('mielonka', "mielonka")
```

Więcej informacji na ten temat można znaleźć w ramce „Formaty wyświetlania str i repr” na stronie 159. Zazwyczaj poleca się wykorzystywanie przy konwersji funkcji `int` oraz `str`.

Choć nie można mieszać ze sobą łańcuchów znaków oraz liczb wokół operatorów takich, jak +, jeśli wystąpi taka potrzeba, można ręcznie przekonwertować argumenty przed wykonaniem działania.

```
>>> S = "42"
>>> I = 1
>>> S + I
TypeError: cannot concatenate 'str' and 'int' objects

>>> int(S) + I                                # Wymuszenie dodawania
43

>>> S + str(I)                               # Wymuszenie konkatenacji
'421'
```

Podobne funkcje wbudowane obsługują konwersje liczb zmiennoprzecinkowych na łańcuchy znaków i odwrotnie.

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)

>>> text = "1.234E-10"
>>> float(text)
1.2340000000000001e-010
```

Nieco później będziemy również omawiać wbudowaną funkcję eval. Wykonuje ona łańcuch znaków zawierający kod wyrażenia Pythona i może konwertować łańcuch znaków na obiekt dowolnego typu. Funkcje int i float dokonują konwersji jedynie na liczby, jednak ograniczenie to oznacza również, że są zazwyczaj szybsze (i bardziej bezpieczne, ponieważ nie przyjmują dowolnego kodu wyrażenia). Jak widzieliśmy w rozdziale 5., konwersji liczb na łańcuchy znaków można również dokonać za pomocą wyrażenia formatującego z łańcuchem znaków. Zostanie to omówione w dalszej części rozdziału.

Konwersje kodu znaków

Jeśli chodzi o konwersje, można również przekształcić pojedynczy znak do jego kodu liczbowego ASCII, przekazując go do wbudowanej funkcji ord. Zwraca ona prawdziwą wartość binarną odpowiadającą znakowi bajta w pamięci. Funkcja chr wykonuje odwrotną operację, przyjmując kod liczbowy ASCII i przekształcając go na odpowiadający mu znak.

```
>>> ord('s')
115
>>> chr(115)
's'
```

By zastosować te funkcje do wszystkich znaków łańcucha, należy wykonać pętlę. Narzędzia te można wykorzystać do wykonania swego rodzaju działań matematycznych opartych na łańcuchach znaków. By na przykład przejść do kolejnego znaku, należy dokonać konwersji i obliczeń dla liczby całkowitej.

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

Przynajmniej w przypadku łańcuchów jednoznakowych jest to alternatywa dla używania wbudowanej funkcji `int` do konwersji łańcucha znaków na liczbę całkowitą.

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```

Taką konwersję można wykorzystać w połączeniu z instrukcją pętli, wprowadzonej w rozdziale 4. i omówionej dogłębnie w następnej części książki, do przekształcenia łańcucha cyfr binarnych na odpowiadające im wartości liczb całkowitych. Przy każdym przejściu aktualna wartość mnożona jest przez 2 i dodawana do wartości liczby całkowitej dla kolejnej cyfry.

```
>>> B = '1101'          # Przekształcenie łańcucha znaków w notacji dwójkowej na liczbę całkowitą z użyciem ord
>>> I = 0
>>> while B != '':
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
13
```

Podobny efekt do mnożenia przez 2 miałaby operacja przesunięcia bitowego w lewo (`I << 1`). Tę modyfikację pozostawiam jako sugerowane ćwiczenie dla Czytelnika, po części dlatego, że jeszcze nie przeanalizowaliśmy pętli, ale też dlatego, że w Pythonie 2.6 i 3.0 przekształcenia z notacji dwójkowej na liczbę całkowitą i z powrotem można dokonać z użyciem, odpowiednio, funkcji `int` i `bin` omówionych w rozdziale 5.

```
>>> int('1101', 2)          # Przekształcenie łańcucha znaków w notacji dwójkowej na liczbę całkowitą
13
>>> bin(13)                # Przekształcenie liczby całkowitej na łańcuch znaków w notacji dwójkowej
'0b1101'
```

Python ma skłonność do automatyzacji powszechnie wykonywanych zadań, wystarczy tylko dać mu trochę czasu!

Modyfikowanie łańcuchów znaków

Warto przypomnieć tutaj pojęcie „niezmiennej sekwencji”. „Niezmienna” oznacza, że nie można zmienić łańcucha znaków w miejscu (na przykład przypisując go do indeksu).

```
>>> S = 'mielonka'
>>> S[0] = "x"
Zwraca błąd!
```

W jaki sposób można zatem zmodyfikować w Pythonie informacje tekstowe? By zmienić łańcuch znaków, należy utworzyć i przypisać nowy łańcuch znaków za pomocą odpowiednich narzędzi, takich jak konkatenacja czy wycinki, a następnie — w miarę potrzeby — przypisać rezultat z powrotem do oryginalnej nazwy łańcucha.

```
>>> S = S + 'MIELONKA!'          # By zmienić łańcuch, należy utworzyć nowy
>>> S
'mielonkaMIELONKA!'
>>> S[8:] + 'Jajka' + S[-1]
>>> S
'mielonkaJajka!'
```

W pierwszym przykładzie na końcu łańcucha `S` dodany zostaje (dzięki konkatenacji) nowy pod-łańcuch znaków (tak naprawdę działanie to tworzy nowy łańcuch znaków i przypisuje go

z powrotem do zmiennej `S`, ale możemy je sobie wyobrazić jako „modyfikację” oryginalnego łańcucha). W drugim przykładzie osiem znaków zostaje zastąpionych pięcioma za pomocą wycinka, indeksowania i konkatenacji. Jak zobaczymy w dalszej części rozdziału, podobny rezultat można osiągnąć za pomocą wywołania metod, takich jak `replace`:

```
>>> S = 'mielonka'  
>>> S = S.replace('lon', 'lonecz')  
>>> S  
'mieloneczka'
```

Tak jak wszystkie operacje zwracające nowe wartości łańcuchów znaków, metody łańcuchów znaków generują nowe obiekty. Jeśli chcemy je zachować, można je przypisać do nazwy zmiennej. Wygenerowanie nowego obiektu łańcucha znaków dla każdej modyfikacji tego łańcucha nie jest aż tak niewydajne, jak może brzmieć — należy pamiętać, że — tak jak wspomnieliśmy w poprzednim rozdziale — Python automatycznie czyści stare, nieużywane obiekty łańcuchów znaków w miarę postępu programu, dzięki czemu nowsze obiekty otrzymują przestrzeń zajmowaną wcześniej przez poprzednie wartości. Python zazwyczaj jest bardziej wydajny, niż moglibyśmy się tego spodziewać.

Można również budować nowe wartości tekstowe za pomocą wyrażeń formatujących z łańcuchami znaków. Obydwa poniższe przykłady dokonują podstawienia obiektów do łańcucha znaków, to znaczy przekształcają obiekty na łańcuchy znaków i zmieniają oryginalny łańcuch znaków zgodnie ze specyfikacją formatowania.

```
>>> 'Ten %d %s jest martwy!' % (1, 'ptak')           # Wyrażenie formatujące  
'Ten 1 ptak jest martwy'  
>>> 'Ten {0} {1} jest martwy!'.format(1, 'ptak')      # Metoda formatująca w 2.6 i 3.0  
'Ten 1 ptak jest martwy!'
```

Mimo że zastosowana składnia sugeruje podstawienie obiektów do łańcucha znaków, w rzeczywistości obydwie powyższe formy tworzą nowe łańcuchy, nie modyfikując istniejących. Formatowanie omówimy bardziej szczegółowo w dalszej części rozdziału. Jak się okaże, to narzędzie jest znacznie bardziej użyteczne, niż sugerowałby powyższy przykład. Drugie z powyższych wywołań wykorzystuje metodę, najwyższy czas zatem omówić inne metody obiektów tekstowych, zanim zagłębimy się w tajniki formatowania.



W Pythonie 2.6 i 3.0 wprowadzono nowy typ łańcuchów znaków, znany jako `bytearray`, którego szczegóły poznamy w rozdziale 36. Jest to mutowalny typ pozwalający na dokonywanie zmian w miejscu. Obiekty typu `bytearray` nie są łańcuchami znaków, są sekwencjami 8-bitowych liczb całkowitych. Jednak obsługują większość tych samych operacji co zwykłe łańcuchy znaków i można je wyświetlać tak samo jak łańcuchy znaków ASCII. Dzięki swoim cechom szczególnym doskonale nadają się do obsługi dużych ilości tekstu, które muszą być często modyfikowane. W rozdziale 36. zobaczymy również, że funkcje `ord` i `chr` obsługują także znaki Unicode, które do zapisu mogą potencjalnie wymagać kilku bajtów.

Metody łańcuchów znaków

Poza operatorami wyrażeń łańcuchy znaków udostępniają zbiór *metod* implementujących bardziej wyszukane zadania związane z przetwarzaniem tekstu. Metody to po prostu funkcje powiązane z określonymi obiektami. Z technicznego punktu widzenia są atrybutami dodawianymi

do obiektów, które odnoszą się do wywoływalnych funkcji. W Pythonie wyrażenia i funkcje wbudowane mogą działać na różnych typach obiektów, ale metody są specyficzne dla typów obiektów — na przykład metody łańcuchów znaków działają tylko na obiektach łańcuchów znaków. Zestawy metod niektórych typów nakładają się na siebie w Pythonie 3.0 (na przykład wiele typów posiada metodę `count`), ale mimo to metody są bardziej *specyficzne dla typów* niż większość innych narzędzi.

Mówiąc bardziej szczegółowo, funkcje to pakiety kodu, a wywołania metod łączą w sobie dwie operacje naraz (pobranie atrybutu i wywołanie).

Pobieranie atrybutu

Wyrażenie w postaci `obiekt.atrybut` oznacza: „Pobierz wartość atrybutu z obiektu”.

Wywołanie

Wyrażenie w postaci `funkcja(argumenty)` oznacza: „Wywołaj kod funkcji, przekazując jej zero lub większą liczbę rozdzielonych przecinkami obiektów argumentów, i zwróć wartość wyniku funkcji”.

Połączenie ze sobą tych dwóch elementów pozwala na wywołanie metody obiektu. Wyrażenie będące wywołaniem metody `obiekt.metoda(argumenty)` jest przetwarzane od lewej strony do prawej. Oznacza to, że Python najpierw pobiera metodę obiektu, a następnie wywołuje ją, przekazując argumenty. Jeśli metoda oblicza wynik, zostanie on zwrócony jako wynik całego wyrażenia wywołującego metodę.

Jak zobaczymy w tej części książki, większość obiektów posiada wywoływalne metody i każda z nich wywoływana jest za pomocą tej samej składni. Jak się okaże, aby wywołać metodę obiektu, trzeba przejść przez istniejący obiekt.

W tabeli 7.3 podsumowano metody i wzorce wywoływania większości metod obiektów wbudowanych typów znakowych w Pythonie 3.0. Metody tych typów często ulegają zmianom, warto więc zerknąć do dokumentacji biblioteki standardowej Pythona w celu przejrzenia najbardziej aktualnej listy lub wywołać funkcję `help` dla łańcucha znaków w sesji interaktywnej. W Pythonie 2.6 metody łańcuchów znaków nieco się różnią, na przykład `decode`, w której różni się obsługa znaków Unicode (co omówimy w rozdziale 36.). W tej tabeli `S` reprezentuje obiekt typu `string`, opcjonalne argumenty zostały ujęte w nawiasy kwadratowe. Metody łańcuchów znaków znajdujące się w tabeli implementują operacje wyższego poziomu, takie jak dzielenie i łączenie, zmiana wielkości liter, sprawdzanie zawartości oraz szukanie i zastępowanie podłańcuchów znaków.

Jak widać, istnieje spora liczba metod znakowych i nie mamy tu miejsca, aby je wszystkie omówić. Szczegóły można znaleźć w dokumentacji standardowej biblioteki Pythona. Jednak na dobry początek przyjrzyjmy się teraz kodowi demonstrującemu niektóre z częściej stosowanych metod i przy okazji ilustrującemu podstawy przetwarzania tekstu w Pythonie.

Przykłady metod łańcuchów znaków — modyfikowanie

Jak widzieliśmy, ponieważ łańcuchy znaków są niezmienne, nie można ich bezpośrednio zmieniać w miejscu. By z istniejącego łańcucha utworzyć nową wartość tekstową, konstruuje się nowy łańcuch za pomocą operacji takich, jak wycinki czy konkatenacja. Żeby na przykład zastąpić znaki w środku łańcucha, należy skorzystać z poniższego kodu.

Tabela 7.3. Wywołania metod łańcuchów znaków w Pythonie 3.0

S.capitalize()	S.ljust(width [, fill])
S.center(width [, fill])	S.lower()
S.count(sub [, start [, end]])	S.lstrip([chars])
S.encode([encoding [,errors]])	S.maketrans(x[, y[, z]])
S.endswith(suffix [, start [, end]])	S.partition(sep)
S.expandtabs([tabsize])	S.replace(old, new [, count])
S.find(sub [, start [, end]])	S.rfind(sub [,start [,end]])
S.format(fmtstr, *args, **kwargs)	S.rindex(sub [, start [, end]])
S.index(sub [, start [, end]])	S.rjust(width [, fill])
S.isalnum()	S.rpartition(sep)
S.isalpha()	S.rsplit([sep[, maxsplit]])
S.isdecimal()	S.rstrip([chars])
S.isdigit()	S.split([sep [,maxsplit]])
S.isidentifier()	S.splitlines([keepends])
S.islower()	S.startswith(prefix [, start [, end]])
S.isnumeric()	S.strip([chars])
S.isprintable()	S.swapcase()
S.isspace()	S.title()
S.istitle()	S.translate(map)
S.isupper()	S.upper()
S.join(iterable)	S.zfill(width)

```
>>> S = 'jabłka'
>>> S = S[:2] + 'bł' + S[3:]
>>> S
'jabłka'
```

Jeśli jednak naprawdę chcemy tylko zastąpić podłańcuch, możemy zamiast tego skorzystać z metody łańcuchów znaków `replace`.

```
>>> S = 'jabłka'
>>> S.replace('jk', 'błk')
>>> S
'jabłka'
```

Metoda `replace` jest bardziej ogólna, niż wskazuje na to powyższy przykład. Jako argumenty przyjmuje oryginalny podłańcuch (dowolnej długości) oraz łańcuch (również dowolnej długości), którym zostanie on zastąpiony. Następnie wykonuje globalne wyszukanie i zastąpienie.

```
>>> 'aa$bb$cc$dd'.replace('$', 'JAJKA')
'aaJAJKAbbJAJKAccJAJKAdd'
```

W takiej roli metodę `replace` można wykorzystać jako narzędzie implementujące zastąpienia szablonów (na przykład listów z formularzami). Warto zauważyc, że tym razem po prostu wyświetliśmy wynik, zamiast przypisywać go do zmiennej — przypisanie do zmiennej jest konieczne tylko wtedy, gdy chcemy zachować wynik do dalszej pracy.

Jeśli chcemy zastąpić jeden łańcuch znaków o określonej wielkości, który może się pojawić w dowolnym miejscu, można albo ponownie skorzystać z metody `replace`, albo najpierw odszukać podłańcuch za pomocą metody `find`, a później skorzystać z wycinka.

```

>>> S = 'xxxxJAJKAxXXXJAJKAxXXXX'
>>> where = S.find('JAJK')
        # Odszukanie pozycji
>>> where
4
>>> S = S[:where] + 'MIELONKA' + S[where+4:]
>>> S
'xxxxMIELONKAXXXXJAJKAxXXXX'

```

Metoda `find` zwraca wartość pozycji przesunięcia, na której znajduje się podłańcuch (domyślnie wyszukiwanie odbywa się od początku łańcucha), lub `-1`, jeśli nie zostanie on odnaleziony. Jak zauważyliśmy wcześniej, jest to operacja wyszukiwania podłańcuchów znaków działająca podobnie do wyrażenia `in`, ale zwracająca pozycję znalezionej podłańcuchu.

Inną opcją jest skorzystanie z metody `replace` w połączeniu z trzecim argumentem, który ograniczy ją do jednego zastąpienia.

```

>>> S = 'xxxxJAJKAxXXXJAJKAxXXXX'
>>> S.replace('JAJK', 'MIELONKA')           # Zastępuje wszystkie wystąpienia
'xxxxMIELONKAXXXXMIELONKAXXXXX'

>>> S.replace('JAJK', 'MIELONKA', 1)         # Zastępuje jedno wystąpienie
'xxxxMIELONKAXXXXJAJKAxXXXX'

```

Warto zwrócić uwagę na to, że metoda `replace` zwraca za każdym razem nowy obiekt łańcucha znaków. Ponieważ łańcuchy znaków są niezmienne, metody nigdy tak naprawdę nie modyfikują ich w miejscu, nawet jeśli ich nazwy (`replace` — zastąp) mogą to sugerować.

To, że operacje konkatenacji i metoda `replace` generują za każdym razem nowe obiekty łańcuchów znaków, może być potencjalną wadą używania ich do zastępowania łańcuchów. Jeśli w jednym łańcuchu znaków musimy zastosować wiele zmian, być może uda nam się poprawić wydajność skryptu, przekształcając łańcuch znaków na obiekt obsługujący zmiany w miejscu.

```

>>> S = 'brama'
>>> L = list(S)
>>> L
['b', 'r', 'a', 'm', 'a']

```

Wbudowana funkcja `list` (lub wywołanie konstruktora obiektu) tworzy nową listę z elementów dowolnej sekwencji — w tym przypadku rozbijając znaki łańcucha na listę. Kiedy łańcuch będzie miał taką postać, możemy wprowadzić do niego wiele zmian bez generowania za każdym razem nowej jego kopii.

```

>>> L[2] = 'e'                                # Działa dla list, ale nie dla łańcuchów znaków
>>> L[3] = 'j'
>>> L
['b', 'r', 'e', 'j', 'a']

```

Jeśli po tych zmianach musimy przekształcić obiekt z powrotem na łańcuch znaków (na przykład w celu zapisania go do pliku), należy skorzystać z metody łańcuchów znaków `join`, która złoży listę z powrotem w łańcuch.

```

>>> S = ''.join(L)
>>> S
'breja'

```

Metoda `join` może na pierwszy rzut oka wyglądać nieco dziwnie. Ponieważ jest metodą łańcuchów znaków (a nie list), wywoływana jest przez pożądany ogranicznik. Metoda ta łączy ze sobą łańcuchy podane w postaci listy (lub innego obiektu iterowanego), wstawiając pomiędzy

nie podany ogranicznik. W tym przypadku wykorzystuje jako ogranicznik pusty łańcuch znaków do przekształcenia listy z powrotem w łańcuch. Metoda ta ma jednak bardziej uniwersalne zastosowanie i może działać z dowolnym ogranicznikiem łańcucha i listą łańcuchów znaków.

```
>>> 'JAJKA'.join(['mielonka', 'ser', 'szynka', 'tost'])
'mielonkaJAJKAserJAJKAszynkaJAJKAtost'
```

W rzeczywistości łączenie łańcuchów znaków w ten sposób będzie szybsze od konkatenacji indywidualnych łańcuchów. Warto również zwrócić uwagę na wspomniany wyżej typ `bytearray` wprowadzony w Pythonie 3.0 i 2.6 (omówiony szerzej w rozdziale 36.), ponieważ może on być modyfikowany w miejscu. Oferuje on zatem alternatywę dla tego typu kombinacji metody `join` na liście łańcuchów znaków, co może być przydatne w przypadku dużych tekstów, które muszą być często modyfikowane.

Przykłady metod łańcuchów znaków

— analiza składniowa tekstu

Kolejną często spotykaną rolą metod łańcuchów znaków jest *analiza składniowa tekstu*, czyli analiza jego struktury i ekstrakcja podłańcuchów. By pobrać podłańcuch znajdujący się na określonych pozycjach przesunięcia, można skorzystać z wycinków.

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

W powyższym kodzie kolumny danych występują na stałych pozycjach przesunięcia, dzięki czemu można je wyciąć z oryginalnego łańcucha znaków. Technika ta nadaje się do analizy składniowej tekstu tak długo, jak komponenty naszych danych mają stałe pozycje. Gdyby zamiast tego dane rozdzielały jakiś rodzaj ogranicznika, można komponenty łańcucha pobrać za pomocą podziału. Będzie to działało nawet wtedy, gdy dane występują w łańcuchu na przypadkowych pozycjach.

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split(' ')
>>> cols
['aaa', 'bbb', 'ccc']
```

Metoda łańcuchów znaków `split` dzieli łańcuch na listę podłańcuchów według ogranicznika. W poprzednim przykładzie nie przekazaliśmy ogranicznika, przez co użyta została wartość domyślna, którą jest biały znak. Łańcuch znaków jest dzielony w miejscu wystąpienia jednej bądź kilku spacji, tabulatorów lub nowych wierszy. Metoda ta zwraca listę wynikowych podłańcuchów. Można ją również zastosować w połączeniu z innymi ogranicznikami. Poniższy przykład dzieli (i tym samym analizuje składniowo) łańcuch znaków w miejscu wystąpienia przecinka, który często rozdziela dane zwracane przez pewne bazy danych.

```
>>> line = 'amadeusz,haker,40'
>>> line.split(',')
['amadeusz', 'haker', '40']
```

Ograniczniki mogą składać się z większej liczby znaków, jak poniżej.

```
>>> line = "jestemMIELONKAdrwalemMIELONKAiMIELONKAjestemMIELONKAOK"
>>> line.split("MIELONKA")
['jestem', 'drwalem', 'i', 'jestem', 'OK']
```

Choć istnieją ograniczenia co do potencjału wycinków i podziałów w analizie składniowej, obie operacje działają bardzo szybko i świetnie sobie radzą z podstawowymi zadaniami wymagającymi ekstrakcji tekstu.

Inne znane metody łańcuchów znaków w akcji

Pozostałe metody łańcuchów znaków mają bardziej wyspecjalizowane role. Służą na przykład do odcinania białych znaków na końcu wiersza, zmiany wielkości liter, sprawdzania zawartości łańcuchów znaków czy istnienia danego podłańcucha na końcu.

```
>>> line = "Rycerze, którzy mówią Ni!\n"
>>> line.rstrip()
'Rycerze, którzy mówią Ni!'
>>> line.upper()
'RYCERZE, KTÓRZY MÓWIĄ NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\\n')
True
>>> line.startswith('Rycerze')
True
```

Czasami do uzyskania tych samych efektów co za pomocą metod łańcuchów znaków można wykorzystać również alternatywne techniki. Żeby na przykład sprawdzić obecność podłańcucha znaków, można wykorzystać operator `in`. Połączenie operacji obliczenia długości i spóźdzenia wycinków może natomiast zastąpić metodę `endswith`.

```
>>> line
'Rycerze, którzy mówią Ni!\n'

>>> line.find('Ni') != -1           # Szukanie za pomocą wywołania metody lub wyrażenia
True
>>> 'Ni' in line                  # Sprawdzenie końca łańcucha za pomocą wywołania metody lub wycinka
True
>>> line[-len(sub):] == sub
True
```

Zachęcam do zapoznania się z metodami formatowania opisanyimi w dalszej części rozdziału. Znajdziemy tam narzędzia podstawiania łańcuchów znaków wykonujące kilka czynności w pojedynczym wywołaniu.

Ponieważ istnieje tak wiele metod łańcuchów znaków, nie będziemy tutaj omawiać każdej z nich po kolej. Pewne dodatkowe przykłady z łańcuchami znaków znajdują się w dalszej części książki, jednak więcej szczegółów należy odszukać w dokumentacji biblioteki Pythona czy innych źródłach bądź też samodzielnie poeksperymentować w sesji interaktywnej. Więcej sugestii użycia można uzyskać również, wywołując funkcję `help(S.metoda)`.

Warto podkreślić, że żadna z metod łańcuchów znaków nie przyjmuje wzorców. Do przetwarzania tekstu opartego na wzorcach należy użyć modułu wyrażeń regularnych `re` z biblioteki standardowej Pythona — zaawansowanego narzędzia przedstawionego w rozdziale 4., które

jednak wykracza poza zakres niniejszego tekstu (dodatkowy przykład można znaleźć pod koniec rozdziału 36.). Ze względu na to ograniczenie metody łańcuchów znaków działają czasami szybciej od narzędzi z modułu `re`.

Oryginalny moduł `string` (usunięty w 3.0)

Historia metod łańcuchów znaków Pythona jest nieco zagmatwana. Mniej więcej przez pierwszą dekadę istnienia języka Python udostępniany był moduł biblioteki standardowej o nazwie `string`. Zawierał on funkcje, które w dużej mierze odzwierciedlały obecny zbiór metod obiektów łańcuchów znaków. W odpowiedzi na prośby użytkowników w Pythonie 2.0 funkcje te zostały udostępnione jako metody obiektów łańcuchów znaków. Ponieważ jednak wiele osób napisała już duże ilości kodu w oparciu o oryginalny moduł `string`, pozostał on w bibliotece, by kod ten nadal działał.

Dzisiaj powinniśmy korzystać *jadynie z metod łańcuchów znaków*, a nie z oryginalnego modułu `string`. Tak naprawdę wywołania z oryginalnego modułu odpowiadające dzisiejszym metodom łańcuchów znaków zostały zupełnie usunięte w Pythonie 3.0. Ponieważ jednak nadal można napotkać kod korzystający z tego modułu, warto o nim wspomnieć.

Rezultatem tych pozostałości jest to, że w Pythonie 2.6 nadal istnieją dwa sposoby wywoływania bardziej zaawansowanych operacji na łańcuchach znaków — poprzez wywołanie metod obiektu oraz poprzez wywołanie funkcji modułu `string` i przekazanie obiektu jako argumentu. Mając na przykład zmienną `X` przypisaną do obiektu łańcucha znaków, wywołanie metody obiektu:

```
X.metoda(argumenty)
```

jest zazwyczaj odpowiednikiem wywołania tej samej operacji za pośrednictwem modułu `string` (o ile oczywiście najpierw ten moduł zimportowaliśmy):

```
string.metoda(X, argumenty)
```

Poniżej znajduje się przykład zastosowania metody łańcucha znaków.

```
>>> S = 'a+b+c+'
>>> x = S.replace('+', 'jajka')
>>> x
'ajajkabajkacjajka'
```

By uzyskać dostęp do tej samej operacji za pośrednictwem modułu `string` w Pythonie 2.6, należy zimportować ten moduł (przynajmniej raz na proces) i przekazać mu obiekt.

```
>>> import string
>>> y = string.replace(S, '+', 'jajka')
>>> y
'ajajkabajkacjajka'
```

Ponieważ podejście z modelem było standardem przez wiele lat, a także dlatego, że łańcuchy znaków są tak ważnym komponentem większości programów, w kodzie napisanym w Pythonie 2.x można spotkać oba wzorce wywołania.

Ponownie trzeba jednak podkreślić, że obecnie należy korzystać z wywołania metod, a nie ze starszego wywoływania modułów. Ma to swoje uzasadnienie związane nie tylko z tym, że obsługa wywołania modułu zniknęła w Pythonie 3.0. Po pierwsze, wywoływanie modułu wymaga wcześniejszego zainportowania go (w przypadku metod nie jest to konieczne). Po drugie, moduł sprawia, że całe wywołanie robi się o kilka znaków dłuższe (jest tak wtedy, gdy

moduł ładuje się za pomocą instrukcji `import`, a nie `from`). I wreszcie — moduł działa wolniej od metod (moduł odwzorowuje wywołania z powrotem na metody, co wprowadza niepotrzebne dodatkowe wywołanie).

W Pythonie 3.0 pozostawiono moduł `string`, ale usunięto z niego funkcje odpowiadające metodom obiektów tekstowych, ponieważ zawiera dodatkowe narzędzia, w tym zdefiniowane stałe łańcuchów znaków, a także system obiektów szablonów (dość prymitywne narzędzie pominięte w naszym omówieniu; więcej informacji na temat obiektów szablonów można znaleźć w dokumentacji biblioteki standardowej Pythona). Nawet jeśli nie mamy ochoty zmieniać kodu 2.6 w taki sposób, by był w pełni zgodny z 3.0, należy uznać wywołania operacji z modułu `string` za duchy z przeszłości.

Wyrażenia formatujące łańcuchy znaków

Omówione wyżej metody łańcuchów znaków i operacje sekwencji dają duże możliwości, ale Python oferuje bardziej zaawansowane sposoby realizacji zadań związanych z łańcuchami znaków: *formatowanie łańcuchów znaków* pozwala nam przeprowadzić wiele podstawień w łańcuchu znaków w pojedynczej operacji. Użycie tych narzędzi nigdy nie jest niezbędne do realizacji zadania, ale często stanowi wygodne podejście, szczególnie przy formatowaniu łańcuchów znaków wyświetlanych użytkownikowi programu. Dzięki temu, że Python rozwija się wielokierunkowo, również mechanizmy formatowania łańcuchów znaków są dostępne w dwóch odmianach:

Wyrażenia formatujące

Oryginalna technika formatowania łańcuchów znaków dostępna w Pythonie od zawsze.

Jest oparta na zasadzie zastosowanej w funkcji `sprintf` języka C. Przykłady użycia tej metody można znaleźć praktycznie w każdym programie napisanym w Pythonie.

Metoda format

To nowsza technika, dodana w Pythonie 2.6 i 3.0. Jest dość unikalna dla Pythona i w znacznym stopniu pokrywa się z możliwościami wyrażeń formatujących.

Metoda `format` jest nowością, więc istnieje prawdopodobieństwo, że pierwsza z technik formatowania z czasem zostanie wycofana z języka z użyciem mechanizmu przedawnienia, jednak tak naprawdę zależy to od stopnia przywiązania programistów do tego elementu Pythona. Obecnie obie techniki są równoważne, ponieważ stanowią różne podejście do tego samego tematu. Zaczniemy od omawiania wyrażeń formatujących, ponieważ były dostępne jako pierwsze.

Python definiuje operator binarny `%`, który działa na łańcuchach znaków (jak pamiętamy, jest to również operator reszty dzielenia, czy inaczej modulo, dla liczb). Po zastosowaniu do łańcuchów znaków operator ten udostępnia łatwy sposób formatowania wartości w postaci łańcuchów znaków, zgodnie z definicją formatu. Krótko mówiąc, operator `%` udostępnia skrócony sposób kodowania kilku zastąpień łańcucha znaków w jednym wywołaniu, bez konieczności manualnego sklejania wyniku z poszczególnych elementów.

By formatować łańcuchy znaków, należy:

1. Po lewej stronie operatora `%` podać formatowany łańcuch zawierający jeden lub większą liczbę osadzonych celów konwersji, z których każdy rozpoczyna się od znaku `%` (na przykład `%d`).

2. Po prawej stronie operatora % udostępnić obiekt (lub obiekty, zapisane w krotce), który Python ma wstawić do formatowanego łańcucha po lewej stronie w miejsce celu lub celów konwersji.

W przykładzie z wcześniejszej części rozdziału liczba całkowita 1 zastępuje %d w formatowanym łańcuchu znaków znajdującym się po lewej stronie operatora, natomiast łańcuch 'ptak' zastępuje %s. W rezultacie otrzymujemy nowy łańcuch znaków odzwierciedlający te dwa podstawnienia.

```
>>> 'Ten %d %s jest martwy!' % (1, 'ptak') # Wyrażenie formatujące  
Ten 1 ptak jest martwy!
```

Z technicznego punktu widzenia wyrażenia formatujące łańcuchy znaków są zazwyczaj opcjonalne — to samo można najczęściej uzyskać za pomocą kilku konkatenacji i konwersji. Formatowanie pozwala jednak połączyć kilka kroków w jedną operację. Ma na tyle duże możliwości, że zasługuje na kilka kolejnych przykładów.

```
>>> exclamation = "Ni"  
>>> "Rycerze, którzy mówią %s!" % exclamation  
'Rycerze, którzy mówią Ni!'  
  
>>> "%d %s %d you" % (1, 'spam', 4)  
'1 spam 4 you'  
  
>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])  
'42 -- 3.14159 -- [1, 2, 3]'
```

Pierwszy z powyższych przykładów wstawia łańcuch znaków "Ni" w miejsce docelowe po lewej stronie, zastępując znacznik %. W drugim przykładzie do docelowego łańcucha znaków wstawiane są cztery wartości. Warto zauważyć, że kiedy do łańcucha wstawia się większą liczbę wartości, konieczne jest zgrupowanie ich w nawiasy po prawej stronie (czyli umieszczenie w krotce). Operator wyrażenia formatującego % oczekuje po prawej stronie pojedynczego elementu lub krótki elementów.

W trzecim przykładzie ponownie wstawiane są trzy wartości — obiekt liczby całkowitej, liczby zmiennoprzecinkowej i listy, jednak warto zauważyć, że wszystkie miejsca docelowe po lewej stronie to %s, co oznacza konwersję na łańcuch znaków. Ponieważ każdy typ obiektu można przekształcić na łańcuch znaków (taki wykorzystywany w czasie wyświetlania), każdy typ obiektu zadziała w połączeniu z kodem konwersji %s. Z tego powodu, o ile nie wykonujemy jakiegoś formatowania specjalnego, %s jest często jedynym kodem, który trzeba zapamiętać jako wyrażenie formatujące.

Należy również pamiętać, że formatowanie zawsze tworzy nowy łańcuch znaków, a nie zmienia łańcucha znajdującego się po lewej stronie operatora. Ponieważ łańcuchy znaków są niezmienne, musi to działać w taki sposób. Tak jak wcześniej, jeśli chcemy zachować rezultat formatowania, należy przypisać go do zmiennej.

Zaawansowane wyrażenia formatujące

W przypadku zaawansowanego formatowania określonych typów można w wyrażeniach formatujących skorzystać z kodów konwersji przedstawionych w tabeli 7.4. Kody wprowadza się po znaku % w łańcuchu formatującym. Programiści języka C rozpoznają zapewne większość tych kodów, ponieważ formatowanie łańcuchów znaków w Pythonie obsługuje wszystkie często

Tabela 7.4. Kody typów stosowane w wyrażeniach formatujących

Kod	Znaczenie
s	łańcuch znaków (lub dowolny obiekt obsługujący przekształcenie na łańcuch znaków <code>str(x)</code>)
r	To samo co s, jednak wykorzystuje <code>repr</code> , a nie <code>str</code>
c	Znak
d	Liczba dziesiętna (całkowita)
i	Liczba całkowita
u	Równoważne d (przestarzałe, dawniej wymuszało liczbę całkowitą bez znaku)
o	Ósemkowa liczba całkowita
x	Szesnastkowa liczba całkowita
X	To samo co x, jednak wyświetlane wielkimi literami
e	Liczba zmiennoprzecinkowa w formacie wykładniczym, małą literą
E	To samo co e, wyświetlane wielką literą
f	Zmiennoprzecinkowa liczba w zapisie dziesiętnym
F	Zmiennoprzecinkowa liczba w zapisie dziesiętnym
g	Zmiennoprzecinkowe e lub f
G	Zmiennoprzecinkowe E lub F
%	Dosłowne %

używane kody formatów funkcji `sprintf` z języka C (jednak zwraca wynik, zamiast wyświetlać go, jak `printf`). Niektóre z kodów formatów z tabeli udostępniają alternatywne sposoby formatowania tego samego typu. Przykładowo %e, %f oraz %g są różnymi sposobami na formatowanie liczb zmiennoprzecinkowych.

Tak naprawdę cele konwersji w formatowanym łańcuchu znaków po lewej stronie wyrażenia obsługują różne operacje konwersji za pomocą własnej składni. Ogólna struktura celu konwersji wygląda następująco:

`%[(nazwa)][opcje][szerokość][.precyzja]kod_typu`

Znakowe kody typów z tabeli 7.4 umieszcza się na końcu kodu formatującego. Pomiędzy znakiem % i znakiem kodu typu można umieścić dodatkowe informacje formatujące, między innymi: wstawić klucz słownika, podać opcje określające na przykład wyrównanie do lewej strony (-), znak liczbowy (+) i wypełnienie zerami (0), podać całkowitą minimalną długość pola i liczbę cyfr po punkcie dziesiętnym. Szerokość pola oraz precyzja mogą być określone gwiazdką *, co oznacza, że właściwa wartość zostanie przekazana w kolejnym elemencie krotki operandu po prawej stronie operatora % (szerokość pola lub precyzja powinny znaleźć się przed podstawianą wartością).

Składnia formatowania łańcuchów znaków jest w pełni opisana w standardowej dokumentacji Pythona, jednak by zademonstrować przykład jej zastosowania, przyjrzyjmy się kilku przykładom. Poniższy formatuje liczby całkowite w sposób domyślny, a następnie w polu sześcioczątkowym z wyrównaniem do lewej strony i dopełnieniem zerami.

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234 ...001234'
```

Formaty %e, %f oraz %g wyświetlają liczby zmiennoprzecinkowe na różne sposoby, co widać w poniższym kodzie z sesji interaktywnej Pythona (formatowanie %E jest identyczne z %e, z jedną różnicą: kod wykładowicz jest pisany wielką literą).

```
>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+000 | 1.234568 | 1.23457'

>>> '%E' % x
'1.234568E+00'
```

W przypadku liczb zmiennoprzecinkowych można osiągnąć różne dodatkowe efekty w zakresie formatowania dzięki podaniu wyrównania do lewej strony, wypełnienia zerami, znaków liczb, szerokości pola i liczby cyfr po punkcie dziesiętnym. W przypadku prostszych zadań można po prostu przekształcić je na łańcuchy znaków za pomocą wyrażenia formatującego lub zaprezentowanej wcześniej funkcji wbudowanej str.

```
>>> '-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23 | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

Jeśli szerokość znakowa przeznaczona na poszczególne elementy formatowanego łańcucha znaków nie jest znana w trakcie pisania kodu (jest za każdym razem wyliczana przez program), można w kodzie formatującym użyć znaku *, co spowoduje, że szerokość pola zostanie odczytana z krótki danych po prawej stronie operatora % — w poniższym przykładzie liczba 4 w krotce określa precyzję trzeciej liczby.

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

Warto poeksperymentować z tymi przykładami i operacjami, aby oswoić się z ich możliwościami i sposobem użycia.

Wyrażenia formatujące z użyciem słownika

Formatowanie łańcuchów znaków pozwala również, by cele konwersji po lewej stronie odnosili się do kluczów słownika znajdującego się po prawej stronie w celu pobrania odpowiednich wartości. Nie mówiliśmy jeszcze zbyt dużo o słownikach, dlatego poniżej znajduje się przykład, który pomoże zilustrować podstawy tych obiektów.

```
>>> "%(n)d %(x)s" % {"n":1, "x":"mielonka"}
'1 mielonka'
```

W powyższym kodzie (n) i (x) w formatowanym łańcuchu znaków odnoszą się do kluczów literału słownika znajdującego się po prawej stronie i służą do pobrania towarzyszących im wartości. Programy generujące tekst, takie jak HTML czy XML, często korzystają z tej techniki. Można dzięki niej zbudować słownik wartości i zastąpić je wszystkie naraz za pomocą jednego wyrażenia formatującego, wykorzystującego referencje oparte na kluczach słownika.

```
>>> reply = """
    Witamy...
    Witaj %(name)s!
    # Szablon dopasowania
```

```
Twój wiek to %(age)s lat.  
"""  
  
>>> values = {'name': 'Amadeusz', 'age': 40}      # Słownik wartości do podstawienia  
>>> print reply % values                         # Operacja podstawienia
```

```
Witamy...  
Witaj Amadeusz!  
Twój wiek to 40 lat.
```

Ta sama sztuczka jest również wykorzystywana w połączeniu z wbudowaną funkcją `vars`, która zwraca słownik zawierający wszystkie zmienne istniejące w miejscu jej wywołania.

```
>>> food = 'mielonka'  
>>> age = 40  
>>> vars()  
{'food': 'mielonka', 'age': 40, ...więcej... }
```

Kiedy użyjemy `vars` po prawej stronie wyrażenia formatującego, w formatowanym łańcuchu można odnosić się do zmiennych po ich nazwach (czyli między innymi kluczach słownika).

```
>>> "%(age)d %(food)s" % vars()  
'40 mielonka'
```

Słowniki zostaną omówione bardziej szczegółowo w rozdziale 8. Przykłady konwersji łańcuchów znaków na szesnastkowe i ósemkowe łańcuchy liczbowe za pomocą kodów formatujących `%x` i `%o` znajdują się w rozdziale 5.

Metoda format

Jak wspomniałem wcześniej, w Pythonie 2.6 i 3.0 został wprowadzony nowy sposób formatowania łańcuchów znaków, uznawany przez niektórych programistów za bardziej zgodny z duchem Pythona. W przeciwieństwie do wyrażeń formatujących metody formatujące nie wykorzystują składni zapożyczonej z funkcji `printf` języka C i w założeniu miały być bardziej czytelne i jednoznaczne. Z drugiej strony, nowa technika wykorzystuje jednak pewne koncepcje zastosowane w funkcji `printf`, jak kody typów i specyfikacje formatowania. Co więcej, technika ta w znacznym stopniu dubluje możliwości wyrażeń formatujących (czasem nawet wymagając zastosowania większej ilości kodu do realizacji tych samych operacji) i w zaawansowanych zastosowaniach bywa nie mniej skomplikowana w użyciu. Z tego powodu trudno jednoznacznie polecić jedną z dostępnych opcji formatowania, dlatego programiści powinni po prostu zrozumieć podstawy każdej z nich i podjąć decyzję samodzielnie.

Podstawy

W Pythonie 2.6 i 3.0 obiekty znakowe posiadają metodę `format`, wykonywaną na formatującym łańcuchu znaków używanym w charakterze szablonu podstawiania. Metoda ta akceptuje dowolną liczbę argumentów, których wartości są podstawiane w szablonie. W formatującym łańcuchu znaków stosuje się kody podstawienia w postaci nawiasów klamrowych zawierających odwołanie do argumentu po pozycji (na przykład `{1}`) lub po nazwie (na przykład `{food}`). Argumenty funkcji szczegółowo omówimy w rozdziale 18., a w tym momencie wystarczy nam wiedzieć, że mogą one być przekazywane przez pozycję (argumenty pozycyjne) lub przez nazwę (argumenty ze słowami kluczowymi). Dzięki elastycznej naturze Pythona liczba argumentów przekazanych

do funkcji lub metody może być nieograniczona, co pozwala metodzie `format` implementować dość elastyczne możliwości przekazywania wartości do podstawienia. Poniższy listing działa w Pythonie 2.6 i 3.0.

```
>>> template = '{0}, {1} and {2}'                                # Podstawianie pozycyjne
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{motto}, {pork} and {food}'                      # Podstawianie po słowie kluczowym
>>> template.format(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

>>> template = '{motto}, {0} and {food}'                          # Obydwie techniki
>>> template.format('ham', motto='spam', food='eggs')
'spam, ham and eggs'
```

Oczywiście metodę `format` można wykonać również na literale znakowym bez konieczności tworzenia zmiennej, do szablonu mogą być podstawiane dane dowolnych typów.

```
>>> '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
'3.14, 42 and [1, 2]'
```

Podobnie jak w przypadku wyrażeń formatujących z operatorem `%`, metoda `format` tworzy i zwraca nowy łańcuch znaków, który można wyświetlić lub przypisać zmiennej w celu dalszego użycia (jak pamiętamy, łańcuchy znaków są niezmienne, zatem format *musi* tworzyć nowy obiekt znakowy). Formatowanie znaków jest stosowane nie tylko do wyświetlania:

```
>>> X = '{motto}, {0} i {food}'.format(42, motto=3.14, food=[1, 2])
>>> X
'3.14, 42 i [1, 2]'

>>> X.split(' i ')
['3.14', '42', '[1, 2']]

>>> Y = X.replace('i', 'ale pod żadnym pozorem nie')
>>> Y
'3.14, 42 ale pod żadnym pozorem nie [1, 2]'
```

Użycie kluczy, atrybutów i przesunięć

Podobnie jak wyrażenia formatujące z użyciem operatora `%`, metoda `format` obsługuje parametry formatowania pozwalające na obsługę bardziej zaawansowanych sytuacji. Na przykład szablony formatujące mogą wykorzystywać nazwy atrybutów i kluczy słowników. Jak w składni Pythona, nawiasy kwadratowe służą do określania nazw kluczy słownika, a nazwa po kropce określa nazwę atrybutu — wszystko to w ramach elementu podstawianego do szablonu po indeksie lub słowie kluczowym. Poniższy listing prezentuje kilka sposobów użycia tych możliwości. Pierwszy przykład wykorzystuje wartość klucza `'mielonka'` przekazanego słownika oraz atrybut `platform` załadowanego wcześniej modułu `sys`. Drugi przykład wykonuje to samo zadanie, ale obiekty są podstawiane po słowie kluczowym zamiast po indeksie.

```
>>> import sys

>>> 'Mój {1[spam]} ma zainstalowany system {0.platform}'.format(sys, {'spam': 'laptop'})
'Mój laptop ma zainstalowany system win32'

>>> 'Mój {config[spam]} ma zainstalowany system {sys.platform}'.format(sys=sys,
    ↴config={'spam': 'laptop'})
'Mój laptop ma zainstalowany system win32'
```

Nawiasy kwadratowe w szablonach formatujących mogą również służyć do określania indeksów list (i innych sekwencji), ale działają tylko zwykłe indeksy dodatnie, zatem ta możliwość nie jest tak ogólna, jak można by oczekwać. Podobnie jak w przypadku wyrażeń formatujących z operatorem %, w celu uzyskania dostępu do ujemnych indeksów lub wycinków albo użycia innych skomplikowanych wyrażeń należy wykonać je poza szablonem formatowania. W poniższym przykładzie warto zwrócić uwagę na użycie wyrażenia *parts, które służy do rozpakowania elementów krotki do indywidualnych argumentów, co zostało szczegółowo omówione w rozdziale 18.

```
>>> somelist = list('JAJKO')
>>> somelist
['J', 'A', 'J', 'K', 'O']

>>> 'pierwszy={0[0]}, trzeci={0[2]}'.format(somelist)
'pierwszy=J, trzeci=J'

>>> 'pierwszy={0}, ostatni={1}'.format(somelist[0], somelist[-1]) # [-1] w szablonie
# formatującym nie działa
'pierwszy=J, ostatni=0'

>>> parts = somelist[0], somelist[-1], somelist[1:3] # [1:3] w szablonie nie zadziała
>>> 'pierwszy={0}, ostatni={1}, środkowy={2}'.format(*parts)
"pierwszy=J, ostatni=0, środkowy=['A', 'J']"
```

Formatowanie specjalizowane

Kolejne podobieństwo do wyrażeń formatujących polega na tym, że w szablonie można wykorzystać dodatkowe parametry pozwalające zmienić detale formatowania. W przypadku metod formatujących po identyfikacji pola podstawienia należy wpisać dwukropek, a następnie deklarację formatowania określającą rozmiar pola, wyrównanie i kod typu. Oto formalna składnia identyfikatora pola podstawienia w szablonie przy użyciu metod formatujących:

```
{nazwa_pola!znacznik_przekształcenia:specyfikacja_formatu}
```

Znaczenie poszczególnych elementów:

- nazwa_pola jest liczbą lub nazwą słowa kluczowego argumentu, po którym może nastąpić opcjonalne odwołanie do nazwy atrybutu .nazwa lub nazwy klucza albo numer indeksu [indeks].
- znacznik_przekształcenia może mieć wartość r, s lub a, co spowoduje, odpowiednio, wywołanie na argumencie funkcji repr, str lub ascii.
- specyfikacja_formatu określa format prezentacji wartości i może zawierać takie szczegóły, jak szerokość pola, wyrównanie, wypełnienie, precyza wartość dziesiętnych itp. Specyfikacja formatu kończy się opcjonalnym kodem typu danych.

Specyfikacja formatu po dwukropku ma następującą składnię (nawiasy klamrowe sygnalizują elementy opcjonalne i nie powinny być stosowane):

```
[[wypełnienie]wyrównanie][znak][#][0][szerokość][.precyzja][kod_typu]
```

Wyrównanie określa się jednym ze znaków: <, >, = lub ^ dla — odpowiednio — wyrównania do lewej, do prawej, wypełnienia po symbolu znaku liczby lub wycentrowania. Specyfikacja formatu może zawierać zagnieżdżone odwołania do nazw parametrów, które służą do dynamicznego określania parametrów (co działa analogicznie do znaku * w wyrażeniach formatujących).

Więcej informacji na temat składni wywołania metody `format` oraz listę obsługiwanych kodów typów można znaleźć w standardowej dokumentacji Pythona. Kody typów prawie dokładnie pokrywają się z analogicznymi kodami dla wyrażeń formatujących przedstawionymi w tabeli 7.4; w przypadku metody `format` dodatkowo obsługiwany jest kod `b` wykorzystywany do prezentowania liczb całkowitych w notacji dwójkowej (co stanowi odpowiednik funkcji wbudowanej `bin`). W szablonach formatujących metody `format` można bezpośrednio używać znaku `%`, a do formatowania liczb całkowitych w notacji dziesiętnej używany jest wyłącznie kod `d` (nie stosuje się i ani `u`).

W poniższym przykładzie `{0:10}` oznacza pierwszy argument pozycyjny, który będzie sformatowany na 10 znakach, natomiast `{0.platform:>10}` oznacza atrybut `platform` tego pierwszego argumentu pozycyjnego, który będzie wyrównany do prawej w polu o szerokości 10 znaków. W przykładzie została użyta funkcja `dict` przekształcająca na słownik argumenty ze słowami kluczowymi. Funkcja ta została użyta również w rozdziale 4., a szczegółowo będzie omówiona w rozdziale 8.

```
>>> '{0:10} = {1:10}'.format('jajo', 123.4567)
'jajo = 123.457'
>>> '{0:>10} = {1:<10}'.format('jajo', 123.4567)
' jajo = 123.457 '
>>> '{0.platform:>10} = {1[item]:<10}'.format(sys, dict(item='laptop'))
' win32 = laptop '
```

Liczby zmienoprzecinkowe obsługują te same kody typów i parametry formatowania co wyrażenia formatujące z operatorem `%`. Na przykład `{2:g}` oznacza trzeci argument sformatowany w domyślnej reprezentacji zmienoprzecinkowej zgodnej z kodem `g`, natomiast `{1:.2f}` spowoduje zastosowanie formatowania typu `f` z dwoma miejscami po przecinku, a `{2:06.2f}` spowoduje zastosowanie pola o szerokości sześciu znaków z wypełnieniem zerami od lewej strony:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'
>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Metoda `format` obsługuje również formatowanie liczb całkowitych w notacjach szesnastkowej, ósemkowej i dwójkowej. W tym przypadku formatowanie jest odpowiednikiem użycia funkcji wbudowanych, odpowiednio: `hex`, `oct` i `bin`.

```
>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)      # Hex, octal, binary
'FF, 377, 11111111'
>>> bin(255), int('11111111', 2), Ob11111111           # Przekształcenie z/do notacji dwójkowej
('0b11111111', 255, 255)
>>> hex(255), int('FF', 16), 0xFF                      # Przekształcenie z/do notacji ósemkowej
('0xff', 255, 255)
>>> oct(255), int('377', 8), 0o377, 0377              # Przekształcenie z/do notacji szesnastkowej
('0377', 255, 255)
```

Parametry formatowania mogą być zakodowane w szablonie formatującym, mogą być też odczytywane z listy argumentów metody `format`. Służy do tego składnia zagnieżdżonych odwołań, która działa analogicznie do składni gwiazdki w wyrażeniach formatujących:

```

>>> '{0:.2f}'.format(1 / 3.0)                                # Parametry zakodowane w szablonie
'0.33'
>>> '%.2f' % (1 / 3.0)
'0.33'

>>> '{0:{1}f}'.format(1 / 3.0, 4)                            # Wartość odczytywana z argumentów
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)                                  # Wyrażenia mają podobny mechanizm
'0.3333'

```

Python 2.6 i 3.0 posiadają analogiczną funkcję `format`, która może być użyta do formatowania pojedynczego elementu. Funkcja `format` ma być alternatywą dla stosowania metody `format`, a jej działanie jest analogiczne do prostych wyrażeń formatujących z operatorem `%`.

```

>>> '{0:.2f}'.format(1.2345)                                # Metoda łańcucha znaków
'1.23'
>>> format(1.2345, '.2f')                                 # Funkcja wbudowana
'1.23'
>>> '%.2f' % 1.2345                                     # Wyrażenie
'1.23'

```

Wbudowana funkcja `format` wywołuje niejawnie metodę `__format__` pierwszego argumentu. Metoda `__format__` jest również niejawnie wywoływana dla każdego argumentu metody `format`. Jednak zarówno funkcja `format`, jak i metoda `format` wymagają większej ilości kodu, co prowadzi nas do kolejnego tematu.

Porównanie z wyrażeniami formatującymi

Jeśli ktoś uważnie przestudiował poprzedni podrozdział, zauważ z pewnością, że metoda `format` przypomina wyrażenia formatujące w zakresie referencji pozycyjnych i kluczy słowni-kowych, szczególnie w przypadku użycia zaawansowanych opcji formatowania. W rzeczywi-stości najczęściej spotykane przypadki użycia łatwiej i krócej jest implementować z użyciem wyrażeń formatujących, szczególnie w kontekście wywołań funkcji `print` i ogólnych wzorców podstawienia `%s`.

```

print('%s=%s' % ('spam', 42))                         # Wyrażenie formatujące w 2.X+
print('{0}={1}'.format('spam', 42))                      # Metoda formatująca w 3.0 (i 2.6)

```

Jak przekonamy się za chwilę, bardziej skomplikowane przypadki wyrównują tę różnicę (skomplikowane zadania są po prostu skomplikowane, niezależnie od użytej metody) i wiele metod formatowania okazuje się nadmiarowymi.

Z drugiej strony, metoda formatująca oferuje liczne potencjalne zalety. Na przykład oryginalne wyrażenie z operatorem `%` nie potrafi obsługiwać słów kluczowych, odwołań do atrybutów i binarnych kodów typów, ale podobne rezultaty łatwo uzyskać, odpowiednio manipulując odwołaniami do kluczy słownika. Aby zaobserwować obszary pokrywania się tych dwóch technik formatowania, przeanalizujmy następujący listing prezentujący wyrażenia formatujące odpowiadające powyższym wywołaniom metody `format`.

```

# Podstawy: użycie % zamiast format()

>>> template = '%s, %s, %s'
>>> template % ('mielonka', 'szynka', 'jajka')                # Podstawienia pozycyjne
'milonka, szynka, jajka'

>>> template = '%(motto)s, %(pork)s i %(food)s'
>>> template % dict(motto='mielonka', pork='szynka', food='jajka') # Podstawianie słownikowe
'milonka, szynka i jajka'

```

```

>>> '%s, %s and %s' % (3.14, 42, [1, 2])                                # Dowolne typy
'3.14, 42 and [1, 2]'

# Użycie kluczy, atrybutów i indeksów

>>> 'Mój %(spam)s ma zainstalowany system %(platform)s' % {'spam': 'laptop',
   ↳'platform': sys.platform}
'Mój laptop ma zainstalowany system win32'

>>> 'Mój %(spam)s ma zainstalowany system %(platform)s' % dict(spam='laptop',
   ↳platform=sys.platform)
'Mój laptop ma zainstalowany system win32'

>>> somelist = list('JAJKA')
>>> parts = somelist[0], somelist[-1], somelist[1:3]
>>> 'first=%s, last=%s, middle=%s' % parts
"pierwsza]", ostatnia=0, środkowe=['A', 'J']"

```

W przypadku bardziej skomplikowanego formatowania obie techniki są zbliżone pod względem komplikacji kodu, ale porównując poniższy listing z analogcznym kodem wykorzystującym metodę format, musimy stwierdzić, że wyrażenia z operatorem % są nieco prostsze i bardziej zrozumiałe.

Zastosowanie specyficznego formatowania

```

>>> '%-10s = %10s' % ('jajo', 123.4567)
'jajo = 123.4567'

>>> '%10s = %-10s' % ('jajo', 123.4567)
' jajo = 123.4567 '

>>> '%(plat)10s = %(item)-10s' % dict(plat=sys.platform, item='laptop')
' win32 = laptop '

```

Liczby zmiennoprzecinkowe

```

>>> '%e, %.3e, %g' % (3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'

```

Notacja szesnastkowa i ósemkowa, ale brak dwójkowej

```

>>> '%x, %o' % (255, 255)
'ff, 377'

```

Metoda format obsługuje kilka zaawansowanych możliwości, których nie potrafią obsłużyć wyrażenia formatujące, ale nawet bardziej zaawansowane formatowania wyglądają praktycznie tak samo skomplikowanie w każdej z technik formatujących. Na przykład poniższy listing w każdej z zastosowanych technik daje te same wyniki, z polami o ustalonej szerokości i określonym wyrównaniem, z zastosowaniem różnych metod dostępu do argumentów.

Obie metody wykorzystują zakodowane odwołania

```

>>> import sys

>>> 'Mój {1[spam]:<8} ma zainstalowany system {0.platform:>8}'.format(sys,
   ↳{'spam': 'laptop'})
'Mój laptop ma zainstalowany system win32'

```

```
>>> 'Mój %(spam)-8s ma zainstalowany system win32 %(plat)8s' % dict(spam='laptop',
    <plat=sys.platform>
'Mój laptop ma zainstalowany system win32'
```

W praktyce w programach rzadko stosuje się odwołania zakodowane w sposób statyczny, częściej tworzony jest kod budujący zestaw podstawień (na przykład dane są gromadzone, a następnie podstawiane w szablonie HTML). Jeśli weźmiemy pod uwagę tego typu wzorce zastosowań, podobieństwo między metodą `format` a wyrażeniami z operatorem `%` okaże się jeszcze wyraźniejsze (jak się dowiemy w rozdziale 18., `**data` w wywoaniu metody jest przykładem specjalnej składni przekazania argumentów ze słowami kluczowymi do wywołania funkcji, dzięki czemu mogą być one odczytywane po nazwach przez metodę `format`).

Budowanie danych i dynamiczne formatowanie

```
>>> data = dict(platform=sys.platform, spam='laptop')

>>> 'Mój {spam:<8} ma zainstalowany system {platform:>8}'.format(**data)
'Mój laptop ma zainstalowany system win32'

>>> 'Mój %(spam)-8s ma zainstalowany system %(platform)8s' % data
'Mój laptop ma zainstalowany system win32'
```

Jak zwykle to społeczność Pythona zdecyduje, czy próbę czasu przetrwają wyrażenia formatujące z operatorem `%`, czy metoda `format`, czy też wykorzystywane będą obie z tych technik. Zalecam przeprowadzenie eksperymentów z tymi mechanizmami, aby rozpoznać oferowane przez nie możliwości, oraz zapoznanie się z dokumentacją standardową Pythona 2.6 i 3.0.



Udoskonalenia metody format w 3.1: Najnowsza wersja Pythona, 3.1, wprowadza do metody `format` obsługę separatora tysięcy dla liczb: pomiędzy kolejnymi grupami cyfr umieszczany jest przecinek. Wystarczy wpisać przecinek przed kodem typu, na przykład:

```
>>> '{0:d}'.format(999999999999)
'999999999999'
>>> '{0:,d}'.format(999999999999)
'999,999,999,999'
```

W Pythonie 3.1 automatycznie odbywa się odwołanie do kolejnych argumentów metody, jeśli ich pozycje nie zostały zadeklarowane jawnie w łańcuchu formatującym. Użycie tej możliwości nieco jednak zmniejsza zalety użycia formatowania, o czym wspomnę więcej w następnym punkcie.

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'

>>> '{:,d} {:,d}'.format(9999999, 8888888)
'9,999,999 8,888,888'

>>> '{:.2f}'.format(296999.2567)
'296,999.26'
```

Niniejsza książka oficjalnie nie omawia Pythona 3.1, zatem powyższe informacje potraktujmy jako rzut oka w przyszłość. Python 3.1 rozwiązuje podstawowe problemy Pythona 3.0 w zakresie wydajności plikowych operacji wejścia-wyjścia, które powodowały, że Python 3.0 nie nadawał się do wielu zastosowań. Więcej szczegółów na temat nowości można znaleźć w informacjach o wydaniu Pythona 3.1. Jeśli chodzi o separator tysięcy w wersjach Pythona wcześniejszych od 3.1, jesteśmy skazani na własne implementacje. Odpowiedni przykład można znaleźć w rozdziale 24.

Po co nam kolejny mechanizm formatujący?

Skoro już poświęciłem sporo miejsca na porównanie dwóch dostępnych technik formatowania łańcuchów znaków, powiniennem wyjaśnić, w jakim właściwie celu programista miałby stosować jeden lub drugi wariant formatowania? Mimo że metoda `format` wymaga czasem napisania większej ilości kodu, posiada kilka zalet:

- ma kilka możliwości, których nie mają wyrażenia formatujące z operatorem %,
- odwołania do podstawianych argumentów są bardziej jawne i czytelne,
- zamiast symbolu operatora używamy czytelnej nazwy metody,
- nie wymaga stosowania osobnej składni dla pojedynczych i wielokrotnych podstawięń.

We współczesnych wersjach Pythona mamy dostęp do obydwu technik formatowania. Wprawdzie wyrażenia formatujące są bardzo powszechnie stosowane, ale, jak się wydaje, metoda `format` z czasem przejmie dominację w tym zakresie. Wybór jednak należy do programistów. Przeanalizujmy zatem podstawowe różnice między tymi technikami.

Dodatkowe możliwości

Metoda `format` obsługuje kilka możliwości nieobsługiwanych przez wyrażenia formatujące, jak formatowanie liczb w notacji dwójkowej oraz (dostępne od Pythona 3.1) separator tysięcy. Dodatkowo bezpośrednio z szablonu formatującego dostępne są atrybuty i wartości słowników po kluczu. Jednak, jak mieliśmy okazję się przekonać, w wyrażeniach formatujących możemy użyć tych możliwości w nieco inny sposób:

```
>>> '{0:b}'.format((2 ** 16) - 1)
'1111111111111111'

>>> '%b' % ((2 ** 16) - 1)
ValueError: unsupported format character 'b' (0x62) at index 1

>>> bin((2 ** 16) - 1)
'0b1111111111111111'

>>> '%s' % bin((2 ** 16) - 1)[2:]
'1111111111111111'
```

W poprzednim punkcie można znaleźć przykład formatowania z użyciem słowników w wyrażeniach formatujących oraz jego porównanie z odwołaniami do kluczów słowników i atrybutów obiektów w metodzie `format`. W praktyce obydwa te podejścia wydają się jedynie różnymi wariacjami na ten sam temat.

Jawne odwołania do wartości

Jednym z przypadków, w których metoda `format` wydaje się doskonalsza, jest podstawianie większej liczby wartości do szablonu formatującego. Przedstawiony w rozdziale 30. przykład `lister.py` wykorzystuje podstawianie sześciu elementów do jednego szablonu i w tym przypadku odwołanie pozycyjne typu {i} wydaje się dawać czytelniejszy kod w porównaniu do podstawiania z użyciem %s:

```
'\n%s<Klasa %s, adres %s:\n%s%s%s>\n' % (...), # Wyrażenie
'\n{0}<Klasa {1}, adres {2}:\n{3}{4}{5}>\n'.format(...), # Metoda
```

Z drugiej strony, wykorzystując podstawienia z użyciem słownika w wyrażeniach %, można w znaczącym stopniu zniwelować tę różnicę. Trzeba też pamiętać, że powyższy przykład należy do najbardziej skrajnych przypadków komplikacji szablonów formatujących i nie jest powszechnie stosowany w praktyce. Bardziej typowe przypadki użycia są znacznie prostsze. Co więcej, w Pythonie 3.1 numerowanie odwołań do argumentów metody `format` w szablonie formatującym staje się opcjonalne, co sugeruje, że zaleta zwiększenia czytelności może nie być w ogóle wykorzystywana.

```
C:\misc> C:\Python31\python
>>> 'Zawsze {0} na {1} z {2}'.format('patrz', 'życie', 'humorem')
'Zawsze patrz na życie z humorem'
>>>
>>> 'Zawsze {} na {} z {}'.format('patrz', 'życie', 'humorem') # Python 3.1+
'Zawsze patrz na życie z humorem'
>>>
>>> 'Zawsze %s na %s z %s' % ('patrz', 'życie', 'humorem')
'Zawsze patrz na życie z humorem'
```

Tego typu wykorzystanie automatycznego indeksowania argumentów wprowadzonych w wersji 3.1 wydaje się negować największą zaletę metody `format`. Porównajmy przykłady formatowania liczb zmiennoprzecinkowych — wykorzystanie wyrażenia formatującego pozwala użyć krótszy i czytelniejszy kod:

```
C:\misc> C:\Python31\python
>>> '{0:f}, {1:.2f}, {2:0.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 03.14'
>>>
>>> '{:f}, {:.2f}, {:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
>>>
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Nazwy metod i uogólnione argumenty

Jeśli automatyczne numerowanie argumentów w szablonie formatującym uznamy za krok wstecz, jedyną wyraźną zaletą metody `format` jest użycie czytelnej dla człowieka nazwy metody w miejsce nic niemówiącego symbolu operatora % oraz brak rozgraniczenia składni między podstawianiem jednej i większej liczby wartości. Pierwsza zmiana w pierwszych kontaktach z językiem z pewnością wpływa na lepsze zrozumienie kodu (gdy początekający zobaczy nazwę `format`, łatwiej będzie mu się domyślić, co robi ta metoda, niż w przypadku znaczka %), ale ta zaleta jest dość subiektywna.

Druga zaleta metody `format` jest już bardziej istotna: w przypadku wyrażeń formatujących podstawienie pojedynczej wartości pozwala podać ją bezpośrednio po operatorze %, ale w przypadku większej liczby elementów musimy użyć krotki.

```
>>> '%.2f' % 1.2345
'1.23'
>>> '%.2f %s' % (1.2345, 99)
'1.23 99'
```

Z technicznego punktu widzenia wyrażenia formatujące akceptują pojedynczą wartość do podstawienia lub krotkę zawierającą większą liczbę wartości. Co gorsza, w związku z tym, że wartość do formatowania można ująć w krotkę jednoelementową, możemy też podstawić całą krotkę, zagnieżdżając ją w innej krotce:

```
>>> '%s' % 1.23
'1.23'
>>> '%s' % (1.23,)
'1.23'
>>> '%s' % ((1.23,),)
'(1.23,)'
```

Metoda format ujednolica składnię, stosując uogólniony mechanizm argumentów, który działa dla obiektów dowolnych typów:

```
>>> '{0:.2f}'.format(1.2345)
'1.23'
>>> '{0:.2f} {1}'.format(1.2345, 99)
'1.23 99'
>>> '{0}'.format(1.23)
'1.23'
>>> '{0}'.format((1.23,))
'(1.23,)'
```

Takie podejście może być czytelniejsze dla początkujących i pozwoli uniknąć błędów. Jednak nadal to niewielki szczegół: jeśli nabierzemy nawyku ujmowania operandów wyrażenia % w krotkę, nawet dla pojedynczych podstawień, uzyskamy tę samą spójność składniową co w przypadku metody format. Co więcej, metoda powoduje dodatkowy koszt w postaci zwiększenia ilości kodu, jaki musi wprowadzić programista dla uzyskania tego samego efektu. Biorąc dodatkowo pod uwagę, że wyrażenia formatujące były używane przez całą dotychczasową historię Pythona, nie ma pewności, czy pytanie postawione w następnym punkcie uzyska odpowiedź twierdzącą — innymi słowy: czy niewielkie korzyści uzyskane na skutek wprowadzenia nowego narzędzia (które efektywnie niewiele różni się od starego) uzasadniają popuszczenie zgodności wstecz z gigantyczną ilością kodu.

Czy wyrażenia formatujące zostaną w przyszłości wycofane z Pythona?

Jak wspomniałem wcześniej, istnieje sugestia, że programiści Pythona w przyszłych wersjach języka wycofają z użycia wyrażenia formatujące z operatorem % na korzyść użycia metody format. W rzeczywistości istnieje nawet wzmianka o takich planach w standardowym podręczniku Pythona.

Jak do tej pory taka zmiana oczywiście nie nastąpiła i dla programistów Pythona 2.6 i 3.0 dostępne są obie techniki formatujące (a tymi wersjami zajmuje się niniejsza książka). Obie metody będą też dostępne w następnej wersji Pythona, 3.1, zatem w najbliższej przyszłości nie dojdzie do wycofania obsługi wyrażeń formatujących. Co więcej, z faktu powszechnego wykorzystania wyrażeń formatujących w istniejącym kodzie Pythona wynika, iż programiści powinni znać obie techniki.

Jeśli jednak kiedyś dojdzie do wycofania obsługi wyrażeń formatujących, programiści chcący uruchomić swój stary kod w nowej wersji Pythona będą zmuszeni do przekodowania wszystkich użyć operatora % w sposób opisany w tej książce. W tym miejscu zaryzykuję dygresję: mam nadzieję, że tego typu decyzje dotyczące zmian w języku będą oparte raczej na powszechnych praktykach programistycznych, a nie na kaprysach twórców języka, szczególnie jeśli weźmiemy pod uwagę fakt, że niekompatybilne zmiany, jakie miał prawo wprowadzić Python 3.0, mamy już za sobą. Szczerze mówiąc, decyzja o wycofaniu wyrażeń formatujących będzie polegała na zamianie jednej skomplikowanej techniki na inną skomplikowaną technikę i to

w dodatku taką, która daje niewiele korzyści w porównaniu do tej, którą zastępuje! Programiści zainteresowani zachowaniem zgodności swojego kodu z nowymi wersjami Pythona powinni jednak śledzić dalszy rozwój wypadków w tym zakresie.

Generalne kategorie typów

Po omówieniu pierwszych obiektów kolekcji Pythona — łańcuchów znaków — zatrzymajmy się na chwilę w celu zdefiniowania kilku ogólnych koncepcji związanych z typami, które będą miały zastosowanie do wszystkich typów, z jakimi od teraz będziemy się spotykać. Jeśli chodzi o typy wbudowane, okazuje się, że pewne operacje działają tak samo dla wszystkich typów z jednej kategorii, dlatego większość koncepcji wystarczy zdefiniować raz. Dotychczas omówiliśmy jedynie liczby i łańcuchy znaków, jednak ponieważ są one reprezentatywne dla dwóch z trzech ważnych kategorii typów w Pythonie, już teraz wiemy o innych typach więcej, niż nam się wydaje.

Typy z jednej kategorii współdzielą zbiory operacji

Jak przekonaliśmy się wcześniej, łańcuchy znaków to sekwencje niezmienne — nie można ich zmieniać w miejscu („niezmienne”) i są kolekcjami uporządkowanymi po pozycji, do których dostęp odbywa się po wartości przesunięcia („sekwencje”). Tak się składa, że wszystkie sekwencje omawiane w tej części książki reagują na te same operacje na sekwencjach, jakie zostały zaprezentowane w niniejszym rozdziale jako działające na łańcuchach znaków. Dotyczy to na przykład konkatenacji, indeksowania czy wykonywania iteracji. Z bardziej formalnego punktu widzenia w Pythonie istnieją trzy kategorie typów (i powiązanych z nimi operacji).

Liczby (całkowite, zmiennoprzecinkowe, stałoprzecinkowe, ułamki, inne)

Obsługują dodawanie, mnożenie i tym podobne.

Sekwencje (łańcuchy znaków, listy, krotki)

Obsługują indeksowanie, wycinki, konkatenację i tym podobne.

Odwzorowania (słowniki)

Obsługują indeksowanie po kluczu i tym podobne.

Zbiory stanowią specyficzną kategorię danych (nie mapują kluczy na wartości i nie są sekwencjami zachowującymi kolejność elementów), ponadto na razie nie omówiliśmy jeszcze odwzorowań (do słowników przejdziemy w kolejnym rozdziale). Pozostałe typy, z jakimi się spotkamy, będą jednak do siebie dość podobne. Na przykład dla dowolnych dwóch obiektów sekwencji X i Y :

- $X + Y$ tworzy nowy obiekt sekwencji o zawartości obu argumentów,
- $X * N$ tworzy nowy obiekt sekwencji o N kopii argumentu X .

Innymi słowy, operacje te działają tak samo na każdym rodzaju sekwencji, w tym na łańcuchach znaków, listach, krotkach i pewnych typach zdefiniowanych przez użytkownika. Jedyną różnicą jest to, że nowy obiekt wynikowy ma ten sam typ co argumenty X i Y — po konkatenacji dwóch list otrzymamy nową listę, a nie łańcuch znaków. Indeksowanie, wycinki czy inne operacje na sekwencjach działają w ten sam sposób na wszystkich sekwencjach. To typ przetwarzanego obiektu mówi Pythonowi, które zadanie ma wykonać.

Typy zmienne można modyfikować w miejscu

Klasyfikacja typów na zmienne i niezmienne jest ważnym ograniczeniem, o którym należy pamiętać, ponieważ nowi użytkownicy często mają z nią problemy. Jeśli obiekt jest typu niezmienneego, nie można zmodyfikować jego wartości w miejscu. Kiedy spróbujemy to zrobić, Python zwróci błąd. Zamiast tego konieczne jest wykonanie kodu tworzącego nowy obiekt z nową wartością. Do podstawowych typów danych w Pythonie zaliczamy:

Typy niezmienne (liczby, łańcuchy znaków, krotki, zamrożone zbiory)

Typy niezmienne nie obsługują modyfikacji obiektów w miejscu, możemy jednak używać wyrażeń przekształcających obiekty w inne, tworząc nowe obiekty, i przypisywać te nowe wartości na poprzednią nazwę.

Typy zmienne (listy, słowniki, zbiory, bytearray)

Typy zmienne mogą być modyfikowane w miejscu z użyciem operacji nietworzących nowych obiektów. Można je też kopować na nowe obiekty, jednak modyfikowanie w miejscu daje możliwość bezpośredniej modyfikacji istniejących danych.

Zazwyczaj typy niezmienne dają swego rodzaju gwarancję stałości i integralności, ponieważ obiekt nie może być zmieniony przez inną część programu. Przypomnienie, dlaczego jest to ważne, można znaleźć w omówieniu współdzielonych referencji do obiektów w rozdziale 6. W kolejnym rozdziale przekonamy się, w jaki sposób listy, słowniki i krotki biorą udział w określaniu kategorii typów.

Podsumowanie rozdziału

W niniejszym rozdziale dogłębnie zapoznaliśmy się z obiektem łańcucha znaków. Dowiedzieliśmy się, czym są literały łańcuchów znaków, i omówiliśmy operacje na tym typie danych, w tym wyrażenia działające na sekwencjach, formatowanie łańcuchów, wywoływanie ich metod oraz formatowanie łańcuchów znaków z użyciem wyrażeń i metody formatującej. Przy okazji omówiliśmy szczegółowo wiele różnych koncepcji, takich jak wycinki, wywołania metod i bloki łańcuchów znaków umieszczone w potrójnych apostrofach lub cudzysłowach. Zdefiniowaliśmy również pewne podstawowe dla różnych typów koncepcje. Sekwencje współdzielą na przykład cały zbiór operacji.

W kolejnym rozdziale będziemy kontynuować omawianie typów obiektów Pythona, przechodząc do najbardziej uniwersalnych kolekcji obiektów tego języka, czyli list i słowników. Jak się niebawem okaże, wiele z informacji, które uzyskaliśmy tutaj, będzie miało zastosowanie również do tych typów. Jak wspomniałem wcześniej, w ostatniej części książki wróćmy do pythonowego modelu łańcuchów znaków przy okazji analizy użycia tekstów i binarnych danych w formacie Unicode, które są przydatne dla części programistów Pythona. Najpierw jednak czas na krótki quiz podsumowujący niniejszy rozdział.

Sprawdź swoją wiedzę — quiz

1. Czy metody łańcuchów `find` można użyć do przeszukania listy?
2. Czy wyrażenie łańcuchów znaków z wycinkiem może zostać zastosowane do listy?

3. W jaki sposób można przekonwertować znak do jego kodu liczbowego ASCII? W jaki sposób można wykonać odwrotną operację, konwertując kod liczbowy na znak?
4. W jaki sposób można w Pythonie zmienić łańcuch znaków?
5. Mając łańcuch znaków `S` o wartości "j,aj,a", należy podać dwa sposoby dokonania ekstrakcji dwóch środkowych znaków.
6. Ile znaków znajduje się w łańcuchu "a\nb\x1f\000d"?
7. Po co można by użyć modułu `string` zamiast wywołania metod łańcuchów znaków?

Sprawdź swoją wiedzę — odpowiedzi

1. Nie, ponieważ metody są zawsze specyficzne dla typu, co oznacza, że działają one wyłącznie na jednym typie danych. Wyrażenia typu `X+Y` i funkcje wbudowane, jak `len(X)`, są jednak uniwersalne i mogą działać na wielu typach obiektów. W tym przypadku na przykład wyrażenie testu przynależności `in` ma podobne działanie jak metoda `find` łańcucha znaków, ale może być użyte do przeszukiwania łańcuchów znaków oraz list. W Pythonie 3.0 podjęto próbę pogrupowania metod według kategorii (na przykład zmienne typy `list` i `bytearray` posiadają podobne zbiory metod), jednak metody są bardziej związane z typami niż operatory.
2. Tak. W przeciwnieństwie do metod wyrażenia są uniwersalne i mają zastosowanie do wielu typów. W tym przypadku wyrażenie z wycinkiem jest tak naprawdę operacją na sekwencjach — działa na każdym typie obiektu sekwencji, w tym na łańcuchach znaków, listach oraz krotkach. Jedyną różnicą jest to, że kiedy tworzy się wycinek listy, z powrotem otrzymuje się listę.
3. Wbudowana funkcja `ord(S)` konwertuje jednoznakowy łańcuch na kod liczbowy znaku. Funkcja `chr(I)` przekształca z kolei kod liczbowy z powrotem na łańcuch znaków.
4. Łańcuchy znaków są niezmienne, zatem nie można ich modyfikować w miejscu. Można jednak uzyskać podobny efekt, tworząc nowy łańcuch — za pomocą konkatenacji, wycinka, wykonania wyrażenia formatującego czy użycia metody, takiej jak `replace` — a następnie przypisując wynik z powrotem do oryginalnej nazwy zmiennej.
5. Można utworzyć wycinek z tego łańcucha znaków za pomocą wyrażenia `S[2:4]` lub podzielić łańcuch w miejscu wystąpienia przecinka i zindeksować go za pomocą `S.split(',')[1]`. Warto oba rozwiązania wypróbować w sesji interaktywnej Pythona.
6. Sześć. Łańcuch znaków "a\nb\x1f\000d" zawiera następujące bajty: a, nowy wiersz (`\n`), b, binarne 31 (w postaci szesnastkowej ze znakiem ucieczki — `\x1f`), binarne 0 (w postaci ósemkowej ze znakiem ucieczki — `\000`) oraz d. By to sprawdzić, należy przekazać łańcuch znaków do funkcji `len`, a także wyświetlić wynik zastosowania funkcji `ord` na każdym ze znaków w celu zobaczenia ich prawdziwej wartości bajtowej. Więcej informacji na ten temat znajduje się w tabeli 7.2.
7. Obecnie nigdy nie należy używać modułu `string` w miejscu wywołania metod łańcuchów znaków — sposób ten jest przestarzały, a wywołania tego typu całkowicie usunięto w Pythonie 3.0. Jedynym powodem uzasadniającym korzystanie z modułu `string` są jego inne narzędzia, takie jak zdefiniowane stałe. Jego użycie można również nadal znaleźć w starych programach w Pythonie.

Listy oraz słowniki

Niniejszy rozdział prezentuje typy obiektów list oraz słowników — oba są kolekcjami innych obiektów. Te dwa typy stanowią siłę napędową prawie wszystkich skryptów napisanych w Pythonie. Jak zobaczymy, oba są też zdziwiająco elastyczne — można je zmieniać w miejscu, rozszerzać i zmniejszać na żądanie; obydwa mogą również zawierać dowolne inne typy obiektów i mogą być osadzane w innych typach obiektów. Korzystając z tych typów, w skryptach można tworzyć i przetwarzać dowolnie bogate struktury informacji.

Listy

Kolejnym przystankiem w naszym omówieniu wbudowanych typów obiektów Pythona są *listy*. Listy są najbardziej elastycznym typem obiektu uporządkowanej kolekcji. W przeciwieństwie do łańcuchów znaków listy mogą zawierać dowolne rodzaje obiektów — liczby, łańcuchy znaków, a nawet inne listy. W przeciwieństwie do łańcuchów znaków listy można również modyfikować w miejscu poprzez przypisanie do pozycji przesunięcia i wycinków, wywołania metod list czy instrukcje usuwające elementy. Są to obiekty *zmienne*.

Listy w Pythonie wykonują większość pracy na strukturach danych kolekcji, którą w językach niższego poziomu, jak na przykład C, implementować trzeba ręcznie. Poniżej znajduje się przegląd najważniejszych właściwości list.

Listy są uporządkowanymi kolekcjami dowolnych obiektów

Z funkcjonalnego punktu widzenia listy są miejscami zbierającymi inne obiekty, przez co można je traktować jak grupy. Zachowują one również uporządkowanie elementów od lewej do prawej strony (są zatem sekwencjami).

Dostęp do elementów list można uzyskać za pomocą pozycji przesunięcia

Tak jak w przypadku łańcuchów znaków, element listy można z niej pobrać, indeksując listę w pozycji przesunięcia obiektu. Ponieważ elementy listy są uporządkowane pozycyjnie, możliwe jest wykonywanie wycinków czy konkatenacja.

Listy mają zmienną długość, są niejednorodne i można je dowolnie zagnieżdżać

W przeciwieństwie do łańcuchów znaków listy mogą rosnąć i kurczyć się w miejscu, czyli ich długość może się zmieniać. Mogą również zawierać dowolny typ obiektów, nie tylko jednoznakowe łańcuchy — są zatem niejednorodne (heterogeniczne). Ponieważ listy mogą zawierać inne skomplikowane obiekty, obsługują również dowolne zagnieżdżanie. W Pythonie możliwe jest tworzenie list list.

Listy należą do zmiennych sekwencji

Jeśli chodzi o przydział do odpowiedniej kategorii, listy można modyfikować w miejscu (są zmienne) i reagują one na wszystkie operacje na sekwencjach wykorzystywane z łańcuchami znaków, takie jak indeksowanie, wycinki i konkatenacja. Operacje na sekwencjach działają na listach w ten sam sposób jak na łańcuchach znaków. Jedyną różnicą jest to, że operacje takie, jak konkatenacja i wycinki po zastosowaniu do list zwracają nowe listy, a nie nowe łańcuchy znaków. Ponieważ listy są zmienne, obsługują również inne operacje, których nie obsługują łańcuchy znaków (na przykład operacje usuwania czy przypisania do indeksu, zmieniające listę w miejscu).

Listy są tablicami referencji do obiektów

Z technicznego punktu widzenia listy zawierają zero lub większą liczbę referencji do innych obiektów. Listy mogą przypominać nam tablice wskaźników (adresów). Pobranie elementu z listy Pythona jest prawie tak szybkie jak zindeksowanie tablicy języka C. Tak naprawdę wewnętrz standardowego interpretera Pythona listy są tablicami z języka C, a nie połączonymi strukturami. Jak jednak wspominaliśmy w rozdziale 6., Python zawsze śledzi referencję do obiektu za każdym jej użyciem, dlatego program ma do czynienia jedynie z obiektemi. Kiedy przypiszemy obiekt do komponentu struktury danych czy nazwy zmiennej, Python zawsze przechowuje referencję do samego obiektu, a nie jego kopię (o ile tego w jawnym sposobie nie zażądamy).

W tabeli 8.1 przedstawiono reprezentatywną listę często wykorzystywanych operacji na listach. Jak zawsze więcej informacji można znaleźć w dokumentacji biblioteki standardowej Pythona lub wywołując w sesji interaktywnej funkcje `help(list)` bądź `dir(list)` w celu uzyskania pełnej listy metod list. Do funkcji tych można przekazać zarówno prawdziwą, istniejącą listę, jak i słowo `list`, będące nazwą typu danych.

Kiedy listę zapisze się jako wyrażenie z literałem, zostaje ona zakodowana jako rozdzielona przecinkami seria obiektów (a tak naprawdę wyrażeń zwracających obiekty) w nawiasach kwadratowych, oddzielone przecinkami. Drugi wiersz z tabeli 8.1 przypisuje na przykład zmienną `L` do listy czteroelementowej. Osadzona lista kodowana jest jako seria elementów w nawiasach kwadratowych (rząd trzeci), natomiast pusta lista to po prostu para nawiasów kwadratowych niezawierająca niczego (pierwszy wiersz).¹

Wiele z operacji z tabeli 8.1 powinno wyglądać znajomo, ponieważ są one tymi samymi działaniami, jakie widzieliśmy przy omawianiu łańcuchów znaków — jak indeksowanie, konkatenacja czy iteracje. Listy reagują również na wywołania metod specyficznych dla tego typu danych (które udostępniają narzędzia takie, jak sortowanie, odwracanie, dodawanie nowych elementów na końcu) oraz operacje modyfikujące te obiekty w miejscu (usuwające elementy list, przypisujące coś do indeksów oraz wycinków). Listy otrzymują narzędzia obsługujące operacje modyfikacji, ponieważ są zmiennym typem obiektów.

¹ W praktyce niewiele list w programach przetwarzających listy wygląda w taki sposób. Częściej widuje się kod przetwarzający listy skonstruowane dynamicznie (w momencie wykonywania). Choć ważne jest opanowanie składni literałów, większość struktur danych w Pythonie budowana jest przez działający program w czasie wykonywania.

Tabela 8.1. Często stosowane literały list oraz operacje na tym typie danych

Operacja	Interpretacja
L = []	Pusta lista
L = [0, 1, 2, 3]	Cztery elementy — indeksy od 0 do 3
L = ['abc', 'def', 'ghi']	Zagnieżdżone podlisty
L = list('mielonka')	Lista elementów obiektu iterowanego
L = list(range(-4, 4))	Lista kolejnych liczb całkowitych
L[i]	Indeks, indeks indeksu, wycinek, długość
L[i:j]	
L[i:j]	Konkatenacja, powtórzenie
len(L)	
L1 + L2	Iteracja, przynależność
L * 3	Metody: dodawanie elementów
for x in L: print(x)	
3 in L	Metody: przeszukanie
L.append(4)	
L.extend([5,6,7])	Metody: sortowanie, odwrócenie itp.
L.insert(I,X)	
L.index(1)	Zmniejszenie listy
L.count(X)	
L.sort()	Przypisanie do indeksu, przypisanie do wycinka
L.reverse()	Listy składane (rozdziały 14. oraz 20.)
del L[k]	
del L[i:j]	
L.pop()	
L.remove(2)	
L[i:j] = []	
L[i] = 1	
L[i:j] = [4,5,6]	
L = [x**2 for x in range(5)]	
List(map(ord, 'mielonka'))	

Listy w akcji

Chyba najlepszym sposobem na zrozumienie list jest zobaczenie ich w akcji. Zajmijmy się znowu prostymi kodami z sesji interaktywnej, które pomogą nam zilustrować operacje z tabeli 8.1.

Podstawowe operacje na listach

Listy reagują na operatory + oraz * podobnie jak łańcuchy znaków. Tutaj również oznaczają one konkatenację i powtórzenie, jednak rezultatem będzie nowa lista, a nie nowy łańcuch znaków. Listy obsługują wszystkie uniwersalne operacje na sekwencjach, jakie w poprzednim rozdziale wypróbowaliśmy na łańcuchach znaków.

```
% python
>>> len([1, 2, 3])                                # Długość
3
>>> [1, 2, 3] + [4, 5, 6]                         # Konkatenacja
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                                  # Powtórzenie
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Choć operator + działa tak samo dla list iłańcuchów znaków, należy pamiętać, że oczekuje on sekwencji tego samego typu po obu stronach — w przeciwnym razie po wykonaniu kodu otrzymamy błąd typu. Nie można zatem dokonać konkatenacji listy iłańcucha znaków bez wcześniejszej konwersji listy dołańcucha znaków (dzięki apostrofom lewym, str lub formaturze z %) albołańcucha do listy (dzięki wbudowanej funkcji list).

```
>>> str([1, 2]) + "34"                           # To samo co "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")                          # To samo co [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

Iteracje po listach i składanie list

Listy mogą być używane we wszystkich operacjach na sekwencjach, jakie poznaliśmy w poprzednim rozdziale przy okazji omawiania ciągów znaków, dotyczy to również narzędzi iteracyjnych:

```
>>> 3 in [1, 2, 3]                               # Przynależność
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')
...
1 2 3
```

Więcej informacji na temat iteracji w pętli for oraz funkcji wbudowanej range przekażę w rozdziale 13., ponieważ są to zagadnienia związane ze składnią instrukcji. W skrócie: pętla for pozwala na przetwarzanie elementów sekwencji po kolej, wykonując w każdym przebiegu jedną lub większą liczbę instrukcji.

Ostatnie elementy tabeli 8.1, czyli listy składane i wywołania funkcji map, są omówione w rozdziale 14. oraz bardziej szczegółowo w rozdziale 20. Ich działanie jest jednak dość proste: jak wspomniałem w rozdziale 4., listy składane są sposobem budowania nowej listy przez wywołanie wyrażenia na elementach sekwencji. W filozofii działania listy składane przypominają pętle for:

```
>>> res = [c * 4 for c in 'JAJKO']            # Lista składana
>>> res
['AAAA', 'AAAA', 'AAAA', 'KKKK', '0000']
```

To wyrażenie jest właściwie równoważne pętli for składającej listę wyników, ale jak będziemy mieli okazję przekonać się w dalszych rozdziałach, listy składane pisze się prościej i działają one szybciej.

```
>>> res = []
>>> for c in 'JAJKO':
...     res.append(c * 4)                         # Odpowiednik listy składanej
...
>>> res
['AAAA', 'AAAA', 'AAAA', 'KKKK', '0000']
```

Jak również wspominałem w rozdziale 4., funkcja wbudowana `map` ma podobne działanie, ale zamiast wyrażeń na elementach sekwencji wywoływane są funkcje, a wyniki zwracane są w postaci nowej sekwencji:

```
>>> list(map(abs, [1, 2, 0, 1, 2]))          # Wywołanie funkcji map na sekwencji  
[1, 2, 0, 1, 2]
```

Na tym etapie książki nie jesteśmy jeszcze gotowi na przekazanie pełnych informacji dotyczących iteratorów, zatem odłożymy to zagadnienie do późniejszych rozdziałów. W dalszej części niniejszego rozdziału wróćmy jednak na chwilę do tego zagadnienia przy okazji podobnych wyrażeń słowników składanych.

Indeksowanie, wycinki i macierze

Ponieważ listy są sekwencjami, indeksowanie i wycinki działają w ten sam sposób dla list, jak i dla łańcuchów znaków. Wynikiem zindeksowania listy jest jednak dowolny typ obiektu znajdujący się na pozycji o podanej wartości przesunięcia, natomiast wycinek listy zawsze zwraca nową listę.

```
>>> L = ['mielonka', 'Mielonka', 'MIELONKA!']  
>>> L[2]                                         # Wartości przesunięcia rozpoczynają się od 0  
'MIELONKA!'  
>>> L[-2]                                         # Wartość ujemna: odliczamy od końca  
'Mielonka'  
>>> L[1:]                                         # Wycinek pobiera części listy  
['Mielonka', 'MIELONKA!']
```

Jedna uwaga: ponieważ wewnątrz list można zagnieździć inne listy (oraz inne typy), czasami w celu wejścia w głęb struktury danych niezbędne będzie połączenie ze sobą kilku operacji indeksowania. Jednym z łatwiejszych sposobów reprezentowania w Pythonie macierzy (tablic wielowymiarowych) są listy z zagnieżdżonymi podlistami. Poniżej widać prostą dwuwymiarową tablicę 3×3 opartą na listach.

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Jeden indeks pozwala nam pobrać cały wiersz (a tak naprawdę zagnieżdżoną sublistę), natomiast dwa indeksy — pojedynczy element tego wiersza.

```
>>> matrix[1]  
[4, 5, 6]  
>>> matrix[1][1]  
5  
>>> matrix[2][0]  
7  
>>> matrix = [[1, 2, 3],  
...             [4, 5, 6],  
...             [7, 8, 9]]  
>>> matrix[1][1]  
5
```

W powyższym kodzie widać, że listy w naturalny sposób mogą się rozciągać na kilka wierszy, jeśli tego chcemy, ponieważ znajdują się one w parze nawiasów kwadratowych (więcej informacji na temat składni w kolejnej części książki). W dalszej części niniejszego rozdziału zobaczymy macierz utworzoną za pomocą słownika. W przypadku większych zadań z dziedziny programowania numerycznego przyda się omówione w rozdziale 5. rozszerzenie NumPy, które również obsługuje macierze.

Modyfikacja list w miejscu

Ponieważ listy są typem zmiennym, obsługują operacje zmieniające obiekt listy *w miejscu*. Oznacza to, że wszystkie operacje wymienione poniżej modyfikują bezpośrednio obiekt listy, nie zmuszając nas do tworzenia nowej kopii, tak jak było to w przypadku łańcuchów znaków. Ponieważ w Pythonie mamy do czynienia z referencjami do obiektów, rozróżnienie między zmianą obiektu w miejscu a utworzeniem nowego obiektu ma znaczenie. Jak wspomniano w rozdziale 6., jeśli zmienimy obiekt w miejscu, może to wpływać na większą liczbę referencji do niego.

Przypisywanie do indeksu i wycinków

Kiedy używa się list, można zmieniać ich zawartość, przypisując coś do określonego elementu (pozycji o wartości przesunięcia) lub całego fragmentu (wycinka).

```
>>> L = ['mielonka', 'Mielonka', 'MIELONKA!']
>>> L[1] = 'jajka'                                # Przypisanie do indeksu
>>> L
['mielonka', 'jajka', 'MIELONKA!']
>>> L[0:2] = ['najsmaczniejsza', 'jest']        # Przypisanie do wycinka: usunięcie i wstawienie
>>> L
['najsmaczniejsza', 'jest', 'MIELONKA!']      # Zastąpienie elementów 0 i 1
```

Przypisanie do indeksu i wycinka to zmiana w miejscu. Operacje te modyfikują listę w sposób bezpośredni, a nie generują nowy obiekt listy dla wyniku. Przypisanie do indeksu w Pythonie działa mniej więcej tak, jak w C i innych językach programowania — Python zastępuje referencję do obiektu dla określonej pozycji przesunięcia nową referencją.

Przypisanie do wycinka, czyli ostatnia operacja z powyższego przykładu, zastępuje całą część listy za jednym razem. Ponieważ może być nieco skomplikowana, najlepiej jest ją sobie wyobrazić jako połączenie dwóch kroków:

1. *Usunięcie*. Wycinek podany po lewej stronie znaku = jest usuwany.
2. *Wstawienie*. Nowe elementy znajdujące się w obiekcie po prawej stronie znaku = wstawiane są do listy po lewej stronie, w miejscu, z którego wcześniej usunęliśmy stary wycinek.²

Wrzeczywistości nie wygląda to dokładnie tak, ale taki opis pozwala wyjaśnić, dlaczego liczba wstawianych elementów nie musi odpowiadać liczbie elementów usuwanych. Kiedy mamy na przykład listę L o wartości [1, 2, 3], przypisanie L[1:2] = [4, 5] sprawia, że lista [L] ma wartość [1, 4, 5, 3]. Python najpierw usuwa element 2 (wycinek jednoelementowy), a później wstawia 4 i 5 w miejsce usuniętego 2. Wyjaśnia to również, dlaczego L[1:2] = [] jest tak naprawdę operacją usunięcia — Python usuwa wycinek (element znajdujący się na pozycji o wartości przesunięcia 1), a następnie nic w to miejsce nie wstawia.

W rezultacie przypisanie do wycinka zastępuje cały fragment, czy inaczej „kolumnę”, za jednym razem. Ponieważ długość przypisywanej sekwencji nie musi odpowiadać długości wycinka, do którego ją przypisujemy, operację tę można wykorzystać do zastąpienia (poprzez nadpisanie), rozszerzenia (poprzez wstawienie) i skurczania (poprzez usunięcie) listy. Ta technika ma

² Opis ten wymaga rozszerzenia, kiedy wartość i przypisywany wycinek nakładają się na siebie. L[2:5] = L[3:6] zadziała na przykład dobrze, ponieważ wartość do wstawienia jest pobierana, zanim po lewej stronie wykonane zostanie usunięcie.

duże możliwości, jednak, szczerze mówiąc, nieczęsto spotyka się ją w praktyce. Zazwyczaj istnieją łatwiejsze sposoby zastępowania, wstawiania i usuwania (na przykład konkatenacja i metody list `insert`, `pop` oraz `remove`), które wolą stosować programiści Pythona.

Wywołania metod list

Tak jak łańcuchy znaków, obiekty list obsługują w Pythonie wywołania metod specyficznych dla tego typu obiektu.

```
>>> L.append('puszkowana')                                # Dodanie elementu na końcu listy
>>> L
['najsmaczniejsza', 'jest', 'MIELONKA!', 'puszkowana']
>>> L.sort()                                              # Sortowanie listy ('M' < 'j')
>>> L
['MIELONKA!', 'jest', 'najsmaczniejsza', 'puszkowana']
```

Metody zostały przedstawione w rozdziale 7. Mówiąc w skrócie, są one funkcjami (tak naprawdę — atrybutami odnoszącymi się do funkcji), które powiązane są z określonymi obiektami. Metody udostępniają narzędzia specyficzne dla danego typu. Zaprezentowane tutaj metody list są na przykład dostępne wyłącznie dla list.

Chyba najczęściej wykorzystywana metodą list jest `append`, wstawiająca pojedynczy element (referencję do obiektu) na koniec listy. W przeciwieństwie do konkatenacji `append` oczekuje przekazania pojedynczego obiektu, a nie listy. Rezultat działania `L.append(X)` jest podobny do `L+[X]`, jednak podczas gdy pierwszy ze sposobów modyfikuje listę `L` w miejscu, drugi tworzy nowy obiekt listy.³

Inna popularna metoda, `sort`, porządkuje elementy listy w miejscu. Domyślnie wykorzystuje standardowe testy porównania Pythona (tutaj porównywanie łańcuchów znaków) i sortuje listę w porządku rosnącym.

Działanie funkcji `sort` można modyfikować, wykorzystując *argumenty ze słowami kluczowymi* (z użyciem specjalnej składni `nazwa=wartość`, pozwalającej na przekazywanie wybranych argumentów funkcji, która jest często stosowana w Pythonie do przekazywania opcji konfiguracyjnych). Argument `key` pozwala na przekazanie jednoargumentowej funkcji zwracającej wartość, która ma być użyta w sortowaniu, natomiast argument `reverse` służy do odwracania kolejności sortowania (malejąco zamiast rosnąco).

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                              # Sortowanie bez uwzględnienia wielkości liter
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                                  # Normalizacja do małych liter
>>> L
['abc', 'ABD', 'aBe']
>>>
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)                   # Zmiana kolejności sortowania
>>> L
['aBe', 'ABD', 'abc']
```

³ W przeciwieństwie do konkatenacji (z operatorem `+`) metoda `append` nie musi generować nowych obiektów, więc jest zazwyczaj szybsza. Metodę tę można również sprytnie symulować za pomocą odpowiedniego przypisywania do wycinków — `L[:len(L):]` jest jak `L.append(X)`, natomiast `L[:0] = [X]` przypomina dodawanie elementu na początku listy. Oba usuwają pusty wycinek i wstawiają `X`, modyfikując listę `L` w miejscu tak szybko jak metoda `append`.

Argument `key` funkcji `sort` może być również użyty do wydobywania klucza słownika, w przypadku gdy chcemy sortować sekwencję słowników. Więcej informacji na temat słowników znajdziemy w dalszej części tego rozdziału, a na temat argumentów ze słowami kluczowymi funkcji w części IV.



Porównywanie i sortowanie w 3.0: w Pythonie 2.6 i wcześniejszych bez problemu działa porównywanie obiektów różnych typów (np. ciągu znaków i listy) — język ma zdefiniowany algorytm określania porządku danych różnych typów, który jest deterministyczny, choć być może mało elegancki. Kolejność jest zdefiniowana w oparciu o kolejność alfabetyczną nazw typów, czyli liczby są mniejsze od ciągów znaków, ponieważ `int` jest w alfabetie wcześniej niż `str`. Porównanie nigdy nie wykonuje konwersji typów, z wyjątkiem porównywania obiektów typu liczbowego.

W Pythonie 3.0 to uległo zmianie: próba porównania danych różnych typów wywołuje wyjątek. Sortowanie wykorzystuje operację porównywania, zatem wyrażenie `[1, 2, 'spam'].sort()` zadziała w Pythonie 2.X, ale w Pythonie 3.0 i nowszych wywoła wyjątek.

W Pythonie 3.0 zrezygnowano z możliwości przekazywania do funkcji `sort` dowolnej funkcji porównującej elementy. Sugerowanym obejściem tego braku jest użycie argumentu ze słowem kluczowym `key=func` służącego do przekształcania wartości przed sortowaniem, a w celu odwrócenia kolejności elementów należy używać parametru `reverse=True`. Takie użycie funkcji metody `sort` było również najpowszechniej stosowane w przeszłości.

Jedno ostrzeżenie: należy uważać na to, że metody `append` oraz `sort` modyfikują powiązany obiekt listy w miejscu, jednak nie zwracają listy jako wyniku (z technicznego punktu widzenia zwracają wartość `None`). Jeśli napiszemy coś podobnego do `L = L.append(X)`, nie otrzymamy zmodyfikowanej wartości `L` (a tak naprawdę całkowicie stracimy referencję do listy). Kiedy używa się atrybutów takich jak `append` czy `sort`, efektem ubocznym jest modyfikacja obiektów, dlatego nie ma potrzeby ponownego ich przypisywania.

Częściowo z powodu tego typu ograniczeń operacja sortowania jest w nowszych wersjach Pythona dostępna również w postaci funkcji wbudowanej `sorted`, która może być użyta do sortowania dowolnej sekwencji (nie tylko list) i zwraca nową listę (zamiast modyfikowania obiektu źródłowego).

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)                      # Wbudowana funkcja sortująca
['aBe', 'ABD', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True)               # Wstępne przekształcenie elementów: zmienione
                                                               # elementy wyniku!
['abe', 'abd', 'abc']
```

Warto zwrócić uwagę na ostatni przykład: dane wejściowe przekształcamy przed sortowaniem, wykorzystując wyrażenie listy składanej, co powoduje, że wynik nie zawiera oryginalnych wartości, jak to ma miejsce w przypadku użycia argumentu `key`. W przypadku tego ostatniego dane są przekształcane tymczasowo na potrzeby porównań, ale w wyniku zwracana jest ich oryginalna postać. W dalszej części rozdziału przedstawię kilka zastosowań, w których funkcja wbudowana `sorted` sprawdzi się lepiej od metody `sort`.

Podobnie do łańcuchów znaków, listy mają inne metody wykonujące różne wyspecjalizowane operacje. Metoda `reverse` odwraca na przykład listę w miejscu, `extend` wstawia kilka elemen-

tów na końcu listy, a pop usuwa element znajdujący się na końcu. Mamy również możliwość wykorzystania wbudowanej funkcji reversed, która działa tak samo jak sorted, ale zwraca iterator, dlatego w celu wyświetlenia jej wyników należy przekształcić je na listę (więcej na temat iteratorów w dalszej części książki).

```
>>> L = [1, 2]
>>> L.extend([3,4,5])                                # Dodanie kilku elementów
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                                         # Usunięcie i zwrócenie ostatniego elementu
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                                     # Odwrócenie w miejscu
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))                            # Wbudowana funkcja odwracająca sekwencję
[1, 2, 3, 4]
```

W pewnych rodzajach programów wykorzystana powyżej metoda pop jest często używana w połączeniu z append do szybkiego zaimplementowania struktury *stosu* typu LIFO (ang. *last-in-first-out*). Koniec listy służy za górę stosu.

```
>>> L = []
>>> L.append(1)                                    # Wstawienie na stos
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                                       # Usunięcie ze stosu
2
>>> L
[1]
```

Metoda pop akceptuje opcjonalny argument offset, służący określeniu indeksu elementu, który ma być zwrócony (domyślnie jest to ostatni element). Istnieją też metody usuwające elementy po wartości (remove), wstawiające element we wskazanym miejscu (insert), wyszukujące pozycję elementu o zadanej wartości (index) i wiele innych.

```
>>> L = ['mielonka', 'jajka', 'szynka']
>>> L.index('jajka')                             # Pozycja elementu o zadanej wartości
1
>>> L.insert(1, 'tost')                          # Wstawienie na pozycji
>>> L
['mielonka', 'tost', 'jajka', 'szynka']
>>> L.remove('jajka')                           # Usunięcie elementu o zadanej wartości
>>> L
['mielonka', 'tost', 'szynka']
>>> L.pop(1)                                     # Usunięcie elementu na zadanej pozycji
'tost'
>>> L
['mielonka', 'szynka']
```

Więcej informacji na temat metod obiektów list można znaleźć w dokumentacji Pythona, warto też poeksperymentować z nimi w konsoli interaktywnej.

Inne popularne operacje na listach

Ponieważ listy są typami zmiennymi, można użyć instrukcji del do usunięcia elementu lub części listy w miejscu.

```

>>> L
['MIELONKA!', 'jest', 'najsmaczniejsza', 'puszkowana']
>>> del L[0]                                     # Usunięcie jednego elementu
>>> L
['jest', 'najsmaczniejsza', 'puszkowana']
>>> del L[1:]                                    # Usunięcie całej części
>>> L                                         # To samo co L[1:] = []
['jest']

```

Ponieważ przypisanie wycinka składa się z usunięcia i wstawienia, można również usunąć fragment listy, przypisując do wycinka pustą listę (`L[i:j] = []`). Python usuwa wycinek podany po lewej stronie, a następnie nie wstawia nic w jego miejsce. Z drugiej strony, przypisanie pustej listy do indeksu po prostu przechowuje referencję do pustej listy w określonym miejscu, zamiast coś usuwać.

```

>>> L = ['Mam', 'już', 'coś']
>>> L[1:] = []
>>> L
['Mam']
>>> L[0] = []
>>> L
[]

```

Choć wszystkie omawiane wyżej operacje są typowe, istnieją dodatkowe metody list, których nie zilustrowaliśmy (w tym metody służące do wyszukiwania i wstawiania). Wyczerpującą i zawsze aktualną listę narzędzi dla określonego typu można znaleźć w dokumentacji Pythona za pomocą funkcji `dir` oraz `help` (z którymi spotkaliśmy się już w rozdziale 4.), a także w książce *Python. Leksykon kieszonkowy* (Helion) i innych tekstach przedstawionych w przedmowie do niniejszej książki.

Chciałbym również przypomnieć raz jeszcze, że wszystkie omówione wyżej operacje modyfikujące obiekt w miejscu działają jedynie na obiektach zmiennych. Nie będą działały na łańcuchach znaków (ani omówionych w rozdziale 9. krotkach) — bez względu na to, jak bardzo będziemy się o to starać. Zmienna jest wrodzoną właściwością każdego typu obiektu.

Słowniki

Obok list, *słowniki* (ang. *dictionary*) są chyba najbardziej elastycznym wbudowanym typem danych w Pythonie. Jeśli wyobrażmy sobie listy jako uporządkowane kolekcje obiektów, słowniki będą kolekcjami nieuporządkowanymi. Podstawowa różnica między tymi dwoma typami danych polega na tym, że w słownikach elementy przechowywane są zgodnie z kluczem, a nie pozycją przesunięcia. W podobny sposób odbywa się dostęp do nich — za pomocą *klucza*.

Ponieważ słowniki są typami wbudowanymi, mogą zastąpić wiele mechanizmów wyszukiwania i struktur danych, które w językach niższego poziomu musielibyśmy implementować ręcznie — indeksowanie słownika jest bowiem wyjątkowo szybką operacją wyszukiwania. Słowniki czasami wykonują również pracę rekordów i tablic symbolicznych z innych języków programowania. Mogą też na przykład reprezentować rzadkie (w większości puste) struktury danych. Poniżej znajduje się podsumowanie ich właściwości.

Dostęp do słowników odbywa się po kluczu, a nie wartości przesunięcia

Słowniki czasami nazywane są *tablicami asocjacyjnymi* lub *tablicami mieszającymi* (ang. *associative array* lub *hash*). Wiążą one zbiór wartości z kluczami, tak by element słownika można było pobrać za pomocą klucza, pod którym jest on przechowywany. Do pobrania

komponentów słownika wykorzystuje się tę samą operację indeksowania co w przypadku list, jednak indeks ma tutaj postać klucza, a nie względnej wartości przesunięcia.

Słowniki są nieuporządkowanymi kolekcjami dowolnych obiektów

W przeciwnieństwie do listy elementy przechowywane w słowniku nie są utrzymywane w żadnej szczególnej kolejności. Tak naprawdę kolejność elementów słownika jest w Pythonie losowa, tak by szybciej można je było przeszukiwać. Klucze udostępniają symboliczną (nie fizyczną) lokalizację elementów w słowniku.

Słowniki mają zmienną długość, są heterogeniczne i mogą być dowolnie zagnieżdżane

Tak jak listy, słowniki mogą rozszerzać się i kurczyć w miejscu (bez tworzenia nowych kopii) i mogą zawierać obiekty dowolnego typu. Obsługują również zagnieżdżanie na dowolną głębokość (mogą zawierać listy czy inne słowniki).

Słowniki należą do kategorii zmiennych odwzorowań

Słowniki można zmieniać w miejscu dzięki przypisywaniu do indeksów (kluczy), co wynika z tego, iż są one zmiennym typem danych. Nie obsługują jednak operacji na sekwencjach działających na łańcuchach znaków oraz listach. Ponieważ słowniki są kolekcjami nieuporządkowanymi, wszystkie operacje opierające się na stałej kolejności elementów (na przykład konkatenacja czy wycinki) nie mają w ich przypadku większego sensu. Zamiast tego słowniki są jedynym wbudowanym przedstawicielem kategorii odwzorowań (obiektów odwzorowujących klucze na wartości).

Słowniki są tabelami referencji do obiektów (tablicami asocjacyjnymi)

Jeśli listy są tablicami referencji do obiektów obsługującymi dostęp za pomocą pozycji elementów, słowniki to nieuporządkowane tabele referencji do obiektów obsługujące dostęp za pomocą klucza. Wewnętrznie słowniki zaimplementowane są jako tablice asocjacyjne (struktury danych obsługujące bardzo szybkie pobieranie), które mogą rozszerzać się i kurczyć na żądanie. Co więcej, Python wykorzystuje zoptymalizowane algorytmy miejszące służące do odnajdywania kluczy, dzięki czemu to pobieranie jest naprawdę szybkie. Podobnie do list, słowniki przechowują referencje do obiektów (a nie ich kopie).

W tabeli 8.2 przedstawione zostały najczęściej wykorzystywane i najbardziej reprezentatywne operacje na słownikach (jak zawsze pełną ich listę można znaleźć w dokumentacji Pythona, a także wywołując funkcję `dir(dict)` lub `help(dict)` — `dict` to nazwa typu słownika). Kiedy słownik zakoduje się w postaci wyrażenia z literałem, składa się on z serii par `klucz:wartość` rozdzielonych przecinkami i umieszczonego w nawiasach klamrowych. Pusty słownik to pusta para nawiasów klamrowych.⁴ Słowniki można osadzać, wstawiając je jako wartość wewnętrz innego słownika, listy lub krotki.

Słowniki w akcji

Jak widać w tabeli 8.2, dostęp do słowników odbywa się po kluczu. Do zagnieżdżonych wpisów ze słownika można się odnieść za pomocą serii indeksów (kluczy w nawiasach kwadratowych). Kiedy Python tworzy słownik, przechowuje elementy w dowolnym porządku od

⁴ Tak jak w przypadku list, słowniki rzadko konstruuje się za pomocą literalów. Listy i słowniki tworzone są jednak na różne sposoby. Jak zobaczymy w następnym podrozdziale, słowniki zazwyczaj rozszerza się, przypisując wartości do nowych kluczy w czasie wykonywania. Takie podejście nie działa w listach — listy rozszerza się za pomocą metody `append`.

Tabela 8.2. Popularne literały i operacje słowników

Operacja	Interpretacja
D = {}	Pusty słownik
D = {'mielonka': 2, 'jajka': 3}	Słownik dwuelementowy
D = {'jedzenie': {'szynka': 1, 'jajka': 2}}	Zagnieżdżanie
D = dict(name='Bob', age=40)	Alternatywne techniki konstruowania: słowa kluczowe, zzipowane pary, listy kluczy
D = dict(zip(keyslist, valslist))	
D = dict.fromkeys(['a', 'b'])	
D['jajka']	Indeksowanie po kluczu
D['jedzenie']['szynka']	
'jajka' in D	Metody: sprawdzanie przynależności
D.keys()	Metody: lista kluczy
D.values()	lista wartości,
D.items()	klucze+wartości
D.copy()	kopie,
D.get(key, default)	wartości domyślne,
D.update(D2)	łączenie,
D.pop(key)	usuwanie itp
len(D)	Długość (liczba przechowywanych wpisów)
D[key] = 42	Dodawanie lub modyfikacja kluczy, usuwanie kluczy
del D[key]	Usuwanie elementu po kluczu
list(D.keys())	Widoki słowników (Python 3.0)
D1.keys() & D2.keys()	
D = {x: x*2 for x in range(10)}	Słowniki składane (Python 3.0)

lewej do prawej strony. By pobrać wartość ze słownika, należy podać powiązany z nią klucz. Wróćmy zatem do interpretera Pythona w celu wypróbowania kilku z operacji na słownikach zaprezentowanych w tabeli 8.2.

Podstawowe operacje na słownikach

Zazwyczaj tworzy się słowniki i przechowuje elementy (oraz później uzyskuje do nich dostęp) za pomocą kluczy.

```
% python
>>> D = {'mielonka': 2, 'szynka': 1, 'jajka': 3} # Utworzenie słownika
>>> D['mielonka'] # Pobranie wartości po kluczu
2
>>>D # Kolejność zostaje pomieszana
{'szynka': 1, 'jajka': 3, 'mielonka': 2}
```

W powyższym kodzie słownik przypisywany jest do zmiennej D. Wartością klucza 'mielonka' jest liczba całkowita 2; podobnie wyglądają pozostałe wpisy. Do zindeksowania słownika po kluczu wykorzystaliśmy tę samą składnię z nawiasami kwadratowymi, jaką była używana przy indeksowaniu list za pomocą pozycji przesunięcia. Tutaj oznacza ona jednak dostęp po kluczu, a nie pozycji elementu.

Warto również zwrócić uwagę na końówkę tego przykładu. Kolejność elementów słownika od lewej do prawej strony prawie zawsze jest inna od ich początkowej kolejności. Jest to celowe — w celu zaimplementowania szybkiego wyszukiwania kluczy (inaczej mieszania, ang. *hashing*) muszą one zostać w pamięci ułożone w sposób losowy. Z tego powodu operacje zakładające stały porządek elementów od lewej do prawej strony (na przykład wycinki czy konkatenacja) nie mają zastosowania do słowników. Wartości można pobrać jedynie za pomocą klucza, a nie pozycji.

Wbudowana funkcja `len` działa również na słownikach. Zwraca ona liczbę elementów przechowywanych w słowniku lub — co na jedno wychodzi — długość listy kluczy. Metoda słownika `has_key` oraz operator przynależności `in` pozwalają sprawdzać istnienie klucza, natomiast metoda `keys` zwraca wszystkie klucze ze słownika zebrane w listę. Może ona być przydatna do sekwencyjnego przetwarzania słowników, jednak nie powinno się przykładać zbyt dużej wagi do kolejności listy kluczy. Ponieważ metoda ta zwraca jednak normalną listę, zawsze można ją posortować, jeśli kolejność ma dla nas jakieś znaczenie.

```
>>> len(D)                                # Liczba wpisów w słowniku
3
>>> 'szynka' in D                          # Alternatywne sprawdzanie istnienia klucza
True
>>> list(D.keys())                         # Utworzenie nowej listy kluczy
['szynka', 'jajka', 'mielonka']
```

Warto zwrócić uwagę na trzecie wyrażenie z tego listingu. Jak wspomniano wcześniej, test przynależności `in` wykorzystywany w łańcuchach znaków oraz listach będzie również działał na słownikach. Sprawdza on, czy klucz jest przechowywany w słowniku. Z technicznego punktu widzenia rozwiązywanie to działa, ponieważ słowniki definiują *iteratory*, które przechodzą listy ich kluczy. Inne typy udostępniają iteratory odzwierciedlające ich częste zastosowania. W przypadku plików istnieją na przykład iteratory wczytujące wiersz po wierszu. Iteratory zostaną omówione bardziej szczegółowo w rozdziałach 14. oraz 20.

Warto zwrócić uwagę na ostatni przykład tego listingu. W Pythonie 3.0 musimy przekształcić na listę wynik metody `keys`: w 3.0 metoda ta zwraca iterator, nie gotową listę. Wywołanie funkcji `list` powoduje wygenerowanie wszystkich elementów i przekształcenie ich na listę, dzięki czemu można je wyświetlić. W 2.6 metoda `keys` buduje i zwraca rzeczywistą listę, zatem wywołanie funkcji `list` nie jest konieczne do wyświetlenia wyników. Więcej informacji na ten temat przedstawię w dalszej części rozdziału.



Kolejność kluczy w słowniku jest nieokreślona i może ulegać zmianom w różnych wersjach Pythona, zatem nie należy się niepokoić, gdy prezentowany kod zwróci wyniki w innej kolejności niż w moich przykładach. W moim przypadku wyniki są generowane w Pythonie 3.0, a we wcześniejszych wersjach Pythona kolejność wyników była inna. Nie należy opierać logiki pisanych programów (ani książek) na kolejności danych w słownikach!

Modyfikacja słowników w miejscu

Kontynuujmy naszą sesję interaktywną. Słowniki, podobnie jak listy, są zmienne, zatem można je modyfikować, rozszerzać i kurczyć w miejscu bez tworzenia nowych słowników. Wystarczy przypisać wartość do klucza lub utworzyć nowy wpis. Działa również instrukcja `del`, która

usuwa wpis powiązany z kluczem podanym w indeksie. Warto również zwrócić uwagę na zagnieżdżenie listy wewnętrz słownika (wartość klucza 'szynka'). Wszystkie typy kolekcji z Pythona mogą być wewnątrz siebie dowolnie zagnieżdżane.

```
>>> D
{'jajka': 3, 'szynka': 1, 'mielonka': 2}

>>> D['szynka'] = ['grillowanie', 'pieczenie', 'smażenie'] # Zmiana wpisu
>>> D
{'jajka': 3, 'szynka': ['grillowanie', 'pieczenie', 'smażenie'], 'mielonka': 2}

>>> del D['jajka']                                # Usunięcie wpisu
>>> D
{'szynka': ['grillowanie', 'pieczenie', 'smażenie'], 'mielonka': 2}

>>> D['lunch'] = 'Bekon'                          # Dodanie nowego wpisu
>>> D
{'lunch': 'Bekon', 'szynka': ['grillowanie', 'pieczenie', 'smażenie'], 'mielonka': 2}
```

Tak jak w przypadku listy, przypisanie elementu do istniejącego indeksu w słowniku również zmienia jego wartość. Jednak w przeciwnieństwie do list przypisanie *nowego* klucza słownika powoduje utworzenie nowego wpisu, tak jak w powyższym przykładzie w przypadku klucza 'lunch' (tego, który nie był wcześniej przypisany). Nie działa to dla list, ponieważ Python uznał wartość przesunięcia przekraczającą długość listy za błąd. By rozszerzyć listę, konieczne jest skorzystanie z innego narzędzia, na przykład metody append lub przypisania do wycinka.

Inne metody słowników

Metody słowników udostępniają różnorodne narzędzia. Na przykład metody values oraz items zwracają odpowiednio listę wartości oraz krotki par (klucz, wartość) dla słownika (w celu wyświetlenia wyników w Pythonie 3.0 ich wynik należy przekształcić na listę, podobnie jak w przypadku metody keys).

```
>>> D = {'mielonka': 2, 'szynka': 1, 'jajka': 3}
>>> list(D.values())
[1, 3, 2]
>>> (D.items())
[('jajka', 3), ('szynka', 1), ('mielonka', 2)]
```

Takie listy przydają się w pętlach przechodzących po kolej wpisy słownika. Pobranie nieistniejącego klucza jest normalnie błędem, jednak metoda get zwraca wartość domyślną (None lub przekazaną wartość domyślną) dla klucza, który nie istnieje. Poniżej widać łatwy sposób na podanie wartości domyślnej dla nieistniejącego klucza i tym samym uniknięcia błędu.

```
>>> D.get('mielonka')                           # Klucz istnieje
2
>>> print(D.get('tost'))                         # Brakujący klucz
None
>>> D.get('tost', 88)
88
```

Metoda update udostępnia słownikom coś podobnego do konkatenacji, ale w tym przypadku nie jest zachowana kolejność elementów (przypominam: w przypadku słowników kolejność nie ma znaczenia). Łączy klucze i wartości słownika z innym słownikiem, ślepo nadpisując wartości o tym samym kluczu.

```

>>> D
{'jajka': 3, 'szynka': 1, 'mielonka': 2}
>>> D = {'tost':4, 'ciastko':5}
>>> D.update(D2)
>>> D
{'tost': 4, 'ciastko': 5, 'jajka': 3, 'szynka': 1, 'mielonka': 2, }

```

Wreszcie metoda słownika pop usuwa klucz ze słownika i zwraca wartość, jaką miał ten klucz. Przypomina ona metodę listy pop, jednak jako argument przyjmuje klucz, a nie opcjonalną pozycję.

```

# Usunięcie wpisu ze słownika po jego kluczu

>>> D
{'tost': 4, 'ciastko': 5}, 'jajka': 3, 'szynka': 1, 'mielonka': 2,
>>> D.pop('ciastko')
5
>>> D.pop('tost')                                # Usunięcie i zwrócenie wartości klucza
4
>>> D
{'jajka': 3, 'szynka': 1, 'mielonka': 2}

```

Usunięcie elementu z listy po jego pozycji

```

>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                                     # Usunięcie i zwrócenie ostatniego elementu
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                                    # Usunięcie elementu z określonej pozycji
'bb'
>>> L
['aa', 'cc']

```

Słowniki udostępniają również metodę copy. Omówimy ją w rozdziale 9., ponieważ jest ona sposobem pozwalającym na uniknięcie potencjalnych efektów ubocznych wynikających ze współdzielonych referencji do tego samego słownika. Tak naprawdę słowniki mają o wiele więcej metod, niż zaprezentowano w tabeli 8.2. Pełną ich listę można znaleźć w dokumentacji biblioteki Pythona lub innych źródłach.

Przykład z tabelą języków programowania

Przyjrzyjmy się zatem bardziej realistycznemu przykładowi słownika. Poniższy kod tworzy tabelę odwzorowującą nazwy języków programowania (klucze) na ich twórców (wartości). Dane twórców można uzyskać, indeksując słownik po nazwach języków.

```

>>> table = {'Python': 'Guido van Rossum',
...           'Perl':    'Larry Wall',
...           'Tcl':     'John Ousterhout' }
...
>>> language = 'Python'
>>> creator = table[language]
>>> creator
'Guido van Rossum'

>>> for lang in table:                         # Równoznaczne: for lang in table.keys()
...     print lang, '\t', table[lang]
...

```

Tcl	John Ousterhout
Python	Guido van Rossum
Perl	Larry Wall

W ostatnim poleceniu wykorzystano pętlę `for`, której jeszcze nie omawialiśmy. Osobom niezaznajomionym z pętlami `for` wystarczy informacja, że polecenie to wykonuje iterację po każdym kluczu tabeli i wyświetla rozdzieloną tabulatorami listę kluczów i wartości. Więcej informacji na temat pętli `for` można znaleźć w rozdziale 13.

Ponieważ słowniki nie są sekwencjami, nie można za pomocą `for` iterować po nich w sposób bezpośredni, tak jak robi się to w przypadku łańcuchów znaków oraz list. Jeśli jednak chcemy przejść elementy słownika, jest to łatwe. Wystarczy wywołać metodę słownika `keys` zwracającą listę wszystkich przechowywanych kluczów, po której można iterować za pomocą pętli `for`. Jeśli wystąpi taka potrzeba, możemy wewnątrz pętli indeksować słownik kluczem w celu otrzymywania wartości — tak jak ma to miejsce w kodzie powyżej.

Tak naprawdę Python w większości pętli `for` pozwala również przechodzić listę kluczów słownika bez wywoływania metody `keys`. Dla słownika `D` wyrażenie `for key in D:` działa tak samo jak pełne wyrażenie `for key in D.keys():`. To tak naprawdę kolejny przypadek wspomnianych wcześniej iteratorów, które pozwalają na działanie operatora przynależności `in` również na słownikach (więcej informacji na temat iteratorów w dalszej części książki).

Uwagi na temat korzystania ze słowników

Słowniki są dość proste w użyciu, kiedy się je już opanuje, jednak warto pamiętać o kilku wskazówkach i uwagach związanych z ich stosowaniem:

- **Na słownikach nie działają operacje na sekwencjach.** Słowniki są odwzorowaniami, a nie sekwencjami. Ponieważ w przypadku ich elementów nie ma mowy o kolejności i uporządkowaniu, nie mają tu zastosowania pewne operacje, takie jak konkatenacja (uporządkowane łączenie) czy wycinki (ekstrakcja przylegającego fragmentu). Tak naprawdę jeśli spróbujemy zrobić coś takiego, w trakcie wykonywania kodu Python zwróci błąd.
- **Przypisanie do nowych indeksów dodaje wpisy.** Klucze można tworzyć albo przy zapisywaniu literala słownika (w tym przypadku są one osadzone w samym literale), albo kiedy przypisuje się wartości do nowych kluczów istniejącego obiektu słownika. Rezultat będzie taki sam.
- **Klucze nie muszą być łańcuchami znaków.** W naszych przykładach kluczami były właśnie łańcuchy, ale ich rolę mogą równie dobrze spełniać dowolne obiekty *niezmienné* (czyli nie listy). Można na przykład jako klucze wykorzystywać liczby całkowite, co sprawia, że słownik zaczyna przypominać listę (przynajmniej w czasie indeksowania). Czasami w tej roli wykorzystywane są również krótki, co pozwala na tworzenie złożonych wartości kluczów. Jako klucze można wykorzystać również obiekty instancji klas (omówione w szóstej części książki), o ile mają one odpowiednie metody protokołów. W przybliżeniu muszą one przekazać Pythonowi, że ich wartości nie zmieniają się, ponieważ w przeciwnym razie nie mogłyby być wykorzystywane jako klucze o stałej wartości.

Wykorzystywanie słowników do symulowania elastycznych list

Ostatni punkt poprzedniej listy jest ważny na tyle, by poprzeć go kilkoma przykładami. Kiedy używa się list, nie można przypisać elementu do pozycji przesunięcia znajdującej się poza długością tej listy.

```
>>> L = []
>>> L[99] = 'mielonka'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Choć można wykorzystać powtórzenie do uprzedniego utworzenia listy wystarczająco dużej na nasze potrzeby (na przykład `[0]*100`), coś podobnego można również uzyskać za pomocą słowników i nie wymaga to takiej alokacji miejsca. Używając kluczy w postaci liczb całkowitych, słowniki mogą emulować listy rosnące dzięki przypisywaniu do pozycji przesunięcia.

```
>>> D = {}
>>> D[99] = 'mielonka'
>>> D[99]
'mielonka'
>>> D
{99: 'mielonka'}
```

D wygląda na listę stwierdzeniową, jednak tak naprawdę jest tylko słownikiem z jednym wpisem. Wartością klucza 99 jest łańcuch znaków 'mielonka'. Dostęp do tej struktury można uzyskać za pomocą wartości przesunięcia (podobnie jak w liście), jednak nie musimy alokować miejsca na wszystkie pozycje, jakie kiedykolwiek będziemy mieli potrzebować do przypisania w przyszłości. W takiej postaci słowniki są bardziej elastycznymi odpowiednikami list.

Wykorzystywanie słowników z rzadkimi strukturami danych

W podobny sposób klucze słowników można wykorzystać do implementowania *rzadkich struktur danych* (ang. *sparse data structure*) — na przykład tablic wielowymiarowych, w których jedynie kilka pozycji zawiera wartości.

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4                                # Znak ; rozdziela instrukcje
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

W powyższym kodzie słownik wykorzystaliśmy do reprezentacji tablicy trójwymiarowej, która jest prawie pusta — zawiera jedynie dwie pozycje: (2, 3, 4) oraz (7, 8, 9). Klucze są *krotkami* przechowującymi współrzędne niepustych wpisów. Zamiast alokować dużą i w większości pustą macierz trójwymiarową, możemy skorzystać z prostego, dwuelementowego słownika. W takim układzie próba dostępu do pustego wpisu kończy się błędem nieistniejącego klucza, ponieważ wpisy takie nie są fizycznie przechowywane.

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: (2, 3, 6)
```

Unikanie błędów z brakującymi kluczami

Błędy wynikające z prób pobrania nieistniejących kluczy zdarzają się często w rzadkich macierzach, jednak na ogół wolelibyśmy, by nie kończyły one całego programu. Istnieją przynajmniej trzy sposoby wypełnienia wartości domyślnych w miejsce otrzymania błędów tego typu. Można

uprzednio sprawdzać istnienie kluczy za pomocą instrukcji `if`, wykorzystać instrukcję `try` do przechwycenia i obsługi błędu albo po prostu wykorzystać zaprezentowaną wcześniej metodę słownika `get` do podania wartości domyślnej dla nieistniejących kluczy.

```
>>> if ((2,3,6) in Matrix:  
...     print Matrix[(2,3,6)]  
... else:  
...     print 0  
...  
0  
  
>>> try:  
...     print Matrix[(2,3,6)]  
... except KeyError:  
...     print 0  
...  
0  
>>> Matrix.get((2,3,4), 0)  
88  
>>> Matrix.get((2,3,6), 0)  
0
```

Sprawdzanie kluczy przed pobraniem
Więcej w rozdziale 12. przy okazji zagadnienia if/else

Próba indeksowania
Przechwycenie i obsługa błędu
Więcej w rozdziale 33. przy okazji zagadnienia try/except

Istnieje — pobranie i zwrócenie
Nie istnieje — użycie argumentu domyślnego

Oczywiście metoda `get` jest najbardziej zwięzła. Instrukcje `if` oraz `try` zostaną omówione bardziej szczegółowo w dalszej części książki.

Wykorzystywanie słowników w postaci „rekordów”

Jak widać, słowniki mogą w Pythonie odgrywać różne role. Mogą zastępować struktury wyszukujące dane (ponieważ indeksowanie po kluczu jest operacją wyszukiwania), a także reprezentować różne typy ustrukturyzowanych informacji. Słowniki są na przykład jednym ze sposobów opisu właściwości obiektu z dziedziny zastosowania programu. Pełnią zatem tę samą rolę co „rekordy” czy „struktury” (ang. *struct*) w innych językach programowania.

Poniższy przykład wypełnia słownik, przypisując do niego z czasem nowe klucze.

```
>>> rec = {}  
>>> rec['name'] = 'mel'  
>>> rec['age'] = 45  
>>> rec['job'] = 'instruktor'  
>>>  
>>> print rec['name']  
mel
```

Wbudowane typy danych Pythona pozwalają z łatwością reprezentować ustrukturyzowane informacje, w szczególności po zagnieźdzeniu. Poniższy przykład wykorzystuje słownik do zamieszczenia właściwości obiektu, jednak teraz wszystko kodowane jest za jednym razem (a nie przypisywane pojedynczo do kluczy). Zagnieżdżone lista oraz słownik reprezentują wartości ustrukturyzowanych właściwości.

```
>>> mel = {'name': 'Mark',  
...         'jobs': ['instruktor', 'pisarz'],  
...         'web': 'www.rimi.net/~lutz',  
...         'home': {'state': 'CO', 'zip': 80513}}
```

By pobrać komponenty zagnieżdżonych obiektów, należy połączyć ze sobą operacje indeksowania.

```
>>> mel['name']  
'Mark'  
>>> mel['jobs']
```

```
['instruktor', 'pisarz']
>>> mel['jobs'][1]
'pisarz'
>>> mel['home']['zip']
80513
```

W części VI dowiemy się, że klasy (łączące dane i logikę) lepiej nadają się do tego typu zadań, ale słowniki są łatwiejsze w użyciu w przypadku prostych zastosowań.

Znaczenie interfejsów do słowników

Poza tym, że słowniki są wygodnym sposobem przechowywania informacji po kluczu w programach, interfejsy wyglądające i działające jak słowniki są również obecne w niektórych rozszerzeniach do Pythona. Interfejs Pythona służący do dostępu do plików DBM dostępnych po kluczu przypomina na przykład słownik, który trzeba otworzyć. Łańcuchy znaków przechowuje się i pobiera za pomocą indeksów w postaci kluczy.

```
import anydbm
file = anydbm.open("nazwa_pliku")           # Łącze do pliku
file['key'] = 'data'                         # Przechowanie danych po kluczu
data = file['key']                           # Pobranie danych po kluczu
```

W rozdziale 27. zobaczymy również, że w ten sposób można pobierać całe obiekty Pythona, jeśli zastąpimy `anydbm` w kodzie wyżej odwołaniem do `shelve` (`shelve` to baza danych trwałych obiektów Pythona dostępna po kluczu). W przypadku pracy z Internetem obsługa skryptów CGI w Pythonie również zawiera interfejs podobny do słownika. Wywołanie `cgi.FieldStorage` zwraca obiekt podobny do słownika z jednym wpisem na pole wejściowe na stronie klienta.

```
import cgi
form = cgi.FieldStorage()                   # Analiza danych z formularza
if form.has_key('name'):
    showReply('Witaj, ' + form['name'].value)
```

Wszystkie te formy (wraz ze słownikami) są przykładami odwzorowań. Kiedy opanujemy interfejsy słowników, zobaczymy, że mają one zastosowanie do wielu narzędzi wbudowanych Pythona.

Inne sposoby tworzenia słowników

Warto pamiętać, że dzięki popularności słowników z czasem powstało wiele sposobów ich tworzenia. W Pythonie 2.3 i nowszych mamy do dyspozycji konstruktor `dict`, któremu przekazuje się poszczególne elementy tworzonego słownika w parametrach słownikowych lub w postaci listy par klucz/wartość. W wyniku tego typu wywołanie konstruktora daje ten sam efekt co literał słownikowy i przypisania do kluczy.

```
{'name': 'mel', 'age': 45}                  # Tradycyjne wyrażenie literalu słownika
D = {}                                         # Dynamiczne przypisanie do kluczy
D['name'] = 'mel'
D['age'] = 45

dict(name='mel', age=45)                       # Argument ze słowem kluczowym
dict([('name', 'mel'), ('age', 45)])          # Krotki klucz/wartość
```

Wszystkie powyższe formy tworzą taki sam słownik zawierający dwa klucze, ale każda z tych form jest przydatna w nieco innych okolicznościach:

- pierwsza z nich przydaje się w sytuacjach, gdy jesteśmy w stanie statycznie zadeklarować zawartość słownika;
- druga jest użyteczna w sytuacji, gdy słownik chcemy tworzyć dynamicznie, po jednym elemencie;
- trzecia forma wymaga mniej pisania od pierwszej, ale wymaga, aby klucze słownika były ciągami znaków;
- ostatnia forma jest najbardziej użyteczna w przypadku, gdy posiadamy listę par klucz/wartość, wygenerowaną dynamicznie w innej części programu.

Argumenty ze słowami kluczowymi spotkaliśmy wcześniej w niniejszym rozdziale, przy okazji sortowania. Trzecia forma tworzenia słownika stała się szczególnie popularna w Pythonie, ponieważ wykorzystuje najprostszą składnię (dzięki czemu prowokuje mniejszą ilość błędów). Jak sugerowałem w tabeli 8.2, ostatnia forma przedstawiona na powyższym listingu jest również powszechnie stosowana w połączeniu z funkcją `zip`, pozwalając na połączenie osobnych list kluczów i wartości. Więcej przykładów użycia tej formy w następnym punkcie.

Jeśli wszystkie klucze mają początkowo mieć przypisaną tę samą wartość, słownik można zbudować z użyciem jeszcze jednej specjalnej formy: konstruktora `fromkeys` klasy `dict`, pozwalającej podać listę kluczów oraz pojedynczą wartość początkową dla wszystkich z nich (domyślnie wartością tą jest `None`).

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

Na początkowym etapie poznawania Pythona większość programistów wystarczają literały i przypisanie po kluczu, ale warto znać praktyczne zastosowania różnych form tworzenia słowników, które są przydatne w realistycznych, elastycznych i wydajnych programach w Pythonie.

Przykłady prezentowane w powyższym listingu działają w Pythonie 2.6 i 3.0, ale istnieje jeszcze jeden sposób tworzenia słowników, dostępny wyłącznie w 3.0: *wyrażenie słowników składanych*. Szczegóły zastosowania tej techniki są opisane w następnym punkcie.

Zmiany dotyczące słowników w 3.0

Tematyka słowników omawiana w niniejszym rozdziale dotychczas dotyczała jedynie zagadnień wspólnych dla wszystkich wydań Pythona, ale mechanizm słowników uległ zmianom w 3.0. Użytkownicy kodu napisanego z myślą o serii 2.X mogą spotkać się z sytuacjami, gdy narzędzia obsługi słowników będą działały inaczej, lub w ogóle ich nie znajdą w 3.0. Co więcej, programiści, którzy pracują wyłącznie w 3.0, mają dostęp do nowych mechanizmów niedostępnych w wydaniach z serii 2.X. Do nowości w mechanizmach obsługi słowników Pythona 3.0 zalicza się:

- obsługę nowego wyrażenia słowników składanych, będącego analogią do list i zbiorów składanych;
- w przypadku metod `D.keys`, `D.values` i `D.items` zamiast list zwracane są widoki, będące obiektami iterowanymi;
- z powyższego wynika konieczność stosowania nieco innych technik programowania w celu użycia posortowanych kluczów;

- brak bezpośredniej obsługi względnych porównań wielkości słowników, wielkość słowników porównujemy ręcznie;
- brak metody `D.has_key`, zamiast tego stosuje się test przynależności.

Przyjrzyjmy się po kolejnym nowościom dotyczącym słowników w 3.0.

Słowniki składane

Jak wspomniałem w poprzednim punkcie, w Pythonie 3.0 słowniki mogą być tworzone z użyciem nowej konstrukcji: wyrażenia słowników składanych (ang. *dictionary comprehension*). Podobnie jak wyrażenia zbiorów składanych omówione w rozdziale 5., słowniki składane są dostępne od wersji 3.0 (nie można ich użyć w 2.6). Podobnie jak dostępne od dawna listy składane, które mieliśmy okazję napotkać w rozdziale 4. i wcześniej w niniejszym rozdziale, słowniki składane budują wynik wewnętrznej pętli, w każdej iteracji generując parę klucz/wartość, którymi wypełniają wynikowy słownik. Zmienna użyta w pętli może być wykorzystana jako element, z którego konstruowane są klucze i wartości.

Na przykład standardowym sposobem tworzenia słowników w 2.6 i 3.0 jest połączenie za pomocą funkcji `zip` list kluczy i wartości i przekazanie wyniku tej operacji do konstruktora `dict`. W rozdziale 13. przeanalizujemy szczegółowo działanie funkcji `zip`, w tym miejscu wystarczy nam wiedzieć, że funkcja ta pozwala w pojedynczym wywołaniu połączyć klucze i wartości słownika. Jeśli nie można przewidzieć z góry zestawu kluczy i wartości słownika, można zbudować je jako listy, po czym podać działaniu funkcji `zip`:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))           # Połączenie kluczy i wartości
[('a', 1), ('b', 2), ('c', 3)]
>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))      # Stworzenie słownika z wyniku funkcji zip
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

W Pythonie 3.0 możemy uzyskać ten sam efekt, wykorzystując wyrażenie słownika składanego. Poniższy listing przedstawia sposób utworzenia nowego słownika na podstawie wyniku funkcji `zip` wywołanej na listach kluczy i wartości:

```
C:\misc> c:\python30\python                         # Użycie słownika składanego
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

W tym przypadku słownik składany wymagał napisania większej ilości kodu, ale wyrażenia te są znacznie bardziej ogólne, niż mógłby sugerować ten przykład: można za ich pomocą na przykład zbudować słownik w oparciu o jeden strumień danych, a klucze, jak i wartości, mogą być generowane w wyrażeniach:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}          # Lub: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}
>>> D = {c: c * 4 for c in 'JAJKO'}                # Iteracja po dowolnym iteratorze
>>> D
{'A': 'AAAA', 'K': 'KKKK', 'J': 'JJJJ', 'O': 'OOOO'}
>>> D = {c.lower(): c + '!' for c in ['MIELONKA', 'JAJKA', 'SZYNKA']}
>>> D
{'szynka': 'SZYNKA!', 'jajka': 'JAJKA!', 'mielonka': 'MIELONKA!}'
```

Słowniki składane są również użyteczne do inicjalizacji słowników z list, efektywnie działają tak samo jak metoda `fromkeys`, którą spotkaliśmy w poprzednim punkcie:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)           # Inicjalizacja słownika z listy kluczy
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = {k:0 for k in ['a', 'b', 'c']}            # To samo, ale z użyciem słownika składanego
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = dict.fromkeys('mielonka')                 # Inne iteratory, użycie wartości domyślnej
>>> D
{'a': None, 'p': None, 's': None, 'm': None}

>>> D = {k: None for k in ' mielonka '}
>>> D
{'a': None, 'p': None, 's': None, 'm': None}
```

Podobnie jak inne wyrażenia składające, słowniki składane pozwalają na zastosowanie dodatkowej składni, której nie demonstrowałem, między innymi chodzi o warunki `if`. Niestety, aby w pełni zrozumieć wyrażenia słowników składanych, należy poznać szczegóły instrukcji iteracyjnych i koncepcji iteratorów w Pythonie, a niestety w tym momencie nie posiadamy wystarczających informacji, aby w pełni przeanalizować tę tematykę. Wszelkie typy wyrażeń składanych (listy, zbiory i słowniki składane) przeanalizujemy szczegółowo w rozdziałach 14. i 20., zatem dalszą analizę odkładam na później. Funkcję wbudowaną `zip`, którą wykorzystywaliśmy w przykładach, przeanalizujemy w rozdziale 13, przy okazji omawiania pętli.

Widoki słowników

W Pythonie 3.0 metody słowników `keys`, `values` i `items` zwracają *obiekty widoków*, natomiast w 2.6 zwracane są listy. Obiekty widoków są *iteratorami*, czyli obiektami generującymi swoje wyniki po jednym elemencie, zamiast zwracać całą listę wyników jednorazowo w całości. Widoki słowników są obiektami iterowanymi, a dodatkowo są ściśle zintegrowane ze słownikami, z których powstały: zachowują oryginalną kolejność elementów słownika, odzwierciedlają zmiany wprowadzone w słowniku i obsługują operacje zbiorowe. Nie są jednak listami i nie obsługują takich operacji jak dostęp po indeksie czy sortowanie list. W przypadku bezpośredniego użycia w funkcji `print` nie wyświetlają swoich wyników.

W rozdziale 14. bardziej formalnie przeanalizujemy tematykę obiektów iterowanych, na potrzeby naszych aktualnych rozważań wystarczy wiedzieć, że przed wykonaniem operacji typowych dla list, lub w celu wyświetlania wyników w całości, wyniki metod `keys`, `values` i `items` należy przekształcić na listę za pomocą funkcji wbudowanej `list`.

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()                                     # Tworzy obiekt widoku w 3.0, nie listę
>>> K
<dict_keys object at 0x026D83C0>
>>> list(K)                                         # Wymusza listy w 3.0, o ile to konieczne
['a', 'c', 'b']

>>> V = D.values()                                    # To samo w przypadku widoków values i items
>>> V
<dict_values object at 0x026D8260>
```

```

>>> list(V)
[1, 3, 2]

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> K[0]                      # Operacje na listach nie działają, dopóki nie zostaną przekształcone
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'a'

```

Ta zmiana z reguły nie powinna być zauważalna, z wyjątkiem pracy w konsoli interaktywnej, gdy zechcemy wyświetlić wyniki na ekranie. W szczególności konstrukcje pętli w Pythonie działają podobnie jak w przypadku list, z tą różnicą, że kolejne wartości są generowanie w locie dla każdego przebiegu pętli.

```

>>> for k in D.keys(): print(k)      # Iteratory są automatycznie używane w pętlach
...
a
c
b

```

Dodatkowo, słowniki w 3.0 są same w sobie iteratorami zwracającymi kolejne klucze i, podobnie jak w 2.6, w wielu sytuacjach nie ma konieczności jawnego wywoływania metody keys.

```

>>> for key in D: print(key)        # Nie ma potrzeby wywoływać metody keys() do iteracji
...
a
c
b

```

W przeciwnieństwie do list zwracanych w 2.X, widoki słowników w 3.0 po utworzeniu nie są obiektami statycznymi: *dynamicznie odzwierciedlają zmiany wprowadzane w słowniku*.

```

>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()
>>> V = D.values()
>>> list(K)                      # Widok zachowuje tę samą kolejność elementów co słownik źródłowy
['a', 'c', 'b']
>>> list(V)
[1, 3, 2]

>>> del D['b']                  # Modyfikacja słownika
>>> D
{'a': 1, 'c': 3}

>>> list(K)                      # Zmiana jest uwzględniona w widoku
['a', 'c']
>>> list(V)                      # W 2.X tak się nie dzieje!
[1, 3]

```

Widoki słowników i operacje zbiorowe

Kolejna różnica Pythona 2.X w stosunku do 3.0 polega na tym, że wyniki metody keys zachowują się jak zbiory i obsługują wiele operacji typowych dla zbiorów, jak iloczyn czy unia zbiorów. W przypadku wyników metody values taka prawidłowość nie zachodzi, przede wszystkim dlatego, że wartości słowników nie są unikalne, w przeciwieństwie do kluczy, a w przypadku metody items operacje zbiorowe będą dostępne, jeśli pary (klucz, wartość) będą unikalne

i haszowalne. Zbiory zachowują się jak słowniki bez wartości (a w 3.0 wprowadzono nawet literał zbiorów wykorzystujący nawiasy klamrowe, podobnie jak literał słowników) taka symetria jest jak najbardziej logiczna. Podobnie jak klucze słowników, elementy zbiorów nie zachowują kolejności, są unikalne i niemutowalne.

Poniższy listing prezentuje zachowanie wyników metody keys w operacjach zbiorowych. W takim kontekście widoki słowników można łączyć wzajemnie z innymi widokami słowników, zbiorami i słownikami (w tym kontekście słowniki są traktowane jak widoki kluczów).

```
>>> K | {'x': 4}                                # Widoki keys (oraz niektóre widoki items) zachowują się jak zbiory
{'a', 'x', 'c'}

>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'

>>> D = {'a':1, 'b':2, 'c':3}
>>> D.keys() & D.keys()                      # Iloczyn widoków keys
{'a', 'c', 'b'}
>>> D.keys() & {'b'}
{'b'}
>>> D.keys() & {'b': 1}                        # Iloczyn widoku keys i słownika
{'b'}
>>> D.keys() | {'b', 'c', 'd'}                  # Unia widoku keys i zbioru
{'a', 'c', 'b', 'd'}
```

Widok items jest również obiektem zbiorowym, pod warunkiem że zawiera wyłącznie niemutowalne elementy.

```
>>> D = {'a': 1}
>>> list(D.items())                          # Widok items zachowuje się jak zbiór, jeśli jest haszowalny
[('a', 1)]
>>> D.items() | D.keys()                    # Unia widoku items i keys
{('a', 1), 'a'}
>>> D.items() | D                          # Słownik jest traktowany jak jego widok keys
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)}        # Zbiór par klucz/wartość
{('a', 1), ('d', 4), ('c', 3)}
>>> dict(D.items() | {('c', 3), ('d', 4)})    # Funkcja dict akceptuje również iterowane zbiory
{'a': 1, 'c': 3, 'd': 4}
```

Więcej informacji na temat operacji zbiorowych można znaleźć w rozdziale 5. Teraz przyjmyjmy się kilku zasadom użycia słowników w 3.0.

Sortowanie kluczów słowników

Metoda keys w 3.0 nie zwraca listy, więc tradycyjne (stosowane w 2.X) metody programowania stosowane w celu przeglądania słownika po posortowanej liście kluczów nie będą działać w 3.0. Wynik metody keys należy zatem przekodować ręcznie lub zastosować funkcję sorted opisaną w rozdziale 4. i na początku niniejszego rozdziału. Funkcja sorted zadziała na widoku keys, jak również na samym słowniku.

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> Ks = D.keys()                           # Sortowanie obiektu widoku nie działa!
>>> Ks.sort()
```

```

AttributeError: 'dict_keys' object has no attribute 'sort'

>>> Ks = list(Ks)                                # Przekształcenie na listę i posortowanie wyniku
>>> Ks.sort()
>>> for k in Ks: print(k, D[k])
...
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}                         # Można też użyć funkcji sorted() na kluczach
>>> Ks = D.keys()                               # Funkcja sorted() akceptuje dowolny obiekt iterowany
>>> for k in sorted(Ks): print(k, D[k])        # sorted() zwraca listę
...
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}                         # Jeszcze lepiej będzie posortować sam słownik
>>> for k in sorted(D): print(k, D[k])          # Iteratory słowników zwracają klucze
...
a 1
b 2
c 3

```

Porównywanie rozmiarów słowników nie działa

W Pythonie 2.6 mamy możliwość porównywania słowników z użyciem operatorów `<`, `>` itp. W takim przypadku Python porównuje ich rozmiary, ale w 3.0 taka operacja nie jest już obsługiwana. Zachowanie to można jednak zasymulować, porównując posortowane listy kluczów:

```
sorted(D1.items()) < sorted(D2.items())      # Odpowiednik D1 < D2 w 2.6
```

Operacja sprawdzenia równości jednak zadziała. Zagadnienie to będziemy omawiać szczegółowo w kolejnym rozdziale, zatem w tym miejscu nie będziemy się w nie zagłębiać.

Metoda `has_key` nie istnieje, niech żyje `in!`

Powszechnie stosowana we wcześniejszych wersjach Pythona metoda `has_key` przestała istnieć w 3.0. Zamiast niej należy stosować test przynależności `in` lub metodę `get` z wartością domyślną (operator `in` jest preferowany).

```

>>> D
{'a': 1, 'c': 3, 'b': 2}                         # Tylko dla 2.X: True/False
>>> D.has_key('c')
AttributeError: 'dict' object has no attribute 'has_key'

>>> 'c' in D
True
>>> 'x' in D
False
>>> if 'c' in D: print('present', D['c'])        # Metoda preferowana w 3.0
...
present 3

>>> print(D.get('c'))
3
>>> print(D.get('x'))

```

```
None
>>> if D.get('c') != None: print('present', D['c'])      # Inne rozwiązanie
...
present 3
```

Programiści używający 2.6, którzy chcą zachować zgodność z 3.0, powinni pamiętać, że pierwsze dwie nowości (słowniki składane i widoki) są dostępne wyłącznie w 3.0, ale pozostałe trzy (`sorted`, ręczne porównywanie słowników i operator `in`) mogą być stosowane w 2.6, co w przyszłości uprości migrację do Pythona 3.0.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy typy listy oraz słownika — dwa chyba najczęściej spotykane, najbardziej elastyczne i mające największe możliwości typy kolekcji spotykanych i używanych w kodzie napisanym w Pythonie. Dowiedzieliśmy się, że typ listy obsługuje uporządkowane kolekcje dowolnych obiektów, które można dowolnie zagnieżdżać. Lista może również na przykład rosnąć i kurczyć się na żądanie. Typ słownika jest podobny, jednak przechowuje elementy po kluczu, a nie ich pozycji, i nie zachowuje żadnego stałego uporządkowania elementów od lewej do prawej strony. Zarówno listy, jak i słowniki są typami zmiennymi, dlatego obsługują różnorodne operacje modyfikujące je w miejscu, jakie nie są dostępne dla łańcuchów znaków. Listy można na przykład rozszerzać za pomocą wywołań metody `append`, a słowniki poprzez przypisanie wartości do kluczy.

W kolejnym rozdziale zakończymy nasze pogłębione omówienie podstawowych typów danych, prezentując krotki oraz pliki. Potem przejdziemy do instrukcji kodujących logikę przetwarzającą nasze obiekty, co będzie kolejnym krokiem na drodze do tworzenia kompletnych programów. Zanim jednak zajmiemy się tymi zagadnieniami, pora przejść do quizu podsumowującego rozdział.

Sprawdź swoją wiedzę — quiz

1. Należy podać dwa sposoby utworzenia listy zawierającej pięć liczb całkowitych 0.
2. Należy podać dwa sposoby utworzenia słownika zawierającego dwa klucze ('`a`' oraz '`b`'), z których każdy ma przypisaną wartość 0.
3. Należy podać cztery operacje modyfikujące obiekt listy w miejscu.
4. Należy podać cztery operacje modyfikujące obiekt słownika w miejscu.

Sprawdź swoją wiedzę — odpowiedzi

1. Wyrażenie z literałem, takie jak `[0, 0, 0, 0, 0]`, i wyrażenie z powtórzeniem, takie jak `[0] * 5`, utworzą listę pięciu zer. W praktyce można również utworzyć taką listę za pomocą pętli, która rozpoczyna się od pustej listy i przy każdej iteracji dodaje do niej 0 — `L.append(0)`. Może tutaj zadziałać również lista składana (`[0 for i in range(5)]`), jednak jest to opcja dodatkowa.

2. Wyrażenie z literałem, takie jak `{'a': 0, 'b': 0}`, lub seria przypisań, jak `D = [0], D['a'] = 0, D['b'] = 0`, przydadzą się do utworzenia pożądanego słownika. Można również skorzystać z nowszej i prostszej do zakodowania formy ze słowami kluczowymi `dict(a=0, b=0)` lub bardziej elastycznej formy z sekwencjami klucz-wartość `dict([('a', 0), ('b', 0)])`. A także — ponieważ wszystkie klucze mają tę samą wartość — można skorzystać ze specjalnej formy `dict.fromkeys(['a', 'b'], 0)`.
3. Metody `append` i `extend` pozwalają rozszerzać listę w miejscu, metody `sort` i `reverse` sortują i odwracają listę, metoda `insert` wstawia element w podane miejsce, metody `remove` i `pop` usuwają element z listy po jego wartości i pozycji, instrukcja `del` usuwa element lub wycinek, natomiast instrukcje przypisania do indeksu i wycinka zastępują element lub cały fragment listy. Na potrzeby quzu wystarczy wybrać cztery odpowiedzi.
4. Słowniki zmienia się przede wszystkim poprzez przypisanie do nowego lub istniejącego klucza, co tworzy lub modyfikuje wpis klucza w tabeli. Instrukcja `del` usuwa wpis dla danego klucza, metoda słownika `update` łączy jeden słownik z drugim w miejscu, natomiast metoda `D.pop(klucz)` usuwa klucz i zwraca wartość, jaką on miał. Słowniki mają również inne, bardziej egzotyczne metody modyfikujące je w miejscu, które nie zostały wymienione w niniejszym rozdziale (na przykład `setdefault`). Więcej informacji na ten temat można znaleźć w innych materiałach źródłowych.

Krotki, pliki i pozostałe

Niniejszy rozdział kończy nasze pogłębione omówienie podstawowych typów obiektów w Pythonie, przedstawiając *krotkę* (kolekcję innych obiektów, która nie może być zmodyfikowana) oraz *plik* (interfejs do plików zewnętrznych na naszym komputerze). Jak zobaczymy niebawem, krotka to względnie prosty obiekt wykonujący operacje, które omówiliśmy już dla łańcuchów znaków oraz list. Obiekt pliku jest często używanym narzędziem służącym do przetwarzania plików. Podstawowe omówienie plików znajdujące się w niniejszym rozdziale zostanie wzbogacone większymi przykładami, które pojawią się w późniejszych rozdziałach książki.

Niniejszy rozdział kończy również tę część książki, przedstawiając właściwości wspólne dla wszystkich obiektów podstawowych, z jakimi się spotkaliśmy — koncepcje równości, porównania czy kopii obiektów. Krótko omówimy inne typy obiektów Pythona. Jak się okaże, choć przedstawiliśmy wszystkie najważniejsze typy wbudowane, koncepcja obiektów w Pythonie jest znacznie szersza, niż może to wynikać z dotychczasowych opisów. Zakończymy tę część książki, przyglądając się często popełnianym błędom związanym z typami obiektów i proponując kilka ćwiczeń, które pozwolą na własne eksperymenty z przedstawionymi tu koncepcjami.

Krotki

Ostatnim obiektem kolekcji w naszym przeglądzie typów podstawowych Pythona jest krotka (ang. *tuple*). Krotki tworzą proste grupy obiektów. Działają dokładnie tak jak listy, jednak nie mogą być modyfikowane w miejscu (są niezmienne) i zazwyczaj są zapisywane jako seria elementów w zwykłych nawiasach, a nie w nawiasach kwadratowych. Choć nie obsługują tak wielu metod, dzielą większość właściwości z listami. Poniżej znajduje się krótkie podsumowanie właściwości krotek.

Krotki są uporządkowanymi kolekcjami dowolnych obiektów

Tak jak łańcuchy znaków i listy, krotki są uporządkowanymi kolekcjami obiektów — zachowują zatem kolejność elementów od lewej do prawej strony. Tak jak listy mogą osadzać dowolny typ obiektu.

Dostęp do krotek odbywa się po wartości przesunięcia

Podobnie jak w przypadku łańcuchów znaków i list, dostęp do elementów krotki odbywa się za pomocą wartości przesunięcia (a nie klucza). Obsługują one wszystkie operacje oparte na wartościach przesunięcia, w tym indeksowanie i wycinki.

Krotki należą do kategorii niezmiennych sekwencji

Tak jak łańcuchy znaków oraz listy, krotki są sekwencjami — obsługują wiele z operacji odnoszących się do tych dwóch typów obiektów. Tak samo jak łańcuchy znaków, krotki są jednak niezmienne. Nie obsługują żadnych operacji modyfikujących je w miejscu, które mają zastosowanie do list.

Krotki mają stałą długość, są heterogeniczne i można je dowolnie zagnieżdżać

Ponieważ krotki są niezmienne, nie można zmieniać ich rozmiaru bez tworzenia kopii. Z drugiej strony, krotki mogą przechowywać dowolny typ obiektu, w tym inne obiekty złożone (na przykład listy, słowniki i inne krotki), dlatego obsługują dowolne zagnieżdżanie.

Krotki są tablicami referencji do obiektów

Tak jak listy, krotki najlepiej jest sobie wyobrazić jako tablice referencji do obiektów. Krotki przechowują punkty dostępu do innych obiektów (referencje), a indeksowanie krotek jest dość szybkie.

W tabeli 9.1 przedstawiono najczęściej wykonywane operacje na krotkach. Krotka zapisywana jest jako seria obiektów (z technicznego punktu widzenia: wyrażeń generujących obiekty) rozdzielonych przecinkami i zazwyczaj umieszczonych w zwykłych nawiasach. Pusta krotka to po prostu para nawiasów bez zawartości.

Tabela 9.1. Popularne literały oraz operacje na krotkach

Operacja	Interpretacja
()	Pusta krotka
T = (0,)	Krotka jednoelementowa (nie jest wyrażeniem)
T = (0, 'Ni', 1.2, 3)	Krotka czteroelementowa
T = 0, 'Ni', 1.2, 3	Inna krotka czteroelementowa (ta sama co wyżej)
T = ('abc', ('def', 'ghi'))	Zagnieżdżone krotki
T = tuple('mielonka')	Krotka elementów w obiekcie, na którym da się wykonywać iterację
T[i]	Indeks, indeks indeksu, wycinek, długość
T[i][j]	
T[i:j]	
len(t1)	
T1 + T2	Konkatenacja, powtórzenie
T2 * 3	
for x in T: print(x)	Iteracja, przynależność
'mielonka' in T	
[x ** 2 for x in T]	
T.index('Ni')	Metody z wersji 2.6 — wyszukiwanie, zliczanie
T.count('Ni')	

Krotki w akcji

Jak zawsze należy rozpocząć sesję interaktywną, by na własne oczy przekonać się, jak działają krotki. W tabeli 9.1 warto zwrócić uwagę na to, że krotki nie mają wszystkich metod posiadanych przez listy (zatem na przykład wywołanie append nie zadziała). Obsługują jednak zwykłe

operacje na sekwencjach, które widzieliśmy już przy okazji omawiania zarówno łańcuchów znaków, jak i list.

```
>>> (1, 2) + (3, 4)                                # Konkatenacja
(1, 2, 3, 4)

>>> (1, 2) * 4                                    # Powtórzenie
(1, 2, 1, 2, 1, 2, 1, 2)

>>> T = (1, 2, 3, 4)                             # Indeksowanie, wycinek
>>> T[0], T[1:3]
(1, (2, 3))
```

Właściwości składni krotek — przecinki i nawiasy

Drugi i czwarty wpis z tabeli 9.1 zasługują na słowo wyjaśnienia. Ponieważ nawiasy mogą również ograniczać wyrażenia (rozdział 5.), musimy jakoś przekazać Pythonowi, że pojedynczy obiekt w nawiasach jest obiektem krotki, a nie prostym wyrażeniem. Jeśli naprawdę chcemy utworzyć krotkę jednoelementową, wystarczy w nawiasach po tym pojedynczym elemencie dopisać przecinek.

```
>>> x = (40)                                     # Liczba całkowita
>>> x
40

>>> y = (40,)                                    # Krotka zawierająca liczbę całkowitą
>>> y
(40,)
```

W tym specjalnym przypadku Python pozwala również na pominięcie nawiasu otwierającego i zamkijającego dla krotki w kontekstach, w których nie jest to mylące składniowo. Czwarty wiersz tabeli 9.1 po prostu wymienia cztery elementy rozdzielone przecinkami. W kontekście instrukcji przypisania Python rozpozna taki zapis jako krotkę, nawet jeśli nie ma ona nawiasów.

Niektórzy zalecają, by krotki zawsze umieszczać w nawiasach. Inni mówią, żeby w krotkach nigdy nie używać nawiasów (a jeszcze inni mają własne życie i nie zajmują się mówieniem innym, co mają robić ze swoim!). Jedynym istotnym miejscem, w którym nawiasy są *obowiązkowo wymagane*, jest przekazywanie krotek jako literału w wywołaniu funkcji (gdzie nawiasy mają znaczenie) i kiedy są one wstawione do instrukcji `print` z Pythona 2.X (gdzie znaczenie mają przecinki).

Osobom poczatkującym łatwiej będzie chyba używać nawiasów, niż zastanawiać się nad tym, kiedy są one opcjonalne. Wielu programistów (ze mną na czele) uważa również, że nawiasy zwiększą czytelność skryptów, bo dzięki nim krotki są lepiej widoczne. Każdy może jednak mieć swoje zdanie.

Konwersje, metody oraz niezmienność

Poza różnicami w składni literałów operacje na krotkach (trzy środkowe wiersze z tabeli 9.1) są identyczne z operacjami na łańcuchach znaków i listach. Jedyna różnica, jaką warto podkreślić, polega na tym, że `+`, `*` oraz wycinki zwracają po zastosowaniu do krotek *nowe krotki*. Krotki nie udostępniają również tych samych metod, z jakimi spotkaliśmy się przy okazji omawiania łańcuchów znaków, list oraz słowników. Kiedy chcemy na przykład posortować krotkę, zazwyczaj musimy ją najpierw przekonwertować na listę w celu uzyskania dostępu do wywołania metody sortującej i zmodyfikowania obiektu na zmienny. Można także skorzystać z nowszej funkcji wbudowanej `sorted`, która przyjmuje dowolny obiekt sekwencji (a nawet jeszcze więcej).

```

>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)                                # Sporządzenie listy z elementów krotki
>>> tmp.sort()                                 # Sortowanie listy
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)                             # Sporządzenie krotki z elementów listy
>>> T
('aa', 'bb', 'cc', 'dd')

>>> sorted(T)                                  # Można także użyć funkcji wbudowanej sorted
['aa', 'bb', 'cc', 'dd']

```

W powyższym kodzie wbudowane funkcje list oraz tuple wykorzystano do przekształcenia obiektu na listę, a następnie konwersji tej listy z powrotem na krotkę. Oba wywołania tworzą nowe obiekty, jednak ich rezultat przypomina konwersję.

Do konwersji krotek można również wykorzystać listy składane. Poniższy kod robi z krotki listę, dodając przy okazji 20 do każdego elementu.

```

>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]

```

Listy składane są tak naprawdę operacjami na sekwencjach — zawsze tworzą nowe listy, jednak można je również wykorzystać do iteracji po dowolnych obiektach sekwencji, w tym krotkach, łańcuchach znaków i innych listach. Jak zobaczymy nieco później, działają nawet na niektórych elementach niebędących fizycznie przechowywanymi sekwencjami — wystarczą dowolne obiekty, po których można iterować, w tym pliki, które automatycznie wczytywane są wiersz po wierszu.

Choć krotki nie obsługują tych samych metod, co listy i łańcuchy znaków, w Pythonie 2.6 oraz 3.0 mają dwie własne metody. Metody index oraz count działają tak samo jak w przypadku list, jednak zdefiniowane są dla obiektów krotek.

```

>>> T = (1, 2, 3, 2, 4, 2)                      # Metody krotek w Pythonie 2.6 oraz 3.0
>>> T.index(2)                                 # Przesunięcie pierwszego wystąpienia obiektu 2
1
>>> T.index(2, 2)                            # Przesunięcie wystąpienia po przesunięciu 2
3
>>> T.count(2)                               # Ile jest obiektów 2?
3

```

Przed Pythonem 2.6 oraz 3.0 krotki nie miały żadnych metod — była to stara konwencja Pythona dla typów niezmiennych, która z powodu niepraktyczności została złamana wiele lat temu w przypadku łańcuchów znaków, a ostatnio — dla liczb oraz krotek.

Warto również zauważyć, że reguła dotycząca *niezmienności* krotki odnosi się jedynie do najwyższej jej poziomu, a nie do całej zawartości. Lista znajdująca się wewnętrz krotki może się na przykład normalnie zmieniać.

```

>>> T = (1, [2, 3], 4)                         # Nie działa — nie da się zmienić krotki
>>> T[1] = 'mielonka'
TypeError: object doesn't support item assignment

>>> T[1][0] = 'mielonka'                      # Działa — można modyfikować jej zmienne elementy
>>> T
(1, ['mielonka', 3], 4)

```

W większości programów taka niezmienna na najwyższym poziomie jest zupełnie wystarczająca w zastosowaniach, w których najczęściej pojawiają się krotki. Zagadnienie to przenosi nas również do kolejnego punktu.

Dlaczego istnieją listy i krotki?

To chyba pierwsze pytanie, jakie pojawia się przy omawianiu krotek — po co nam one, skoro mamy listy? Uzasadnienie jest częściowo historyczne — twórca Pythona jest z wykształcenia matematykiem i wspomina się, że krotki wyobrażały sobie jako proste powiązanie obiektów, natomiast listy — jako struktury danych zmieniające się z czasem. Tak naprawdę samo słowo „krotka” pochodzi z matematyki, podobnie jak jej częste zastosowanie dla wiersza w tabeli relacyjnej bazy danych.

Najtrudniejszą odpowiedzią na to pytanie wydaje się jednak to, że niezmiennaść krotek zapewnia pewien stopień *integralności*. Możemy być pewni, że krotka nie zostanie zmodyfikowana przez inną referencję umieszczoną gdzieś w programie; w przypadku list takiej gwarancji nie ma. Krotki pełnią zatem rolę deklaracji stałych z innych języków programowania, choć w Pythonie stałość powiązana jest z obiektami, a nie zmiennymi.

Krotki można również wykorzystywać w miejscach, w których nie można użyć list — na przykład jako klucze słowników (zobacz przykład rzadkiej macierzy w rozdziale 8.). Niektóre operacje wbudowane mogą również wymagać krotek, a nie list, choć w ostatnich latach wiele z takich operacji zostało uogólnionych. Generalnie listy są narzędziem wybieranym dla uporządkowanych kolekcji obiektów, które mogą się zmieniać. Krotki sprawdzają się w innych przypadkach stałych powiązań.

Pliki

Większość osób zna pliki, czyli nazwane pojemniki przechowujące dane na komputerze, którymi zarządza system operacyjny. Ostatnim wbudowanym typem obiektów, jaki będziemy omawiać, jest sposób uzyskiwania dostępu do tych plików z programów napisanych w Pythonie.

Mówiąc w skrócie, wbudowana funkcja `open` tworzy obiekt pliku Pythona, który służy jako łącze do pliku znajdującego się na komputerze. Po jej wywołaniu można przenieść łańcuchy znaków danych do powiązanego pliku zewnętrznego oraz z niego, wywołując metody obiektu zwartego pliku.

W porównaniu z typami, z jakimi się dotychczas spotkaliśmy, obiekty plików są w pewien sposób niezwykle. Nie są liczbami, sekwencjami ani odwzorowaniami; nie reagują na operatory wyrażeń. Zamiast tego eksportują jedynie metody służące do wykonywania popularnych zadań związanych z przetwarzaniem plików. Większość metod plików zajmuje się pobieraniem danych wejściowych z plików zewnętrznych i zapisywaniem do nich danych wyjściowych. Inne metody pozwalają na odszukanie nowej pozycji w pliku czy opróżnienie bufora wyjściowego. W tabeli 9.2 przedstawiono popularne operacje na plikach.

Tabela 9.2. Popularne operacje na plikach

Operacja	Interpretacja
output = open('C:\\spam', 'w')	Utworzenie pliku do zapisu ('w' pochodzi od ang. <i>write</i> — zapis)
input = open('data', 'r')	Utworzenie pliku do odczytu ('r' pochodzi od ang. <i>read</i> — odczyt)
input = open('data')	To samo co w poprzednim wierszu ('r' jest trybem domyślnym)
aString = input.read()	Wczytanie całego pliku do jednego łańcucha znaków
aString = input.read(N)	Wczytanie kolejnych N bajtów (jednego lub więcej) do łańcucha znaków
aString = input.readline()	Wczytanie kolejnego wiersza (wraz ze znacznikiem końca wiersza) do łańcucha znaków
aList = input.readlines()	Wczytanie całego pliku do listy łańcuchów znaków z poszczególnymi wierszami
output.write(aString)	Zapisanie łańcucha bajtów do pliku
output.writelines(aList)	Zapisanie do pliku wszystkich łańcuchów znaków z wierszami znajdującymi się w liście
output.close()	Ręczne zamknięcie (robione za nas, kiedy kończymy pracę z plikiem)
output.flush()	Opróżnienie bufora wyjściowego na dysk bez zamykania pliku
anyFile.seek(N)	Zmiana pozycji w pliku na wartość przesunięcia N dla następnej operacji
for line in open('data'): użycie line	Iteratory plików wchodzą wiersz po wierszu
open('f.txt', encoding='latin-1')	Pliki tekstowe Unicode Pythona 3.0 (łańcuchy znaków str)
open('f.bin', 'rb')	Pliki bajtów binarnych Pythona 3.0 (łańcuchy znaków bytes)

Otwieranie plików

By otworzyć plik, program wywołuje wbudowaną funkcję `open`, przekazując jej najpierw nazwę pliku zewnętrznego, a później *tryb* przetwarzania. Tryb ten to zazwyczaj łańcuch znaków 'r' dla odczytania pliku (wartość domyślna), 'w' dla utworzenia go i otwarcia do zapisu lub 'a' dla dodania tekstu na końcu pliku. Argument trybu przetwarzania może określać dodatkowe opcje:

- Dodanie b na końcu łańcucha trybu zezwala na dane *binarne* (wyłączane są tłumaczenia znaku końca wiersza oraz kodowanie Unicode z Pythona 3.0).
- Dodanie + oznacza, że plik otwierany jest *zarówno* do odczytu, jak i zapisu (możemy zatem odczytywać jeden obiekt pliku i do niego zapisywać, często w połączeniu z operacjami wyszukiwania służącymi do zmiany pozycji w pliku).

Oba argumenty funkcji `open` muszą być łańcuchami znaków Pythona, natomiast opcjonalny trzeci argument można wykorzystać do kontrolowania bufora wyjściowego — przekazanie zera oznacza, że dane wyjściowe nie będą buforowane (będą przenoszone do pliku zewnętrznego natychmiast po wywołaniu metody zapisu). Argument z nazwą pliku zewnętrznego może zawierać przedrostek z względną lub bezwzględną ścieżką do katalogu charakterystyczną dla danego systemu operacyjnego. Bez podania ścieżki do katalogu zakłada się, że plik znajduje się w bieżącym katalogu roboczym (czyli tam, gdzie wykonywany jest skrypt). Podstawy plików oraz proste przykłady omówimy tutaj, natomiast nie będziemy się zagłębiać w opcje trybu przetwarzania plików. Więcej szczegółów na ten temat można, jak zwykle, znaleźć w dokumentacji biblioteki standardowej.

Wykorzystywanie plików

Kiedy dzięki metodzie `open` mamy już utworzony obiekt pliku, możemy wywołać jego metody w celu wczytania danych z pliku zewnętrznego lub zapisania ich do niego. W każdym z tych przypadków tekst pliku przybiera w programach w Pythonie postać łańcuchów znaków. Wczytanie pliku powoduje zwrócenie jego tekstu osadzonego w łańcuchach znaków, a do metod zapisujących dane do plików przekazuje się łańcuchy znaków. Metody wczytujące i zapisujące dane mają wiele odmian; w tabeli 9.2 zaprezentowano te najbardziej popularne. Poniżej znajduje się kilka najważniejszych uwag dotyczących używania plików.

Do wczytywania wierszy najlepiej nadają się iteratory plików

Choć wymienione w tabeli metody służące do odczytu i zapisu są dość popularne, warto pamiętać, że chyba najlepszym sposobem na wczytanie wierszy z pliku tekstowego wcale nie jest wczytanie pliku. Jak zobaczymy w rozdziale 14., pliki mają również *iterator* automatycznie wczytyujący po jednym wierszu na raz w pętli `for`, liście składanej czy innym kontekście iteracyjnym.

Zawartość pliku to łańcuchy znaków, a nie obiekty

W tabeli 9.2 warto zwrócić uwagę na to, że dane wczytane z pliku zawsze trafiają do skryptu w postaci łańcucha znaków. Jeśli zatem chcemy korzystać z innego typu obiektu Pythona, będziemy musieli najpierw dokonać konwersji. I podobnie — w przeciwieństwie do sytuacji z operacją `print` — Python przy zapisie danych do pliku nie dodaje żadnego formatowania i nie przekształca obiektów na łańcuchy znaków automatycznie; do pliku należy przesyłać uprzednio sformatowany łańcuch znaków. Z tego powodu poznane wcześniej narzędzia (jak `int`, `float`, `str` i wyrażenia formatujące łańcuchy znaków) konwertujące obiekty na łańcuchy znaków (i odwrotnie) przydadzą się także przy pracy z plikami. Python zawiera również zaawansowane narzędzia z biblioteki standardowej pozwalające na obsługę uniwersalnego przechowywania obiektów (na przykład moduł `pickle`) i radzenie sobie ze spakowanymi danymi binarnymi z plików (jak moduł `struct`). Z obydwoma modułami spotkamy się w dalszej części rozdziału.

Wywołanie close jest zazwyczaj opcjonalne

Wywołanie metody pliku `close` kończy nasze połączenie z plikiem zewnętrznym. Jak wspomniano w rozdziale 6., w Pythonie miejsce zajmowane w pamięci przez obiekt jest automatycznie zwalniane, kiedy do tego obiektu nie istnieją już żadne referencje w programie. Kiedy obiekty plików są zwalniane, Python automatycznie zamyka również same pliki, jeśli są one nadal otwarte (to samo ma miejsce, gdy zamykamy program). Oznacza to, że nie zawsze musimy ręcznie zamykać wszystkie pliki, w szczególności w krótkich skryptach, które nie działają zbyt długo. Z drugiej strony, ręczne wywołanie metody `close` nie jest dużym obciążeniem i w większych systemach jest dobrym pomysłem. Ściśle mówiąc, opcja automatycznego zamykania w momencie, gdy skończymy pracę z plikiem, nie jest częścią definicji języka i może z czasem się zmienić. Z tego powodu ręczne wywoływanie metody `close` jest dobrym obyczajem. Alternatywny sposób zagwarantowania automatycznego zamykania plików znajduje się w dalszym omówieniu *menedżerów kontekstu* obiektu pliku, wykorzystywanych w Pythonie 2.6 oraz 3.0 w połączeniu z nową instrukcją `with/as`.

Pliki są buforowane i można je przeszukiwać

Uwagi z poprzedniego akapitu dotyczące zamykania plików są istotne, ponieważ zamknienie zarówno zwalnia zasoby systemu operacyjnego, jak i opróżnia bufory wyjścia.

Domyślnie pliki wyjścia są zawsze buforowane, co oznacza, że tekst, który piszemy, może nie być natychmiast przenoszony z pamięci na dysk. Zamknięcie pliku lub wykonanie jego metody `flush` wymusza transfer danych na dysk. Buforowanie można wyłączyć za pomocą dodatkowych argumentów metody `open`, jednak może to mieć negatywny wpływ na wydajność. Python oferuje swobodny dostęp do plików z możliwością odwołania się do każdego bajta w pliku — metoda plików `seek` pozwala skryptom na przeskakiwanie z wczytywaniem i zapisywaniem w określonych pozycjach.

Pliki w akcji

Przyjrzymy się prostemu przykładowi demonstrującemu podstawy przetwarzania plików. Poniższy kod rozpoczyna się od otwarcia nowego pliku tekstowego do zapisu, później zapisuje do tego pliku dwa wiersze (łańcuchy znaków zakończone znacznikiem nowego wiersza `\n`), a następnie plik został zamknięty. Potem kod ponownie otwiera ten sam plik, tym razem w trybie do odczytu, i wczytuje umieszczone w nim wiersze — za pomocą metody `readline`, każdorazowo po jednym. Warto zwrócić uwagę na to, że trzecie wywołanie metody `readline` zwraca pusty łańcuch znaków. W ten sposób metody plików Pythona przekazują nam, że osiągnęliśmy koniec pliku (puste wiersze z pliku zwracane są jako łańcuchy znaków zawierające znak nowego wiersza, a nie jako puste łańcuchy znaków). Poniżej widać kod przykładu.

```
>>> myfile = open('myfile.txt', 'w')           # Otwarcie do zapisu tekstu (tworzy pusty plik)
>>> myfile.write('witaj, pliku tekstowy\n')    # Zapisanie wiersza tekstu
22
>>> myfile.write('żegnaj, pliku tekstowy\n')
23
>>> myfile.close()                           # Zrzucenie bufora wyjściowego na dysk
>>> myfile = open('myfile.txt')               # Otwarcie do odczytu tekstu — 'r' jest domyślne
>>> myfile.readline()                      # Wczytanie wierszy z powrotem
'witaj, pliku tekstowy\n'
>>> myfile.readline()
'żegnaj, pliku tekstowy\n'
>>> myfile.readline()                      # Pusty łańcuch znaków — koniec pliku
..
```

Warto zauważyc, że wywołania metody `write` pliku zwracają w Pythonie 3.0 liczbę znaków. W wersji 2.6 tak nie jest, dlatego w sesji interaktywnej nie zobaczymy tych liczb. Powyższy przykład zapisuje każdy z wierszy tekstu (wraz ze znacznikiem kończącym wiersz `\n`) w postaci łańcucha znaków. Metody zapisujące do plików nie dodają znaku końca wiersza za nas, dlatego musimy to zrobić sami, by w poprawny sposób zakończyć wiersz tekstu. Jeśli tego nie zrobimy, kolejny zapis rozszerzy aktualny wiersz pliku.

Jeśli chcemy wyświetlić zawartość pliku ze zinterpretowanymi znakami końca wiersza, należy wczytać cały plik do jednego łańcucha znaków *naraz* za pomocą metody `read` obiektu pliku, a następnie wyświetlić go.

```
>>> open('myfile.txt').read()                # Wczytanie wszystkiego naraz do łańcucha znaków
'witaj, pliku tekstowy\nżegnaj, pliku tekstowy\n'
>>> print(open('myfile.txt').read())          # Sposób wyświetlania przyjazny dla użytkownika
witaj, pliku tekstowy
żegnaj, pliku tekstowy
```

Jeśli natomiast chcemy przejrzeć plik tekstowy wiersz po wierszu, najlepszą opcją jest często skorzystanie z iteratorów plików.

```
>>> for line in open('myfile'):
...     print(line, end='')
...
witaj, pliku tekstowy
żegnaj, pliku tekstowy
```

Użycie iteratorów plików, a nie wczytywania

Przy takim kodzie tymczasowy obiekt pliku utworzony za pomocą metody open automatycznie wczyta i zwróci po jednym wierszu z każdą iteracją pętli. Forma ta jest zazwyczaj najłatwiejsza do utworzenia w kodzie, charakteryzuje się nieźleym zużyciem pamięci i może być szybsza od części innych rozwiązań (co oczywiście uzależnione jest od innych czynników). Ponieważ nie dotarliśmy jeszcze do instrukcji ani iteratorów, na pełne wyjaśnienie powyższego kodu będziemy musieli poczekać do rozdziału 14.

Pliki tekstowe i binarne w Pythonie 3.0

Uściślając, przykład z poprzedniego podrozdziału wykorzystuje pliki tekstowe. W Pythonie 3.0 oraz 2.6 typ pliku określa się za pomocą drugiego argumentu metody open, łańcucha trybu. Dołączenie do wywołania łańcucha znaków „b” oznacza tryb binarny. Python zawsze obsługiwał zarówno pliki tekstowe, jak i binarne, jednak w wersji 3.0 rozróżnienie między nimi jest znacznie mocniejsze:

- *Pliki tekstowe* reprezentują zawartość w postaci normalnych łańcuchów znaków str, wykonując automatycznie kodowanie i dekodowanie Unicode oraz domyślnie dokonując przekładów końca wierszy.
- *Pliki binarne* reprezentują zawartość w postaci specjalnego typu łańcuchów znaków o nazwie bytes i pozwalają programom na dostęp do niezmienionej zawartości plików.

W przeciwieństwie do tego pliki tekstowe Pythona 2.6 obsługują zarówno tekst 8-bitowy, jak i dane binarne; specjalny typ łańcucha znaków oraz interfejs plików (łańcuchy unicode oraz metoda codecs.open) obsługują tekst Unicode. Różnice w Pythonie 3.0 wynikają z faktu, że tekst zwykły oraz Unicode zostały połączone w jeden typ normalnego łańcucha znaków — co ma sens, biorąc pod uwagę, że cały tekst, w tym ASCII oraz inne 8-bitowe systemy kodowania, jest kodem Unicode.

Ponieważ spora część programistów ma do czynienia jedynie z tekstem ASCII, wystarczające będą dla nich podstawowy interfejs plików tekstowych, wykorzystany w poprzednim przykładzie, oraz normalne łańcuchy znaków. W Pythonie 3.0 wszystkie łańcuchy znaków są Unicode, czego użytkownicy ASCII zazwyczaj nawet nie zauważą. Tak naprawdę pliki i łańcuchy znaków działają w wersjach 3.0 oraz 2.6 tak samo, jeśli zakres naszego pliku ograniczony jest do prostych form tekstu.

Jeśli musimy obsłużyć aplikacje zinteryonalizowane lub dane bajtowe, różnice z Pythonem 3.0 wpłyną jednak na nasz kod (zazwyczaj na lepsze). Mówiąc ogólnie, dla plików binarnych musimy użyć łańcuchów znaków bytes; zwykłe łańcuchy znaków str przeznaczone są dla plików tekstowych. Co więcej, ponieważ pliki tekstowe implementują kodowanie Unicode, nie można otworzyć pliku z danymi binarnymi w trybie tekstowym — dekodowanie jego zawartości na tekst Unicode z dużym prawdopodobieństwem się nie powiedzie.

Przyjrzyjmy się przykładowi. Po wczytaniu pliku z danymi binarnymi otrzymujemy obiekt bytes — sekwencję niewielkich liczb całkowitych reprezentujących bezwzględne wartości bajtowe (które mogą, ale nie muszą, odpowiadać znakom). Wygląda ona i działa prawie dokładnie tak samo jak normalny łańcuch znaków.

```

>>> data = open('data.bin', 'rb').read()          # Otwarcie pliku binarnego: rb = „read binary”
>>> data                                       # Łąćuch bytes przechowuje dane binarne
b'\x00\x00\x00\x07mielonka\x00\x08'
>>> data[4:12]                                    # Działa jak łańcuch znaków
b'mielonka'
>>> data[4:12][0]                                # Ale tak naprawdę to małe 8-bitowe liczby całkowite
109
>>> bin(data[4:12][0])                          # Funkcja bin z Pythona 3.0
'0b1101101'

```

Pliki binarne nie wykonują żadnego przekładu końca wierszy na danych. Pliki tekstowe przy transferze domyślnie odwzorowują wszystkie formy na i z \n przy zapisie i odczytanie kodowania Unicode. Ponieważ dane Unicode oraz binarne są mało interesujące dla wielu programistów Pythona, pełne ich omówienie odłożymy do rozdziału 36. Na razie przejdźmy do nieco większych przykładów plików.

Przechowywanie obiektów Pythona w plikach i przetwarzanie ich

Kolejny przykład zapisuje różne obiekty Pythona do pliku tekstowego z kilkoma wierszami. Warto pamiętać, że kod ten musi przekształcić obiekty na łańcuchy znaków za pomocą odpowiednich narzędzi konwersji. Powtórzmy raz jeszcze: dane z plików są zawsze w skryptach łańcuchami znaków, natomiast metody zapisu nie formatują za nas obiektów na łańcuchy znaków. Z uwagi na brak miejsca pomijam odtąd wartości zliczania bajtów zwracane z metod write.

```

>>> X, Y, Z = 43, 44, 45                      # Obiekty Pythona muszą być łańcuchami znaków,
>>> S = 'Mielonka'                           # by można je było umieścić w plikach
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w')              # Utworzenie pliku wyjściowego
>>> F.write(S + '\n')                         # Zakończenie wierszy za pomocą \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z))        # Konwersja liczb na łańcuchy znaków
>>> F.write(str(L) + '$' + str(D) + '\n')      # Konwersja i rozdzielenie znakiem $
>>> F.close()

```

Po utworzeniu pliku możemy sprawdzić jego zawartość, otwierając go i wczytując do łańcucha znaków (w jednej operacji). Warto zwrócić uwagę, że w sesji interaktywnej zwracana jest dokładna zawartość bajtowa, natomiast operacja print interpretuje osadzone znaki końca wiersza, tak by wynik był bardziej przyjazny dla użytkownika.

```

>>> chars = open('datafile.txt').read()          # Wyświetlenie „surowego” łańcucha znaków
>>> chars
'Mielonka\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n'
>>> print(chars)                             # Wyświetlenie w sposób przyjazny dla użytkownika
Mielonka
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}

```

Teraz musimy skorzystać z narzędzi konwersji w celu przekształcenia łańcuchów znaków z pliku tekstowego na prawdziwe obiekty Pythona. Ponieważ Python nigdy nie konwertuje łańcuchów znaków na liczby czy inne typy obiektów w sposób automatyczny, jest to konieczne, jeśli chcemy uzyskać dostęp do normalnych narzędzi obiektów, takich jak indeksowanie czy dodawanie.

```

>>> F = open('datafile.txt')                    # Ponowne otwarcie pliku
>>> line = F.readline()                       # Wczytanie jednego wiersza
>>> line

```

```
'Mielonka\n'  
>>> line.rstrip()  
'Mielonka'
```

Usunięcie znaku końca wiersza

W przypadku pierwszego wiersza do pozbycia się końcowego znaku nowego wiersza wykorzystaliśmy metodę `rstrip`. Zadziałałby również wycinek `line[:-1]`, gdybyśmy byli pewni, że wszystkie wiersze mają na końcu znak `\n` (czasami brakuje go w ostatnim wierszu).

Jak na razie udało nam się wczytać wiersz zawierający łańcuch znaków. Teraz pobierzemy kolejny wiersz, zawierający liczby, i dokonamy ekstrakcji obiektów z tego wiersza.

```
>>> line = F.readline()  
>>> line  
'43,44,45\n'  
>>> parts = line.split(',')  
>>> parts  
['43', '44', '45\n']
```

Kolejny wiersz pliku

Tutaj jest on łańcuchem znaków

Podział na przecinkach

Tutaj metodę łańcuchów znaków `split` wykorzystaliśmy do pocięcia wiersza w miejscu wystąpienia przecinków. W rezultacie otrzymujemy listę podłańcuchów zawierających poszczególne liczby. Gdybyśmy chcieli na nich wykonać jakieś działania matematyczne, nadal musielibyśmy przekształcić je z łańcuchów znaków na liczby.

```
>>> int(parts[1])  
44  
>>> numbers = [int(P) for P in parts]  
>>> numbers  
[43, 44, 45]
```

Konwersja z łańcucha na liczbę

Konwersja wszystkich liczb naraz

Jak już wiemy, metoda `int` przekształca łańcuch cyfr w obiekt liczby całkowitej, a wyrażenie listy składanej z rozdziału 4. można wykorzystać do wywołania wszystkich elementów listy naraz (więcej informacji na temat list składanych znajdziesz się w dalszej części książki). Warto zauważyc, że nie musielibyśmy wywoływać metody `rstrip` w celu usunięcia znaku `\n` z końca ostatniego elementu — `int` i inne konwertery ignorują białe znaki znajdujące się wokół cyfr.

Wreszcie, by przekształcić listę oraz słownik z trzeciego wiersza pliku, możemy wywołać wbudowaną funkcję `eval`, która traktuje łańcuch jako fragment kodu wykonywalnego (z technicznego punktu widzenia: łańcuch znaków zawierający wyrażenie Pythona).

```
>>> line = F.readline()  
>>> line  
"[1, 2, 3]${'a': 1, 'b': 2}\n"  
>>> parts = line.split('$')  
>>> parts  
[['1', '2', '3'], {"'a': 1, 'b': 2}\n"]  
>>> eval(parts[0])  
[1, 2, 3]  
>>> objects = [eval(P) for P in parts]  
>>> objects  
[[1, 2, 3], {'a': 1, 'b': 2}]
```

Podział na znaku \$

Konwersja na dowolny typ obiektu

To samo w jednej liście

Ponieważ w rezultacie analizy składniowej i konwersji otrzymujemy listę normalnych obiektów Pythona w miejsce łańcuchów znaków, teraz możemy na tych obiektach zastosować zwykłe operacje na listach i słownikach.

Przechowywanie obiektów Pythona za pomocą pickle

Zademonstrowane w ostatnim kodzie wykorzystanie funkcji `eval` do konwersji łańcuchów znaków na obiekty ma duże możliwości. Tak naprawdę — czasem nawet *za duże*. Funkcja `eval` z radością wykona dowolne wyrażenie Pythona, nawet takie, które kasuje wszystkie pliki

z naszego komputera — pod warunkiem że ma do tego odpowiednie uprawnienia. Jeśli jednak naprawdę chcemy przechowywać obiekty Pythona, a nie możemy ufać źródłu danych z pliku, idealnie przyda nam się moduł `pickle` z biblioteki standardowej.

Moduł `pickle` jest zaawansowanym narzędziem pozwalającym na przechowywanie prawie każdego obiektu Pythona bezpośrednio w pliku bez konieczności dokonywania konwersji na łańcuch znaków i z niego. Jest bardzo ogólnym narzędziem do formatowania i przetwarzania danych. By na przykład przechować w pliku słownik, można to zrobić w sposób bezpośredni za pomocą modułu `pickle`.

```
>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                                     # Serializacja dowolnego obiektu w pliku
>>> F.close()
```

By później ponownie otrzymać słownik, wystarczy odtworzyć go raz jeszcze za pomocą `pickle`.

```
>>> F = open('datafile.pkl', 'rb')                         # Załadowanie dowolnego obiektu z pliku
>>> E = pickle.load(F)
>>> E
{'a': 1, 'b': 2}
```

Otrzymamy odpowiedni obiekt słownika, przy którym nie trzeba ręcznie dokonywać podziałów lub konwersji. Moduł `pickle` wykonuje coś, co znane jest jako *serializacja obiektów* — konwersja obiektów na łańcuchy bajtów i z powrotem. Jego zaletą jest to, że wymaga niewiele pracy z naszej strony. Moduł ten wewnętrznie tłumaczy nasz słownik na postać łańcucha znaków, choć nie wygląda to zbyt ciekawie przy odczycie (i może się jeszcze różnić przy użyciu innych trybów protokołu danych).

```
>>> open('datafile.pkl', 'rb').read()
b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'
```

Ponieważ `pickle` potrafi zrekonstruować obiekt z tego formatu, nie musimy się tym zajmować samodzielnie. Więcej informacji na temat modułu `pickle` można znaleźć w dokumentacji biblioteki standardowej Pythona lub importując `pickle` i przekazując ten moduł do funkcji `help` w sesji interaktywnej. Przy okazji warto również zapoznać się z modułem `shelve`. Jest to narzędzie wykorzystujące moduł `pickle` do przechowania obiektów Pythona w systemie plików dostępnych po kluczu, co znacznie wykracza poza omawiane tu zagadnienia. Przykład działania modułu `shelve` zobaczymy jednak w rozdziale 27., natomiast inne przykłady zastosowania modułu `pickle` — w rozdziałach 30. oraz 36.

Przechowywanie i przetwarzanie spakowanych danych binarnych w plikach

I jeszcze jedna uwaga związana z plikami, o której warto wspomnieć przed przejściem dalej. Niektóre zaawansowane aplikacje mają również do czynienia ze spakowanymi danymi binarnymi, utworzonymi na przykład przez program napisany w języku C. Biblioteka standardowa Pythona zawiera narzędzie pomocne w takiej sytuacji — moduł `struct` potrafi zarówno tworzyć, jak i przetwarzać spakowane dane binarne. W pewnym sensie jest on kolejnym narzędziem do konwersji danych, które interpretuje łańcuchy znaków w plikach jako dane binarne.

By na przykład utworzyć plik ze spakowanymi danymi binarnymi, należy otworzyć go w trybie '`wb`' (zapis binarny) i przekazać do `struct` formatujący łańcuch znaków wraz z obiektami Pythona. Zastosowany poniżej formatujący łańcuch znaków oznacza spakowanie danych



Skoro już otwarłem plik wykorzystywany do przechowania zserializowanego obiektu w *trybie binarnym*: tryb ten jest zawsze wymagany w Pythonie 3.0, ponieważ moduł `pickle` tworzy i wykorzystuje obiekt łańcucha znaków `bytes`. Obiekty tego typu wymuszają pliki w trybie binarnym (w wersji 3.0 pliki w trybie tekstowym wymuszają z kolei łańcuchy znaków `str`). We wcześniejszych wersjach Pythona użycie plików w trybie tekstowym dla protokołu 0 (domyślnego, tworzącego tekst ASCII) było w porządku, o ile tryb ten był wykorzystywany w sposób spójny. Wyższe protokoły wymagały plików w trybie binarnym. Domyślnym protokołem Pythona 3.0 jest 3 (binarny), jednak obiekt `bytes` wykorzystywany jest nawet dla protokołu 0.Więcej informacji na ten temat można znaleźć w rozdziale 36., dokumentacji biblioteki standardowej Pythona lub innych książkach.

Python 2.6 zawiera również moduł `cPickle`, będący zoptymalizowaną wersją modułu `pickle`; można go importować w sposób bezpośredni z uwagi na szybkość działania. W Pythonie 3.0 nazwa tego modułu została zmieniona na `_pickle`; jest on wykorzystywany automatycznie przez `pickle` — skrypty po prostu importują moduł `pickle` i pozwalają Pythonowi na samodzielna optymalizację.

w postaci czterobajtowej liczby całkowitej, czteroznakowego łańcucha i dwubajtowej liczby całkowitej z malejącym porządkiem bitów (inne kody formatów obsługują bajty dopełniające czy liczby zmiennoprzecinkowe).

```
>>> F = open('data.bin', 'wb')                                # Otwarcie pliku binarnego do zapisu
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'jajo', 8) # Utworzenie pakowanych danych binarnych
>>> data
'\x00\x00\x00\x07jajo\x00\x08'
>>> F.write(data)                                         # Zapisanie łańcucha bajtowego
>>> F.close()
```

Python tworzy binarny łańcuch danych `bytes`, który zapisujemy do pliku w normalny sposób (składa się on głównie ze znaków niedrukowanych zapisanych w szesnastkowych sekwencjach ucieczki). By przetworzyć te wartości na normalne obiekty Pythona, wystarczy z powrotem wczytać łańcuch i rozpakować go z użyciem tego samego formatującego łańcucha znaków. Python dokonuje ekstrakcji wartości na normalne obiekty Pythona (liczby całkowite i łańcuchy znaków).

```
>>> F = open('data.bin', 'rb')                                # Pobranie pakowanych danych binarnych
>>> data = F.read()
>>> data
'\x00\x00\x00\x07jajo\x00\x08'
>>> values = struct.unpack('>i4sh', data)      # Konwersja na obiekty Pythona
>>> values
(7, 'jajo', 8)
```

Pliki z danymi binarnymi są zaawansowanymi narzędziami niskiego poziomu, których nie będziemy tutaj omawiali bardziej szczegółowo.Więcej informacji na ich temat można znaleźć w rozdziale 36., a także dokumentacji biblioteki standardowej Pythona lub po zimportowaniu modułu `struct` i interaktywnym przekazaniu go do funkcji `help`. Warto również zauważyć, że binarne tryby przetwarzania plików '`wb`' i '`rb`' można wykorzystać do przetworzenia prostszego pliku binarnego, takiego jak obrazek czy plik dźwiękowy, jako całości bez konieczności rozpakowywania jego zawartości.

Menedżery kontekstu plików

Warto również pamiętać o omówieniu obsługi *menedżerów kontekstu* plików z rozdziału 33. — nowości w Pythonie 3.0 oraz 2.6. Choć jest to raczej opcja z dziedziny przetwarzania wyjątków niż samych plików, pozwala nam opakować kod przetwarzający pliki w warstwę logiki pozwalającą dopilnować, że plik przy wyjściu z niego będzie automatycznie zamykany — nie musimy więc polegać na automatycznym zamknięciu w momencie czyszczenia pamięci.

```
with open(r'C:\misc\data.txt') as myfile:    # Więcej informacji w rozdziale 33.  
    for line in myfile:  
        ...tutaj użycie line...
```

Omówiona w rozdziale 33. instrukcja `try/finally` udostępnia podobną funkcjonalność, jednak kosztem dodatkowego kodu — a dokładnie mówiąc, trzech dodatkowych wierszy. Często możemy jednak uniknąć obu rozwiązań i pozwolić Pythonowi na automatyczne zamykanie plików za nas.

```
myfile = open(r'C:\misc\data.txt')  
try:  
    for line in myfile:  
        ...tutaj użycie line...  
finally:  
    myfile.close()
```

Ponieważ obie powyższe opcje wymagają większej ilości informacji, niż przedstawiono dotychczas, ich szczegółowe omówienie odłożymy do dalszej części książki.

Inne narzędzia powiązane z plikami

Istnieją również dodatkowe, bardziej zaawansowane metody plików zaprezentowane w tabeli 9.2, a nawet takie, których w tej tabeli nie ma. Wspomniana wcześniej metoda `seek` ponownie ustawia aktualną pozycję w pliku (kolejna operacja odczytu lub zapisu zacznie się w tym miejscu), a `flush` wymusza zapisanie zbuforowanych danych wyjściowych na dysku (domyślnie pliki zawsze są buforowane).

Kompletną listę metod plików można znaleźć w bibliotece standardowej oraz w książkach wymienionych w „Przedmowie”. Można również wywołać interaktywnie funkcje `dir` bądź `help`, przekazując im otwarty obiekt pliku (w Pythonie 2.6, ale już nie 3.0, można także przekazać nazwę `file`). Więcej przykładów przetwarzania plików można znaleźć w ramce „Znaczenie skanerów plików” w rozdziale 13. Przedstawia ona popularne wzorce kodu pętli skanującej pliki zawierające instrukcje, których jeszcze nie omawialiśmy.

Warto również zauważyć, że choć funkcja `open` oraz zwracane przez nią obiekty plików są naszym głównym interfejsem do plików zewnętrznych ze skryptów Pythona, w zestawie narzędzi Pythona istnieją również inne sposoby. Dostępne są na przykład:

Strumienie standardowe

Otwarte obiekty plików w module `sys`, takie jak `sys.stdout` (więcej informacje w podrozdziale „Polecenia print” w rozdziale 11.).

Pliki deskryptorów z modułu `os`

Uchwyty do plików obsługujące narzędzia niższego poziomu, jak blokowanie pliku.

Gniazda, potoki, pliki FIFO

Obiekty podobne do plików, wykorzystywane do synchronizacji procesów lub komunikacji za pośrednictwem sieci.

Pliki z dostępem po kluczu z modułu shelve

Wykorzystywane do bezpośredniego przechowywania niezmienionych obiektów Pythona po kluczu (użyte w rozdziale 27.).

Strumienie poleceń powłoki

Narzędzia takie jak `os.popen` czy `subprocess.Popen`, obsługujące tworzenie poleceń powłoki oraz wczytywanie ich strumieni standardowych oraz zapisywanie do nich.

W domenie open source można znaleźć jeszcze więcej narzędzi przypominających pliki, w tym obsługę komunikacji z portami szeregowymi za pomocą rozszerzenia *PySerial* czy programy interaktywne, takie jak system *pexpect*. Dodatkowe informacje na temat takich narzędzi można znaleźć w bardziej zaawansowanych tekstuach poświęconych Pythonowi oraz w Internecie.



Uwaga na temat wersji: W Pythonie 2.5 oraz wersjach wcześniejszych wbudowana nazwa `open` jest właściwie synonimem nazwy `file`. Wiele plików można otwierać za pomocą wywołania albo `open`, albo `file` (choć w przypadku otwierania opcją preferowaną jest `open`). W Pythonie 3.0 nazwa `file` nie jest już dostępna z powodu jej pokrywania się z `open`.

Użytkownicy Pythona 2.6 mogą także użyć nazwy `file` jako typu obiektu pliku w celu dostosowania plików do własnych potrzeb za pomocą programowania zorientowanego obiektywnego (opisanego w dalszej części książki). W Pythonie 3.0 pliki drastycznie się zmieniły. Klasy wykorzystywane do implementacji obiektów plików znajdują się w module biblioteki standardowej `io`. Wskazówki można znaleźć w dokumentacji tego modułu lub kodzie klas udostępnianych przez niego na potrzeby dostosowania do własnych potrzeb, a także po wykonaniu wywołania `type(F)` na otwartym pliku `F`.

Raz jeszcze o kategoriach typów

Po zapoznaniu się ze wszystkimi podstawowymi typami wbudowanymi Pythona warto zakończyć omówienie ich, przyglądając się właściwościom dla nich wspólnym. W tabeli 9.3 sklasyfikowano wszystkie najważniejsze omówione typy zgodnie z wprowadzonymi wcześniej kategoriami typów. Oto kilka uwag ogólnych, o których warto pamiętać:

- Obiekty współdzielą operacje zgodnie z kategoriami. Przykładowo łańcuchy znaków, listy oraz krótki współdzielą operacje na sekwencjach, takie jak konkatenacja, obliczenie długości czy indeksowanie.
- Modyfikowane w miejscu mogą być jedynie obiekty zmienne (listy, słowniki oraz zbiory); nie można tego zrobić w przypadku liczb, łańcuchów znaków czy krotek.
- Pliki jedynie eksportują metody, dlatego kwestia zmienności ich nie dotyczy. Ich stan może się zmieniać przy przetwarzaniu, jednak nie ma to wiele wspólnego z ograniczeniami związany z typami Pythona.
- „Liczby” z tabeli 9.3 obejmują wszystkie typy liczb — całkowite (oraz odrębny typ długiej liczby całkowitej z Pythona 2.6), zmiennoprzecinkowe, zespolone, dziesiętne oraz ułamkowe.
- „Łańcuchy znaków” z tabeli 9.3 obejmują typ `str`, a także `bytes` z Pythona 3.0 oraz `Unicode` z wersji 2.6. Typ łańcucha znaków `bytearray` z wersji 3.0 jest zmienny.
- Zbiory przypominają nieco klucze słownika bez wartości, jednak nie odwzorowują wartości i nie są uporządkowane, dlatego zbiory nie są ani odwzorowaniem, ani sekwencją. Typ `frozenset` to niezmienny wariant typu `set`.

- Poza operacjami kategorii typu w Pythonie 2.6 oraz 3.0 wszystkie typy z tabeli 9.3 mają wywoływalne metody, które są zazwyczaj specyficzne dla ich typu.

Tabela 9.3. Klasifikacja typów obiektów

Typ obiektu	Kategoria typu	Zmienny?
Liczby (wszystkie)	Liczbowy	Nie
łańcuchy znaków	Sekwencja	Nie
Listy	Sekwencja	Tak
Słowniki	Odwzorowanie	Tak
Krotki	Sekwencja	Nie
Pliki	Rozszerzenie	Nie dotyczy
Zbiory	Zbiór	Tak
frozenset	Zbiór	Nie
bytearray (Python 3.0)	Sekwencja	Tak

Znaczenie przeciążania operatorów

W szóstej części książki zobaczymy, że obiekty implementowane za pomocą klas mogą w dowolny sposób korzystać z tych kategorii. Jeśli na przykład chcemy udostępnić nowy rodzaj wyspecjalizowanego obiektu sekwencji, spójnego z wbudowanymi sekwencjami, wystarczy zakodować klasę przeciążającą na przykład indeksowanie i konkatenację.

```
class MySequence:
    def __getitem__(self, index):
        # Wywoływany na self[index], others
    def __add__(self, other):
        # Wywoływany na self + other
```

Nowy obiekt może również być zmienny lub niezmienny, w zależności od wyboru metod wywoływanych dla operacji zmieniających obiekt w miejscu (na przykład `__setitem__` jest wywoływanie na przypisaniach `self[index] = value`). Choć wykracza to poza tematykę niniejszej książki, można również implementować nowe obiekty w językach zewnętrznych, jak C — jako rozszerzenia języka C. W tym przypadku wypełnia się wskaźniki do funkcji C w celu wyboru spomiędzy zbioru operacji na liczbach, sekwencjach i odwzorowaniach.

Elastyczność obiektów

W tej części książki wprowadziliśmy kilka złożonych typów obiektów (kolekcji zawierających komponenty). Ogólnie rzecz biorąc:

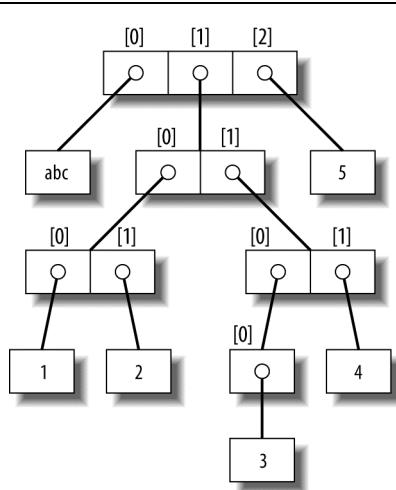
- listy, słowniki oraz krotki mogą przechowywać dowolne obiekty,
- listy, słowniki i krotki można dowolnie zagnieżdżać,
- listy i słowniki mogą rosnąć i kurczyć się w sposób dynamiczny.

Ponieważ złożone typy obiektów Pythona obsługują dowolne struktury, świetnie nadają się do reprezentowania skomplikowanych informacji w programach. Przykładowo wartości słownika

mogą być listami zawierającymi krotki, które z kolei zawierają słowniki — i tak dalej. Zagnieżdżanie może mieć dowolną głębokość niezbędną do przedstawienia danych, które mają być przetworzone.

Przyjrzyjmy się przykładowi zagnieżdżania. Poniższy listing definiuje drzewo zagnieżdżonych złożonych obiektów sekwencji, zaprezentowanych również na rysunku 9.1. By uzyskać dostęp do poszczególnych komponentów, można skorzystać z dowolnej liczby operacji indeksowania. Python oblicza indeksy od lewej do prawej strony i z każdym krokiem pobiera referencje do głębszej zagnieżdżonych obiektów. Rysunek 9.1 może przedstawiać patologicznie skomplikowaną strukturę danych, jednak dobrze ilustruje składnię wykorzystywaną w celu uzyskania dostępu do obiektów zagnieżdżonych.

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```



Rysunek 9.1. Drzewo zagnieżdżonych obiektów wraz z wartościami przesunięcia komponentów, utworzone dzięki wykonaniu wyrażenia literatu `['abc', [(1, 2), ([3], 4)], 5]`. Obiekty zagnieżdżone z punktu widzenia składni są wewnętrznie reprezentowane przez referencje (wskaźniki) do osobnych fragmentów pamięci

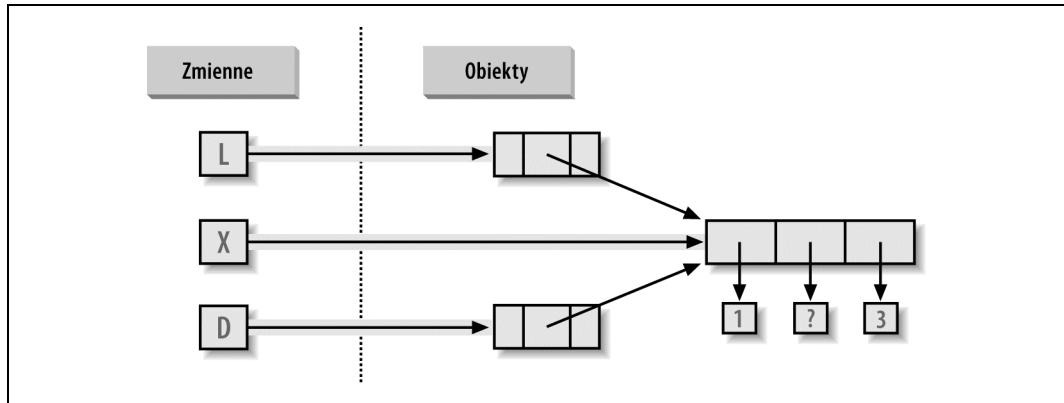
Referencje a kopie

W rozdziale 6. wspomnieliśmy, że przypisanie zawsze przechowuje referencje do obiektów, a nie ich kopie. W praktyce najczęściej o to nam chodzi. Ponieważ jednak przypisania mogą wygenerować wiele referencji do tego samego obiektu, należy być świadomym, że modyfikacja zmiennego obiektu w miejscu może wpłynąć na inne referencje do tego samego obiektu znajdujące się w programie. Jeśli nie życzymy sobie takiego zachowania, będziemy musieli nakazać Pythonowi utworzyć kopię obiektu.

Z tym zjawiskiem zapoznaliśmy się już w rozdziale 6., jednak staje się ono bardziej subtelne, kiedy w grę wchodzą większe obiekty. Poniższy przykład tworzy listę przypisaną do zmiennej `X` i kolejną listę przypisaną do zmiennej `L` z osadzoną referencją do listy `X`. Tworzy również słownik `D` zawierający kolejną referencję do listy `X`.

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']
>>> D = {'x':X, 'y':2} # Osadzone referencje do obiektu zmiennej X
```

W tym momencie istnieją trzy referencje do pierwszej utworzonej listy — ze zmiennej `X`, ze środka listy przypisanej do zmiennej `L` oraz ze środka słownika przypisanego do zmiennej `D`. Sytuacja ta została przedstawiona na rysunku 9.2.



Rysunek 9.2. Współdzielone referencje do obiektu. Ponieważ referencje do listy, do której odnosi się zmienne `X`, znajdują się również wewnętrz obiektów, do których odnoszą się zmienne `L` i `D`, zmiana współdzielonej listy z `X` sprawia, że lista ta zmieni się również w `L` i `D`

Ponieważ listy są zmienne, modyfikacja współdzielonego obiektu listy w dowolnej z trzech referencji zmieni ją również dla dwóch pozostałych referencji.

```
>>> X[1] = 'niespodzianka' # Zmienia wszystkie trzy referencje!
>>> L
['a', [1, 'niespodzianka', 3], 'b']
>>> D
{'x': [1, 'niespodzianka', 3], 'y': 2}
```

Referencje są wysokopoziomową analogią do wskaźników z innych języków programowania. Choć nie możemy uzyskać dostępu do samej referencji, możliwe jest przechowanie tej samej referencji w większej liczbie miejsc (zmiennych czy list). Umożliwia to na przykład przekazywanie większego obiektu w programie bez konieczności kosztownego generowania przy tym kopii. Jeśli jednak naprawdę zależy nam na kopii, możemy ich zażądać.

- Wyrażenia wycinków z pustymi granicami (`L[:]`) kopią sekwencje.
- Metoda słownika oraz zbioru `copy` (`D.copy()`) kopiuje słownik lub zbiór.
- Niektóre wbudowane funkcje, takie jak `list`, także wykonują kopie (`list(L)`).
- Moduł biblioteki standardowej `copy` sporządza pełne kopie.

Powiedzmy na przykład, że mamy listę oraz słownik i nie chcemy, by ich wartości zmieniały się za pośrednictwem innych zmiennych.

```
>>> L = [1, 2, 3]
>>> D = {'a':1, 'b':2}
```

By temu zapobiec, wystarczy przypisać do innych zmiennych kopie, a nie referencje do tych samych obiektów.

```
>>> A = L[:]
>>> B = D.copy()                                # Zamiast A = L (lub list(L))
                                                # Zamiast B = D (tak samo dla zbiorów)
```

W ten sposób zmiany wprowadzone w innych zmiennych będą modyfikowały kopie, a nie oryginalne obiekty.

```
>>> A[1] = 'Ni'
>>> B['c'] = 'mielonka'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'mielonka', 'b': 2})
```

W przypadku naszego początkowego przykładu możemy uniknąć efektów ubocznych referencji, sporządzając wycinek z oryginalnej listy zamiast podawania nazwy zmiennej.

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']                      # Osadzenie kopii obiektu zmiennej X
>>> D = {'x':X[:], 'y':2}
```

To zmienia obraz sytuacji z rysunku 9.2. Zmienne `L` i `D` będą teraz wskazywały na inne listy niż `X`. Rezultat będzie taki, że zmiany wprowadzone za pośrednictwem zmiennej `X` będą miały wpływ wyłącznie na `X`, a nie na `L` i `D`. W podobny sposób modyfikacje `L` i `D` nie wpłyną na `X`.

Ostatnia uwaga dotycząca kopii: wycinki z pustymi granicami i metoda słownika `copy` wykonyują jedynie kopie *najwyższego poziomu*. Oznacza to, że nie kopiują one zagnieżdżonych struktur danych, jeśli takie są obecne. Jeśli potrzebna nam jest pełna i całkowicie niezależna kopia głęboko zagnieżdżonej struktury danych, powinniśmy skorzystać z modułu `copy` z biblioteki standardowej. By skopiować dowolnie zagnieżdżony obiekt `Y`, należy umieścić w kodzie instrukcję `import copy` i wpisać `X = copy.deepcopy(Y)`. Wywołanie to przechodzi obiekt w sposób rekurencyjny w celu skopiowania wszystkich jego części. Jest to jednak o wiele rzadszy przypadek (i dlatego do wykonania go potrzeba więcej kodu). Zazwyczaj zależy nam na referencjach. Kiedy tak nie jest, wycinki i metoda `copy` zwykle będą wystarczające.

Porównania, równość i prawda

Wszystkie obiekty Pythona poddają się również porównaniom — testom równości czy względnego rozmiaru. Porównania w Pythonie zawsze sprawdzają wszystkie części obiektów złożonych, dopóki nie zostanie ustalony wynik. Tak naprawdę, kiedy obecne są obiekty zagnieżdżone, Python automatycznie przechodzi struktury danych w celu zastosowania porównań w sposób *rekurencyjny* od lewej do prawej strony i tak głęboko, jak jest to konieczne. Pierwsza napotkana różnica przesądza o wyniku porównania.

Dla przykładu porównanie obiektów list automatycznie porównuje wszystkie ich komponenty.

```
>>> L1 = [1, ('a', 3)]                         # Ta sama wartość, unikalne obiekty
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2                          # Odpowiednik? Ten sam obiekt?
(True, False)
```

W powyższym kodzie do zmiennych `L1` i `L2` przypisano listy będące odpowiednikami, jednak różnymi obiektami. Ze względu na naturę referencji Pythona (przedstawioną w rozdziale 6.) równość można sprawdzać na dwa sposoby.

- **Operator `==` sprawdza równość wartości.** Python wykonuje test równości, porównując rekurencyjnie wszystkie zagnieżdżone obiekty.
- **Operator `is` sprawdza identyczność obiektów.** Python sprawdza, czy dwa obiekty są tak naprawdę jednym (to znaczy znajdują się pod jednym adresem w pamięci).

W poprzednim przykładzie listy `L1` i `L2` zdają test `==` (mają takie same wartości, ponieważ wszystkie ich komponenty są takie same), natomiast nie zdają testu `is` (są referencjami do dwóch różnych obiektów i tym samym dwóch różnych fragmentów pamięci). Warto jednak zwrócić uwagę, co dzieje się w przypadku krótkich łańcuchów znaków.

```
>>> S1 = 'mielonka'  
>>> S2 = 'mielonka'  
>>> S1 == S2, S1 is S2  
(True, True)
```

Tutaj powinniśmy znowu mieć do czynienia z sytuacją, w której dwa odrębne obiekty mają tę samą wartość. Wynikiem testu `==` powinna być prawda, a testu `is` fałsz. Jednak ponieważ Python wewnętrznie umieszcza niektóre łańcuchy znaków w pamięci podręcznej i ponownie ich używa ze względu na optymalizację działania, tak naprawdę w pamięci istnieje tylko jeden łańcuch znaków: '`mielonka`', wspólnie dzielony przez `S1` i `S2`. Z tego powodu test identyczności `is` zwraca `True`. By wywołać normalne zachowanie, musimy wykorzystać dłuższy łańcuch znaków.

```
>>> S1 = 'dłuższy łańcuch znaków'  
>>> S2 = 'dłuższy łańcuch znaków'  
>>> S1 == S2, S1 is S2  
(True, False)
```

Oczywiście ponieważ łańcuchy znaków są niezmienne, mechanizm umieszczania obiektów w pamięci podręcznej nie ma wpływu na nasz kod. łańcuchów znaków nie można modyfikować w miejscu bez względu na to, ile zmiennych się do nich odnosi. Jeśli testy identyczności wydają się mylące, warto wrócić do rozdziału 6. w celu powtórzenia wiadomości dotyczącej referencji do obiektów.

Generalnie w prawie wszystkich testach równości wykorzystuje się operator `==`. Operator `is` zarezerwowany jest dla wysoce wyspecjalizowanych zadań. W dalszej części książki zobaczymy przykłady zastosowania tych operatorów.

Do zagnieżdżonych struktur danych rekurencyjnie stosują się również porównania względnego rozmiaru.

```
>>> L1 = [1, ('a', 3)]  
>>> L2 = [1, ('a', 2)]  
>>> L1 < L2, L1 == L2, L1 > L2  
(False, False, True) # Mniejszy, równy, większy — krotka wyników
```

W powyższym kodzie lista `L1` jest większa od `L2`, ponieważ zagnieżdżone 3 jest większe od 2. Wynik ostatniego wiersza kodu jest tak naprawdę krotką trzech obiektów — wyników trzech wpisanych wyrażeń (jest to przykład krotki bez nawiasów).

Python porównuje typy w następujący sposób:

- Liczby porównywane są zgodnie z ich względną wielkością.
- łańcuchy znaków porównywane są leksykograficznie, znak po znaku ("abc" < "ac").

- Listy i krotki porównywane są poprzez zestawienie każdego komponentu od lewej do prawej strony.
- Słowniki uznawane są za równe, jeśli ich posortowane listy (*klucz, wartość*) są równe. Porównania względnego rozmiaru nie są obsługiwane dla słowników w Pythonie 3.0, jednak działają w wersji 2.6 oraz wcześniejszych, tak jak porównywano posortowane listy (*klucz, wartość*).
- Porównania mieszanych typów nieliczbowych (na przykład `1 < 'mielonka'`) są w Pythonie 3.0 błędem. Są one dozwolone w Pythonie 2.6, jednak wykorzystują z góry określone reguły uporządkowania. W sposób pośredni odnosi się to również do sortowania, które wewnętrznie wykorzystuje porównania. Kolekcje nieliczbowych typów mieszanych nie mogą być w Pythonie 3.0 sortowane.

Porównania obiektów ustrukturyzowanych wyglądają tak, jakbyśmy obiekty te zapisali jako literaly i porównywali wszystkie ich elementy po jednym na raz od lewej do prawej strony. W kolejnych rozdziałach zobaczymy inne typy danych, które mogą zmienić sposób porównywania.

Porównywanie słowników w Pythonie 3.0

Przedostatni przykład z poprzedniego podrozdziału zasługuje na rozwinięcie. W Pythonie 2.6 oraz wersjach wcześniejszych słowniki obsługują porównania rozmiaru, tak jakbyśmy porównywali posortowane listy kluczów lub wartości.

```
C:\misc> c:\python26\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
True
```

W Pythonie 3.0 porównania rozmiaru dla słowników zostały usunięte, ponieważ w przypadku gdy pożądana jest równość, było to zbyt kosztowne (równość w wersji 3.0 wykorzystuje zoptymalizowane rozwiązanie, które nie porównuje w sposób dosłowny posortowanych list kluczów lub wartości). Alternatywą dla tej wersji Pythona jest albo pisanie pętli porównujących wartości po kluczu, albo ręczne porównanie posortowanych list kluczów lub wartości — wystarczą do tego metoda słownika `items` oraz funkcja wbudowana `sorted`.

```
C:\misc> c:\python30\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
TypeError: unorderable types: dict() < dict()

>>> list(D1.items())
[('a', 1), ('b', 2)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]

>>> sorted(D1.items()) < sorted(D2.items())
True
>>> sorted(D1.items()) > sorted(D2.items())
False
```

W praktyce większość programów wymagających tego działania wytwarza bardziej wydajne sposoby porównywania danych ze słowników zarówno w stosunku do powyższego rozwiązania, jak i oryginalnego działania porównań z Pythona 2.6.

Znaczenie True i False w Pythonie

Warto zwrócić uwagę na to, że wyniki testów zwarcane w dwóch ostatnich przykładach reprezentują wartości prawdy i fałszu. Wyświetlane są one jako słowa `True` i `False`, jednak skoro już na poważnie używamy testów logicznych, powinienem wyjaśnić, co tak naprawdę znaczą te nazwy z formalnego punktu widzenia.

W Pythonie, jak w większości języków programowania, liczba całkowita `0` reprezentuje fałsz, natomiast liczba całkowita `1` — prawdę. Dodatkowo jednak Python rozpoznaje dowolną pustą strukturę danych jako fałsz, a dowolną niepustą strukturę danych jako prawdę. Mówiąc bardziej ogólnie, koncepcje prawdy i fałszu są wrodzonymi właściwościami każdego obiektu w Pythonie. Każdy obiekt jest albo prawdą, albo fałszem, zgodnie z poniższymi regułami:

- liczby są prawdą, jeśli nie są zerem,
- pozostałe obiekty są prawdą, jeśli nie są puste.

W tabeli 9.4 zaprezentowano przykłady prawdy i fałszu na bazie obiektów Pythona.

Tabela 9.4. Przykłady prawdy i fałszu

Obiekt	Wartość
<code>"mielonka"</code>	<code>True</code>
<code>""</code>	<code>False</code>
<code>[]</code>	<code>False</code>
<code>{}</code>	<code>False</code>
<code>1</code>	<code>True</code>
<code>0.0</code>	<code>False</code>
<code>None</code>	<code>False</code>

W jednym z zastosowań, ponieważ same obiekty są albo prawdą, albo fałszem, programiści Pythona zwykle tworzą testy takie jak `if X:`, które, zakładając, że `X` jest łańcuchem znaków, odpowiadają kodowi `if X != ''`. Innymi słowy, można sprawdzać sam obiekt, zamiast porównywać go z obiektem pustym. Więcej informacji na temat instrukcji `if` znajdziesz się w trzeciej części książki.

Obiekt None

Jak widać w tabeli 9.4, Python udostępnia również specjalny obiekt `None`, który zawsze uznawany jest za fałsz. `None` został przedstawiony w rozdziale 4. Jest jedną wartością specjalnego typu danych w Pythonie i zazwyczaj służy za pusty pojemnik, podobnie jak wskaźnik `NULL` w języku C.

Można sobie na przykład przypomnieć, że w przypadku list nie możemy przypisać elementu do wartości przesunięcia, o ile ta wartość przesunięcia jeszcze nie istnieje (lista nie może magicznie rosnąć, jeśli wykonamy przypisanie poza jej granicami). By na początku zdefiniować listę studentową, do której będzie można dodać coś do dowolnej wartości przesunięcia, można ją wypełnić obiektami `None`.

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ... ]
```

Powyższy kod nie ogranicza rozmiaru listy (nadal może ona później rosnąć lub się kurczyć), a po prostu ustawia jej początkową wielkość, tak by pozwolić na przyszłe operacje przypisania do indeksów. W ten sam sposób możemy oczywiście także zainicjalizować listę z serią zer, jednak praktyka wskazuje na użycie `None`, jeśli zawartość listy nie jest jeszcze znana.

Należy pamiętać, że `None` nie oznacza „niezdefiniowany”. Inaczej mówiąc, `None` jest czymś, a nie niczym (mimo że nazwa tego obiektu wskazywałaby na coś innego!). `None` jest prawdziwym obiektem, zajmującym miejsce w pamięci; wbudowaną nazwę otrzymuje od Pythona. Inne zastosowania tego specjalnego obiektu zobaczymy w dalszej części książki. Jest to również domyślna wartość zwracana z funkcji, o czym przekonamy się w czwartej części książki.

Typ `bool`

Warto również pamiętać, iż typ Boolean Pythona (`bool`), wprowadzony w rozdziale 5., po prostu rozszerza koncepcje prawdy i fałszu w tym języku. Jak wiemy z rozdziału 5., wbudowane słowa `True` oraz `False` są po prostu inaczej zapisanymi wersjami liczb całkowitych 1 i 0 — wygląda to trochę tak, jakby w całym Pythonie te dwa słowa zostały z góry przypisane do liczb 1 i 0. Ze względu na sposób implementacji tego nowego typu jest to po prostu niewielkie rozszerzenie opisanych już pojęć prawdy i fałszu, wprowadzone w celu nadania tym wartościom lepiej rozpoznawalnego wyglądu.

- Kiedy w kodzie testów prawdy wykorzystamy te słowa w sposób jawnny, `True` i `False` są odpowiednikami liczb całkowitych 1 i 0, które jednak w jaśniejszy sposób przedstawiają intencje programisty.
- Również wyniki testów Boolean wykonywanych interaktywnie wyświetlane są jako słowa `True` i `False`, a nie liczby całkowite 1 i 0, by całość była bardziej zrozumiała.

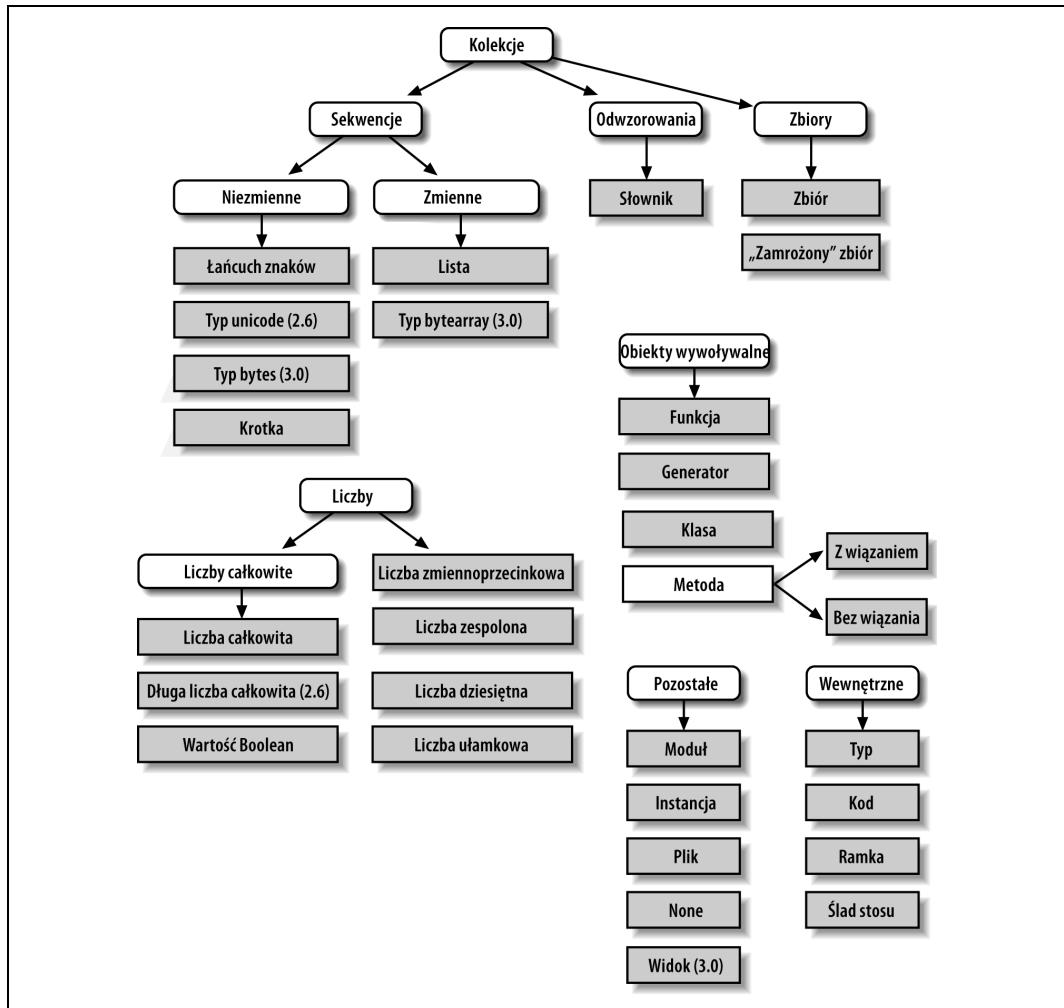
W instrukcjach logicznych, takich jak `if`, nie musimy korzystać jedynie z typów Boolean. Wszystkie obiekty mają wrodzoną wartość prawdy lub fałszu i wszystkie koncepcje z wartościami Boolean opisane w niniejszym rozdziale będą działały, nawet jeśli użyjemy innych typów. Python udostępnia funkcję wbudowaną `bool`, którą można wykorzystać do sprawdzenia wartości Boolean obiektu (to znaczy, czy ma on wartość `True` — a zatem jest niezerowy lub niepusty).

```
>>> bool(1)
True
>>> bool('mielonka')
True
>>> bool({})
False
```

W praktyce rzadko zobaczymy typ Boolean zwracany przez testy logiki, ponieważ są one automatycznie wykorzystywane przez instrukcje `if` oraz inne narzędzia wyboru. Wartości Boolean będziemy jeszcze omawiać przy okazji rozważań nad instrukcjami logicznymi w rozdziale 12.

Hierarchie typów Pythona

Na rysunku 9.3 podsumowano wszystkie wbudowane typy obiektów dostępne w Pythonie wraz z ich wzajemnymi relacjami. Przyjrzelismy się najbardziej znanym. Większość pozostałych typów obiektów z rysunku 9.3 odpowiada częścioom programu (jak funkcje i moduły) lub udostępnionym częścioom wewnętrznym interpretera (jak ramki stosu czy skompilowany kod).



Rysunek 9.3. Najważniejsze wbudowane typy obiektów Pythona podzielone na kategorie. W Pythonie wszystko jest typem obiektu, nawet sam typ obiektu!

Najważniejsze, o czym należy wspomnieć, to to, że w systemie Pythona *wszystko* jest typem obiektu; wszystko można też przetworzyć w programie napisanym w tym języku. Można na przykład klasę przekazać do funkcji, przypisać ją do zmiennej czy umieścić w liście bądź słowniku.

Obiekty typów

Tak naprawdę nawet same typy są w Pythonie typami obiektów — typem obiektu jest obiekt typu `type` (a teraz proponuję trzy razy szybko przeczytać na głos to stwierdzenie). A mówiąc poważnie, wywołanie wbudowanej funkcji `type(x)` zwraca typ obiektu `x`. W praktyce obiekty typów można wykorzystać do ręcznych porównań typów w instrukcjach `if` Pythona. Ze względu na powody wymienione w rozdziale 4. w Pythonie nie wykonuje się raczej ręcznego sprawdzania typów, gdyż ogranicza to elastyczność kodu.

Jedna uwaga dotycząca nazw typów: od Pythona 2.2 każdy typ podstawowy ma nową wbudowaną nazwę, dodaną w celu obsługi dostosowywania typów do własnych potrzeb za pomocą zorientowanych obiektowo podklas. Te nazwy to: `dict` (słownik), `list` (lista), `str` (łańcuch znaków), `tuple` (krotka), `int` (liczba całkowita), `float` (liczba zmiennoprzecinkowa), `complex` (liczba zespolona), `bytes` (łańcuch bajtowy), `type` (typ), `set` (zbiór), oraz inne (w Pythonie 2.6, ale już nie w wersji 3.0, `file` — plik — jest nazwą typu oraz synonimem `open`). Wywołania tych nazw są tak naprawdę wywołaniami konstruktora obiektu, a nie tylko funkcjami konwersji, choć można je traktować jako proste funkcje służące do podstawowych zadań.

Ponadto moduł biblioteki standardowej Pythona 3.0 `types` udostępnia dodatkowe nazwy typów przeznaczone dla typów niedostępnych jako nazwy wbudowane (na przykład typu funkcji). W Pythonie 2.6, ale nie w wersji 3.0, moduł ten zawiera również synonimy dla wbudowanych nazw typów. Sprawdzanie typów można również wykonywać za pomocą funkcji `isinstance`. Na przykład wszystkie z poniższych testów są prawdziwe.

```
type([1]) == type([])           # Typ innej listy
type([1]) == list                # Nazwa typu listy
isinstance([1], list)           # Lista lub jej dostosowanie do własnych potrzeb

import types                     # Moduł types zawiera nazwy dla innych typów
def f(): pass
type(f) == types.FunctionType
```

Ponieważ typy mogą obecnie być wykorzystane przy tworzeniu klas podzielonych, technika `isinstance` jest szeroko polecana. Więcej informacji na temat tworzenia podklas typów wbudowanych w Pythonie 2.2 i późniejszych wersjach można znaleźć w rozdziale 31.

W rozdziale 31. sprawdzimy także, w jaki sposób `type(X)` oraz ogólnie sprawdzanie typów odnoszą się do instancji *klas* zdefiniowanych przez użytkownika. Mówiąc krótko, w Pythonie 3.0 oraz klasach w nowym stylu w Pythonie 2.6 typem instancji klasy jest klasa, z której utworzona została instancja. W przypadku klas klasycznych z Pythona 2.6 i wersji wcześniejszych wszystkie instancje klas są typem „instancja” i w celu porównania ich typów musimy porównać atrybuty `__class__` instancji. Ponieważ jednak nie jesteśmy jeszcze gotowi na klasy, odłożymy omówienie tego zagadnienia do rozdziału 31.

Inne typy w Pythonie

Poza typami podstawowymi omawianymi w tej części książki oraz obiektami jednostek programów, takimi jak funkcje, moduły oraz klasy, z którymi spotkamy się nieco później, typowa instalacja Pythona zawiera dziesiątki innych typów obiektów dostępnych jako dołączone rozszerzenia języka C lub klasy Pythona — obiekty wyrażeń regularnych, pliki DBM, widgety GUI czy gniazda sieciowe.

Podstawowa różnica między tymi dodatkowymi narzędziami a przedstawionymi dotychczas typami wbudowanymi polega na tym, że typy wbudowane udostępniają specjalną składnię tworzącą język dla ich obiektów (na przykład `4` dla liczb całkowitej, `[1, 2]` dla listy, funkcję `open` dla plików, a także `def` oraz `lambda` dla funkcji). Inne narzędzia są zazwyczaj udostępniane przez moduły biblioteki standardowej, które przed użyciem trzeba zaimportować. By na przykład utworzyć obiekt wyrażenia regularnego, należy zaimportować moduł `re` i wywołać metodę `re.compile()`. Wyczerpujący przewodnik po wszystkich narzędziach dostępnych dla programów napisanych w Pythonie można znaleźć w dokumentacji biblioteki standardowej tego języka.

Pułapki typów wbudowanych

To koniec naszego omówienia podstawowych typów danych. Tę część książki zamknimy przedstawieniem często spotykanych problemów, z jakimi stykają się nowi użytkownicy (a czasami nawet eksperci), wraz z ich rozwiązaniami. Część będzie powtórzeniem kwestii, z którymi już się spotkaliśmy, jednak są one na tyle ważne, że zasługują na ponowne przypomnienie.

Przypisanie tworzy referencje, nie kopie

Ponieważ jest to kluczowa koncepcja, powtórzę to raz jeszcze. Zrozumienie tego, co dzieje się ze współdzielonymi referencjami w programie, jest kwestią podstawową. W poniższym kodzie obiekt listy przypisany jest do zmiennej `L`. Referencja do niego znajduje się w `L` oraz w środku listy przypisanej do zmiennej `M`. Modyfikacja w miejscu zmienia także to, do czego odnosi się `M`.

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']                                # Osadzenie referencji do L
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0                                         # Zmiany również w M
>>> M
['X', [1, 0, 3], 'Y']
```

Ten efekt nabiera znaczenia w większych programach i często współdzielone referencje są dokładnie tym, czego chcemy. Jeśli tak nie jest, można uniknąć współdzielania obiektów dzięki kopowaniu ich w jawnny sposób. W przypadku list zawsze można wykonać kopię najwyższego poziomu, używając wycinka z pustymi granicami.

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']                            # Osadzenie kopii L
>>> L[1] = 0                                         # Zmiana tylko L, nie M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Należy pamiętać, że granice wycinka mają wartości domyślne `0` i długość ciętej sekwencji. Jeśli pominiemy obie, wycinek dokonuje ekstrakcji każdego elementu sekwencji, tworząc jednocześnie kopię najwyższego poziomu (nowy, niewspółdzielony obiekt).

Powtórzenie dodaje jeden poziom zagębszenia

Powtórzenia w sekwencjach przypominają dodanie sekwencji do samej siebie jakąś liczbę razy. Kiedy jednak zagnieżdżone zostają zmienne sekwencje, efekt może nie zawsze być tym, czego oczekujemy. W poniższym przykładzie zmienna `X` zostaje przypisana do listy `L` powtarzonej cztery razy, natomiast zmienna `Y` przypisana jest do listy *zawierającej* `L` powtarzone cztery razy.

```
>>> L = [4, 5, 6]
>>> X = L * 4
>>> Y = [L] * 4
                                         # Jak [4, 5, 6] + [4, 5, 6] + ...
                                         # [L] + [L] + ... = [L, L, ...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Ponieważ w drugim powtórzeniu lista `L` została zagnieżdżona, `Y` składa się z osadzonych referencji z powrotem do oryginalnej listy przypisanej do `L`, dlatego jest podatne na różne efekty uboczne odnotowane w poprzednim podrozdziale.

```
>>> L[1] = 0
                                         # Ma wpływ na Y, ale nie na X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

Do sytuacji tej odnoszą się te same rozwiązania co wymienione w poprzednim podrozdziale, ponieważ jest to kolejny przypadek utworzenia współdzielonej referencji do zmiennego obiektu. Jeśli przypomniemy sobie, że powtórzenie, konkatenacja i wycinek kopiąją jedynie najwyższy poziom obiektów będących argumentami, takie przypadki zaczynają nabierać sensu.

Uwaga na cykliczne struktury danych

Z tą koncepcją spotkaliśmy się w poprzednim ćwiczeniu. Jeśli kolekcja obiektów zawiera referencję do samej siebie, nazywana jest *obiektem cyklicznym*. Python wyświetla [...] za każdym razem, gdy wykryje cykl w obiekcie, zamiast zacinać się w nieskończonej pętli.

```
>>> L = ['Graal']
>>> L.append(L)
                                         # Dodanie referencji do tego samego obiektu
                                         # Utworzenie cyklu w obiekcie: [...]
>>> L
['Graal', [...]]
```

Poza zrozumieniem, że trzy kropki w nawiasach kwadratowych oznaczają cykl w obiekcie, warto również zapoznać się z tym przypadkiem, ponieważ może on prowadzić do różnych pułapek. Struktury cykliczne mogą spowodować wpadnięcie kodu w nieskończone pętle, jeśli nie będziemy na to przygotowani. Niektóre programy przechowują na przykład listę czy słownik już odwiedzonych elementów i sprawdzają je w celu przekonania się, że nie są w cyklu. Więcej informacji na temat tego problemu można znaleźć w rozwiązaniach do ćwiczeń podsumowujących pierwszą część książki, zamieszczonych w dodatku B. Rozwiązanie tego problemu można znaleźć w programie `reloadall.py` na końcu rozdziału 24.

O ile naprawdę nie musimy tego robić, nie należy korzystać z referencji cyklicznych. Istnieją ważne przyczyny stosowania cykli, jednak jeśli nie mamy kodu, który wie, jak sobie z nimi radzić, lepiej będzie nie tworzyć zbyt często w obiektach referencji do nich samych.

Typów niezmiennych nie można modyfikować w miejscu

Nie można modyfikować niezmiennej wartością nowego obiektu w miejscu. Zamiast tego należy skonstruować nowy obiekt za pomocą wycinka, konkatenacji czy podobnych operacji i przypisać go z powrotem do pierwotnej referencji, o ile jest to konieczne.

```
T = (1, 2, 3)  
T[2] = 4 # Błąd!  
T = T[:2] + (4,) # OK: (1, 2, 4)
```

Może się to wydawać niepotrzebnym kodowaniem, jednak z drugiej strony, kiedy korzystamy z obiektów niezmiennych, jak łańcuchy znaków i krotki, nie przytrafią się nam poprzednio opisane pułapki. Ponieważ obiekty tych nie można zmodyfikować w miejscu, nie są one podatne na te same efekty uboczne co listy.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy dwa ostatnie istotne podstawowe typy obiektów — krotkę oraz plik. Dowiedzieliśmy się, że krotki obsługują wszystkie operacje na sekwencjach, mają jedynie kilka własnych metod i nie pozwalają na modyfikację w miejscu, ponieważ są niezmienne. Nauczyliśmy się również, że pliki zwieracane są przez wbudowaną funkcję `open` i udostępniają metody służące do odczytu oraz zapisu danych. Sprawdziliśmy, jak można przekształcić obiekty Pythona na łańcuchy znaków służące do przechowywania danych w plikach (i odwrotnie). Przyjrzaliśmy się również modułom `pickle` i `struct` spełniającym bardziej zaawansowane role (serializacja obiektów oraz dane binarne). Wreszcie zakończyliśmy rozdział, przeglądając pewne właściwości wspólne dla wszystkich typów obiektów (na przykład współdzielone referencje) i przejrzaliśmy listę często popełnianych błędów (pułapek) w dziedzinie typów obiektów.

W kolejnej części książki zmienimy tematykę, przechodząc do zagadnienia składni instrukcji w Pythonie. W następnych rozdziałach omówimy wszystkie podstawowe instrukcje proceduralne tego języka. Kolejny rozdział rozpoczyna tę nową część od wprowadzenia do ogólnego modelu składni Pythona, który można zastosować do wszystkich typów instrukcji. Przed przejściem dalej warto jednak wykonać quiz podsumowujący rozdział, a następnie przejść przez serię ćwiczeń kończących tę część książki w celu powtórzenia najważniejszych koncepcji. Instrukcje najczęściej tworzą i przetwarzają obiekty, dlatego przed przejściem do ich omawiania należy się upewnić, że opanowaliśmy zagadnienia z dziedziny obiektów Pythona.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób można ustalić wielkość krotki? Dlaczego narzędzie to znajduje się tam, gdzie się znajduje?
2. Należy napisać wyrażenie zmieniające pierwszy element krotki. Krotka `(4, 5, 6)` powinna w rezultacie stać się krotką `(1, 5, 6)`.
3. Jaka jest domyślna wartość argumentu `z` trybarem przetwarzania w wywołaniu `open` dla pliku?

4. Jaki moduł można wykorzystać do przechowania obiektów Pythona w pliku bez ręcznego konwertowania ich dołańcuchów znaków?
5. W jaki sposób można skopiować wszystkie elementy zagnieżdżonej struktury za jednym razem?
6. Kiedy Python uznaje obiekt za prawdę?
7. Co jest naszym celem?

Sprawdź swoją wiedzę — odpowiedzi

1. Wbudowana funkcja `len` zwraca długość (liczbę elementów) dowolnego obiektu pojedynika w Pythonie, w tym krotek. Jest to funkcja wbudowana, a nie metoda typu, ponieważ ma zastosowanie do wielu różnych typów obiektów. Ogólnie rzecz biorąc, funkcje wbudowane oraz wyrażenia mogą rozciągać się na wiele różnych typów obiektów; metody są specyficzne dla jednego typu, jednak część z nich może być dostępna dla większej ich liczby (jak na przykład `index`, działająca na listach oraz krotkach).
2. Ponieważ krotki są niezmienne, nie można ich modyfikować w miejscu. Można jednak wygenerować nową krotkę o pożądanej wielkości. Mając krotkę `T = (4, 5, 6)`, możemy zmienić jej pierwszy element, tworząc nową krotkę z jej częścią za pomocą wycinka i konkatenacji — `T = (1,) + T[1:]`. Warto przypomnieć, że krotki jednoelementowe wymagają końcowego przecinka. Można również przekonwertować krotkę na listę, zmodyfikować ją w miejscu i przekształcić z powrotem na krotkę. Taka operacja jest jednak bardziej kosztowna i w praktyce rzadko stosowana. Jeśli wiemy, że obiekt będzie wymagał modyfikacji w miejscu, należy od początku zastosować listę.
3. Wartością domyślną argumentu z trybem przetwarzania jest '`r`' — od odczytu (ang. *read*). By otworzyć plik tekstowy do odczytu, wystarczy przekazać nazwę pliku zewnętrznego.
4. Do przechowania obiektów Pythona w pliku bez jawnego konwertowania ich nałańcuchy znaków można wykorzystać moduł `pickle`. Podobnym do niego modułem jest `struct`, który jednak zakłada, że dane muszą być w pliku spakowane w format binarny.
5. Jeśli chcemy skopiować wszystkie części zagnieżdżonej struktury `X`, należy zaimportować moduł `copy` i wywołać `copy.deepcopy(X)`. Takie coś rzadko spotyka się w praktyce. Najczęściej chodzi nam o referencje i w większości przypadków płytkie kopie (na przykład `aList[:], aDict.copy()`) w zupełności wystarczą.
6. Obiekt uznawany jest za prawdę, kiedy jest albo liczbą inną od zera, albo niepustym obiektem kolekcji. Wbudowane słowa `True` i `False` są zdefiniowane tak, by miały to samo znaczenie, co liczby całkowite `1` i `0`.
7. Akceptowane odpowiedzi to: „Nauczenie się Pythona”, „Przejście do kolejnej części książki” albo „Odnalezienie Świętego Graala”.

Sprawdź swoją wiedzę — ćwiczenia do części drugiej

W tej sesji będziemy zajmować się podstawami obiektów wbudowanych. Tak jak wcześniej, po drodze mogą pojawić się nowe koncepcje, dlatego należy koniecznie po wykonaniu ćwiczeń (lub w trakcie, jeśli sprawiają one problemy) przejrzeć odpowiedzi z dodatku B. Osobom

mającym mało czasu proponuję zacząć od ćwiczeń 10. i 11. (najbardziej praktycznych ze wszystkich), a później kontynuację od 1. w miarę znalezienia wolnej chwili. Całość materiału to podstawa, dlatego najkorzystniej będzie wykonać tak dużo ćwiczeń, jak to możliwe.

1. *Podstawy*. Należy poeksperymentować z najczęściej używanymi operacjami znajdującymi się w różnych tabelach tej części książki. Na początek należy uruchomić interpreter interaktywny Pythona, wpisać każde z poniższych wyrażeń i spróbować wyjaśnić, co się dzieje w każdym z tych przypadków. Warto zwrócić uwagę na to, że średnik w części przykładów został użyty do oddzielenia od siebie instrukcji, tak by udało się zmieścić kilka instrukcji w jednym wierszu. Przykładowo kod `X=1; X` przypisuje, a następnie wyświetla zmienną (więcej informacji na temat składni instrukcji znajdzie się w kolejnej części książki). Należy także pamiętać, że przecinek pomiędzy wyrażeniami zazwyczaj tworzy krotkę, nawet jeśli nie zastosowano nawiasów. `X, Y, Z` to krotka trójelementowa, którą Python wyświetla z użyciem nawiasów.

```
2 ** 16
2 / 5, 2 / 5.0

"mielonka" + "jajka"
S = "szynka"
"jajka " + S
S * 5
S[:0]
"zielone %s i %s" % ("jajka", S)
'zielone {0} i {1}'.format('jajka', S)

('x', )[0]
('x', 'y')[1]

L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)

{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
list(D.keys()), list(D.values()), (1,2,3) in D

[], [ "",[],(),{},None]
```

2. *Indeksowanie i wycinki*. W sesji interaktywnej należy zdefiniować listę o nazwie `L`, zawierającą cztery łańcuchy znaków lub liczby (na przykład `L = [0,1,2,3]`). Następnie należy wykonać kilka eksperymentów z przypadkami granicznymi. Być może nigdy nie zobaczymy takich przypadków w prawdziwych programach, ale służą one do zastanowienia się nad modelem leżącym u ich podstaw; część z nich może się przydać w mniej sztucznych formach:

- Co się stanie, kiedy spróbujemy wykorzystać indeks znajdujący się poza długością listy (jak `L[4]`)?
- Co się stanie, kiedy spróbujemy wykonać wycinek wykraczający poza długość listy (jak `L[-1000:100]`)?

- c) Jak radzi sobie Python, kiedy próbujemy dokonać ekstrakcji sekwencji w odwrotnej kolejności — gdy niższa granica jest większa od wyższej (jak `L[3:1]`)? (Wskazówka: warto spróbować przypisać coś do tego wycinka (na przykład `L[3:1] = ['?']`) i zobaczyć, gdzie zostanie wstawiona wartość. Czy jest to to samo zjawisko co przy próbie sporządzenia wycinka poza granicami listy?).
3. *Indeksowanie, wycinki i del.* Należy zdefiniować kolejną listę czteroelementową i przypisać pustą listę do jednej z jej wartości przesunięcia (na przykład `L[2] = []`). Co się stanie? Następnie należy przypisać pustą listę do wycinka (`L[2:3] = []`). Co stanie się teraz? Warto przypomnieć, że przypisanie do wycinka usuwa wycinek i wstawia nową wartość w miejscu, gdzie się on znajdował.

Instrukcja `del` służy do usuwania elementów o określonej wartości przesunięcia, a także kluczy, atrybutów oraz zmiennych. Można jej użyć na liście w celu usunięcia jakiegoś jej elementu (jak w `del L[0]`). Co się stanie, kiedy usuniemy cały wycinek (`del L[1:]`)? Co się stanie, kiedy przypiszemy do wycinka element niebędący sekwencją (na przykład `L[1:2] = 1`)?

4. *Przypisywanie krotek.* Należy wpisać do sesji interaktywnej następujące wiersze kodu:

```
>>> X = 'mielonka'  
>>> Y = 'jajka'  
>>> X, Y = Y, X
```

Co stanie się z `X` i `Y` po wpisaniu tej sekwencji?

5. *Klucze słowników.* Rozważmy poniższy fragment kodu:

```
>>> D = {}  
>>> D[1] = 'a'  
>>> D[2] = 'b'
```

Wiemy już, że dostęp do elementów słownika nie odbywa się po wartościach przesunięcia — co się zatem tutaj dzieje? Czy poniższy kod trochę to wyjaśnia? (Wskazówka: do jakiej kategorii typów należą łańcuchy znaków, liczby całkowite oraz krotki?)

```
>>> D[(1, 2, 3)] = 'c'  
>>> D  
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Indeksowanie słowników.* Należy utworzyć słownik `D` z trzema wpisami dla kluczy '`a`', '`b`' oraz '`c`'. Co się stanie, kiedy spróbujemy zindeksować słownik dla nieistniejącego klucza (`D['d']`)? Co zrobi Python, kiedy spróbujemy przypisać coś do nieistniejącego klucza '`d`' (na przykład `D['d'] = 'mielonka'`)? Jak można to porównać z przypisaniami i referencjami poza granicami listy? Czy wygląda to na regułę dotyczącą nazw zmiennych?

7. *Operacje uniwersalne.* Należy wykorzystać sesję interaktywną do uzyskania odpowiedzi na poniższe pytania.

- Co się stanie, kiedy spróbujemy wykorzystać operator `+` na różnych (mieszanych) typach danych (na przykład łańcuchu znaków i liście, liście i krotce)?
- Czy `+` działa, kiedy jeden z argumentów jest słownikiem?
- Czy metoda `append` działa zarówno dla list, jak i łańcuchów znaków? Czy można użyć metody `keys` na listach? Wskazówka: co metoda `append` zakłada na temat obiektu będącego jej podmiotem?
- Wreszcie, jaki typ obiektu otrzymamy, kiedy sporządzimy wycinek lub wykonamy konkatencję dwóch list lub dwóch łańcuchów znaków?

8. *Indeksowanie łańcuchów znaków.* Należy zdefiniować łańcuch znaków S zawierający cztery znaki: `S = "jajo"`. Później należy wpisać wyrażenie `S[0][0][0][0]`. Jakiś pomysł, co może się stać? Wskazówka: warto przypomnieć, że łańcuch znaków jest kolekcją znaków, a znaki Pythona to jednoznakowe łańcuchy. Czy takie wyrażenie indeksujące nadal będzie działać, kiedy zastosujemy je do listy, takiej jak `['j', 'a', 'j', 'o']`? Dlaczego?
9. *Typy niezmienne.* Należy ponownie zdefiniować łańcuch znaków S zawierający cztery znaki: `S = "jajo"`. Należy napisać instrukcję przypisania zmieniającą ten łańcuch znaków na `"jaja"`, używając do tego wyłącznie wycinka oraz konkatenacji. Czy tę samą operację można wykonać z użyciem samego indeksowania oraz konkatenacji? A przypisania do indeksu?
10. *Zagnieżdżanie.* Należy utworzyć strukturę danych reprezentującą nasze dane osobowe — imiona i nazwisko, wiek, zawód, adres, adres e-mail i numer telefonu. Strukturę tą można zbudować z użyciem dowolnej kombinacji wbudowanych typów obiektów (list, krotek, słowników, łańcuchów znaków, liczb). Następnie należy uzyskać dostęp do poszczególnych komponentów struktury danych za pomocą indeksowania. Czy pewne struktury sprawdzają się w tym obiekcie lepiej od innych?
11. *Pliki.* Należy napisać skrypt tworzący nowy plik wyjściowy o nazwie `myfile.txt` i zapisujący do niego łańcuch znaków `"Witaj, wspaniały świecie!"`. Później należy napisać kolejny skrypt otwierający plik `myfile.txt`, odczytujący i wyświetlający jego zawartość. Oba skrypty należy wykonać z systemowego wiersza poleceń. Czy nowy plik pojawia się w katalogu, w którym wykonaliśmy skrypty? Co się stanie, jeśli do nazwy pliku przekazanej do metody `open` dodamy inną ścieżkę do katalogu? Uwaga: metody plików `write` nie dodają do łańcuchów znaków nowego wiersza. Jeśli chcemy w pełni zakończyć wiersz w pliku, musimy dodać `\n` na końcu łańcucha znaków.

CZĘŚĆ III

Instrukcje i składnia

Wprowadzenie do instrukcji Pythona

Ponieważ znamy już podstawowe wbudowane typy obiektów Pythona, niniejszy rozdział rozpoczniemy od omówienia podstawowych form instrukcji tego języka. Tak jak w poprzedniej części, zaczniemy od ogólnego wprowadzenia do składni instrukcji. W kolejnych rozdziałach znajdą się bardziej szczegółowe informacje dotyczące poszczególnych instrukcji.

Mówiąc ogólnie, *instrukcje* (ang. *statement*) to coś, co piszemy w celu przekazania Pythonowi tego, co mają robić nasze programy. Jeśli program „coś robi”, to instrukcje pozwalają określić, co to konkretnie jest. Python jest językiem proceduralnym, opartym na instrukcjach. Łącząc instrukcje, określamy procedurę wykonywaną przez Pythona w celu spełnienia celów programu.

Raz jeszcze o strukturze programu Pythona

Innym sposobem zrozumienia roli instrukcji jest powrócenie do hierarchii wprowadzonej w rozdziale 4., który omawiał obiekty wbudowane wraz z wyrażeniami służącymi do ich przetwarzania. Niniejszy rozdział stanowi przejście o jeden poziom w górę hierarchii.

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

Składnia Pythona składa się z instrukcji i wyrażeń. Wyrażenia przetwarzają obiekty i są osadzone w instrukcjach. Instrukcje kodują większą logikę operacji programu — wykorzystują i kierują wyrażenia do przetwarzania obiektów omawianych w poprzednich rozdziałach. Ponadto to właśnie w instrukcjach obiekty zaczynają istnieć (na przykład w wyrażeniach wewnętrz instrukcji przypisania), a niektóre instrukcje tworzą zupełnie nowe rodzaje obiektów (na przykład funkcje i klasy). Instrukcje zawsze istnieją w modułach, które z kolei same są zarządzane za pomocą instrukcji.

Instrukcje Pythona

W tabeli 10.1 zaprezentowano zbiór instrukcji Pythona. Niniejsza część książki omawia wpisy z tabeli od góry aż do `break` i `continue`. Niektóre z instrukcji z tej tabeli zostały już nieformalnie wprowadzone wcześniej. W tej części książki uzupełnimy pominięte szczegóły, wprowadzimy

Tabela 10.1. Instrukcje Pythona 3.0

Instrukcja	Rola	Przykład
Przypisanie	Tworzenie referencji	a, *b = 'dobry', 'zły', 'paskudny'
Wywołania	Wykonywanie funkcji	log.write("mielonka, szynka")
Wywołania print	Wyświetlanie obiektów	print('The Killer', joke)
if/elif/else	Wybór działania	if "python" in text: print(text)
for/else	Iteracja po sekwencjach	for x in mylist: print(x)
while/else	Ogólne pętle	while X > Y: print('witaj')
pass	Pusty pojemnik	while True: pass
break	Wyjście z pętli	while True: if exittest(): break
continue	Kontynuacja pętli	while True: if skiptest(): continue
def	Funkcje i metody	def f(a, b, c=1, *d): print(a+b+c+d[0])
return	Wynik funkcji	def f(a, b, c=1, *d): return a+b+c+d[0]
yield	Funkcje generatora	def gen(n): for i in n: yield i*2
global	Przestrzenie nazw	x = 'stary' def function(): global x, y; x = 'nowy'
nonlocal	Przestrzenie nazw (3.0+)	def outer(): x = 'stary' def function(): nonlocal x; x = 'nowy'
import	Dostęp do modułów	import sys
from	Dostęp do atrybutów	from sys import stdin
class	Budowanie obiektów	class Subclass(Superclass): staticData = [] def method(self): pass
try/except/finally	Przechwytywanie wyjątków	try: action() except: print('Błąd w akcji') raise endSearch(location)
raise	Wywoływanie wyjątków	assert X > Y, 'X jest za małe'
assert	Sprawdzanie w debugowaniu	with open('data') as myfile: process(myfile)
with/as	Menedżery kontekstu (2.6+)	del dane[k] del dane[i:j] del obiekt.atrybut del zmienna
del	Usuwanie referencji	

pozostała część zbioru instrukcji proceduralnych Pythona i omówimy ogólny model składni. Instrukcje z dolnej części tabeli 10.1, dotyczące większych części programów — funkcji, klas, modułów oraz wyjątków — prowadzą do zadań programistycznych, dlatego zasługują na

poświęcenie im osobnych części. Instrukcje bardziej wyspecjalizowane (jak `del`, usuwająca różne komponenty) omówione są w dalszej części książki lub w dokumentacji biblioteki standardowej Pythona.

Tabela 10.1 odzwierciedla instrukcje w postaci z Pythona 3.0 — fragmenty kodu o określonej składni i celu. Poniżej kilka uwag na temat zawartości tabeli:

- Instrukcje przypisania mogą mieć różne formy składni, opisane w rozdziale 11. — prostą, sekwencję, rozszerzoną i inną.
- `print` w wersji 3.0 nie jest ani słowem zarezerwowanym, ani instrukcją, a wywołaniem funkcji wbudowanej. Ponieważ jednak prawie zawsze wykonywane jest w postaci instrukcji wyrażenia (czyli w oddzielnym wierszu), zazwyczaj traktuje się je jako typ instrukcji. Operacje `print` zostaną omówione w kolejnym rozdziale.
- `yield` także jest tak naprawdę wyrażeniem, a nie instrukcją, w wersji 2.5. Tak jak `print`, najczęściej wykorzystywane jest w oddzielnym wierszu, dlatego uwzględnione zostało w tej tabeli, jednak jak zobaczymy w rozdziale 20., w skryptach wynik tej instrukcji jest czasami przypisywany lub wykorzystywany w inny sposób. Jako wyrażenie `yield` jest także słowem zarezerwowanym — odwrotnie niż `print`.

Większość powyższej tabeli ma także zastosowanie do Pythona 2.6 — z pewnymi wyjątkami. Osoby korzystające z wersji 2.6 lub starszej mogą zechcieć odnotować poniższe uwagi:

- W wersji 2.6 instrukcja `nonlocal` nie jest dostępna. Jak zobaczymy w rozdziale 17., istnieją alternatywne sposoby pozwalające uzyskać efekt stanu możliwości zapisu udostępnianego przez tę instrukcję.
- W wersji 2.6 `print` jest instrukcją, a nie wywołaniem wbudowanej funkcji, ze składnią omówioną w rozdziale 11.
- W wersji 2.6 wbudowana funkcja wykonywania kodu `exec` jest instrukcją o zdefiniowanej składni. Ponieważ jednak obsługuje ona nawiasy, można używać wywołania z wersji 3.0 także w Pythonie 2.6.
- W wersji 2.5 instrukcje `try/except` oraz `try/finally` zostały połączone. Formalnie były one oddzielnymi instrukcjami, jednak teraz możliwe jest użycie zarówno `except`, jak i `finally` w tej samej instrukcji `try`.
- W wersji 2.5 `with/as` jest opcjonalnym rozszerzeniem i nie jest dostępne, jeśli nie włączymy go w jawnym sposobie, wykonując instrukcję `from __future__ import with_statement` (więcej informacji na ten temat znajduje się rozdziale 33.).

Historia dwóch if

Zanim zagłębimy się w szczegóły którejś z instrukcji z tabeli 10.1, chciałbym zacząć nasze omawianie składni instrukcji od pokazania, czego *nie* będziemy wpisywać do kodu Pythona, tak by można było dokonać porównania tego języka z innymi modelami składni, z jakimi można się spotkać.

Rozważmy poniższą instrukcję `if` zakodowaną w języku podobnym do C.

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

Może to być instrukcja w języku C, C++, Java, JavaScript lub Perl. Teraz przyjrzyjmy się odpowiadającej jej instrukcji z Pythona.

```
if x > y:  
    x = 1  
    y = 2
```

Pierwszą rzeczą, jaką łatwo zauważyc, jest to, że instrukcja napisana w Pythonie jest mniej, nawiązmy to, zaśmiecona — to znaczy jest w niej mniej elementów składniowych. Jest to celowe — Python jest językiem skryptowym, zatem jednym z jego celów jest ułatwienie życia programistom poprzez pisanie mniejszej ilości kodu.

Co więcej, kiedy porównamy oba modele składni, okaże się, że Python dodaje jeden dodatkowy element, a trzy elementy obecne w językach podobnych do C w Pythonie są nieobecne.

Co dodaje Python

Tym jednym dodatkowym elementem składniowym jest w Pythonie znak dwukropka (:). Wszystkie *instrukcje złożone* w Pythonie (czyli instrukcje z zagnieżdżonymi kolejnymi instrukcjami) pisze się zgodnie z jednym wzorcem — z nagłówkiem zakończonym dwukropkiem, po którym następuje zagnieżdżony blok kodu wcięty w stosunku do wiersza nagłówka.

Wiersz nagłówka:
Zagnieżdżony blok instrukcji

Dwukropki jest wymagany i pominięcie go jest chyba najczęściej popełnianym przez początkujących programistów Pythona błędem — na swoich szkoleniach i kursach widziałem go tysiące razy. Każda osoba zaczynająca swoją przygodę z Pythonem szybko zapomina o znaku dwukropka. Większość edytorów do Pythona sprawia, że błąd ten jest łatwo zauważyc, a wpisywanie dwukropka w końcu staje się nieświadomym nawykiem (do tego stopnia, że można odruchowo zacząć wpisywać dwukropki do kodu napisanego w języku C++, generując tym samym wiele zabawnych komunikatów o błędach ze strony kompilatora C++).

Co usuwa Python

Choć Python wymaga dodatkowego znaku dwukropka, istnieją trzy elementy, które muszą uwzględnić programiści języków podobnych do C, a których nie ma w Pythonie.

Nawiasy są opcjonalne

Pierwszym z nich są nawiasy wokół testów znajdujących się na górze instrukcji.

```
if (x < y)
```

Nawiasy wymagane są przez składnię wielu języków podobnych do C. W Pythonie tak jednak nie jest — nawiasy możemy pominąć, a instrukcja nadal będzie działać.

```
if x < y
```

Z technicznego punktu widzenia, ponieważ każde wyrażenie można umieścić w nawiasach, wstawienie ich tutaj nie zaszkodzi i nie będą one potraktowane jako błąd. *Nie należy tego jednak robić* — to niepotrzebne nadużycie klawiatury, które na dodatek zdradza całemu światu, że jesteśmy bylymi programistami języka C, którzy nadal uczą się Pythona (sam takim kiedyś byłem). Sposób stosowany w Pythonie polega na całkowitym pomijaniu nawiasów w tego rodzaju instrukcjach.

Koniec wiersza jest końcem instrukcji

Drugim, bardziej znaczącym elementem składni, którego nie znajdziemy w Pythonie, jest znak średnika (;). W Pythonie nie trzeba kończyć instrukcji za pomocą średników, tak jak robi się to w językach podobnych do C.

```
x = 1;
```

W Pythonie istnieje ogólna reguła mówiąca, że koniec wiersza jest automatycznie końcem instrukcji znajdującej się w tym wierszu. Innymi słowy, można opuścić średniki, a instrukcja będzie działała tak samo.

```
x = 1
```

Istnieje kilka obejść tej reguły, o czym przekonamy się za chwilę. Generalnie jednak w większości kodu w Pythonie pisze się jedną instrukcję w wierszu i średniki nie są wymagane.

Również tutaj osoby troskliwe za programowaniem w języku C (o ile to w ogóle możliwe...) mogą kontynuować używanie średników na końcu każdej instrukcji — sam język nam na to pozwala. Jednak ponownie *nie należy tego robić* (naprawdę!) — kolejny raz zdradza to, że nadal jesteśmy programistami języka C, którzy jeszcze nie przestawili się na kodowanie w Pythonie. Styl stosowany w Pythonie polega na całkowitym opuszczaniu średników.

Koniec wcięcia to koniec bloku

Trzecim i ostatnim komponentem składniowym nieobecnym w Pythonie, i chyba najbardziej niezwykłym dla byłych programistów języka C (dopóki nie poużywają go przez dziesięć minut i nie ucieszą się z jego braku), jest to, że w kodzie nie wpisuje się niczego, co jawnie oznaczałoby początek i koniec zagnieżdzonego bloku kodu. Nie musimy uwzględnić begin/end, then/endif czy umieszczać wokół kodu nawiasów klamrowych, tak jak robi się to w językach podobnych do C.

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

Zamiast tego w Pythonie w spójny sposób wcina się wszystkie instrukcje w danym bloku zagnieżdżonym o tę samą odległość w prawo. Python wykorzystuje fizyczne podobieństwo instrukcji do ustalenia, gdzie blok się zaczyna i gdzie się kończy.

```
if x > y:  
    x = 1  
    y = 2
```

Przez *indentację* rozumiemy puste białe znaki znajdujące się po lewej stronie obu zagnieżdżonych instrukcji. Pythona nie interesuje sposób indentacji (można korzystać ze spacji lub tabulatorów) ani też jej ilość (można użyć dowolnej liczby spacji lub tabulatorów). Wcięcie jednego bloku zagnieżdżonego może tak naprawdę być zupełnie inne od wcięcia innego bloku. Reguła składni mówi jedynie, że w jednym bloku zagnieżdżonym wszystkie instrukcje muszą być zagnieżdżone na tę samą odległość w prawo. Jeśli tak nie jest, otrzymamy błąd składni i kod nie zostanie wykonany, dopóki nie naprawimy indentacji w spójny sposób.

Skąd bierze się składnia indentacji?

Reguła indentacji może programistom języków podobnych do C na pierwszy rzut oka wydać się niezwykła, jednak jest ona celową cechą Pythona oraz jednym ze sposobów, w jaki Python wymusza na programistach tworzenie jednolitego, regularnego i czytelnego kodu. Oznacza to, że kod musi być wyrównany w pionie, w kolumnach, zgodnie ze swoją strukturą logiczną. Rezultat jest taki, że kod staje się bardziej spójny i czytelny (w przeciwieństwie do kodu napisanego w językach podobnych do C).

Mówiąc inaczej, wyrównanie kodu zgodnie z jego strukturą logiczną jest podstawowym narzędziem uczynienia go czytelnym i tym samym łatwym w ponownym użyciu oraz późniejszym utrzymywaniu — zarówno przez nas samych, jak i przez inne osoby. Nawet osoby, które po skończeniu lektury niniejszej książki nigdy nie będą używać Pythona, powinny nabrać nawyku wyrównywania kodu w celu zwiększenia jego czytelności w dowolnym języku o strukturze blokowej. Python wymusza to, gdyż jest to częścią jego składni, jednak ma to znaczenie w każdym języku programowania i ogromny wpływ na użytkczność naszego kodu.

Doświadczenia każdej osoby mogą być różne, jednak kiedy ja byłem pełnoetatowym programistą, byłem zatrudniony przede wszystkim przy pracy z dużymi, starymi programami w języku C++, tworzonymi przez długie lata przez wiele różnych osób. Co było nie do uniknięcia, prawie każdy programista miał swój własny styl indentacji kodu. Często proszono mnie na przykład o zmianę pętli while zakodowanej w języku C, która rozpoczynała się w następujący sposób:

```
while (x > 0) {
```

Zanim jeszcze przejdziemy do samej indentacji, powiem, że istnieją trzy czy cztery sposoby układania nawiasów klamrowych w językach podobnych do C. Wiele organizacji prowadzi niemal polityczne debaty i tworzy podręczniki opisujące standardy, które powinny sobie z tym radzić (co wydaje się nieco przesadne, zważając na to, że trudno uznać to za problem programistyczny). Ignorując te spory, przejdźmy do scenariusza, z jakim często spotykałem się w kodzie napisanym w języku C++. Pierwsza osoba pracująca nad tym kodem wciążała pętlę o cztery spacje.

```
while (x > 0) {
    -----;
    -----;
```

Ta osoba w końcu awansowała, zajęła się zarządzaniem, a jej miejsce zajął ktoś, kto wolał wcięcia jeszcze bardziej przesunięte w prawo.

```
while (x > 0) {
    -----
    -----
    -----
    -----;
```

I ta osoba w pewnym momencie zmieniła pracę, a jej zadania przejął ktoś, kto wolał mniejsze wcięcia.

```
while (x > 0) {
    -----
    -----
    -----
    -----;
    -----
    -----;
}
```

I tak w nieskończoność. Blok ten kończy się nawiasem klamrowym ({}), co oczywiście sprawia, że jest on kodem ustrukturyzowanym blokowo (przynajmniej teoretycznie). W każdym języku ustrukturyzowanym blokowo (w Pythonie i innych), jeśli zagnieżdżone bloki nie są wcięte w spójny sposób, stają się trudne do interpretacji, modyfikacji czy ponownego użycia, ponieważ kod wizualnie nie odzwierciedla już swojego znaczenia logicznego. Czytelność ma duże znaczenie i indentacja jest jej istotnym komponentem.

Poniżej znajduje się kolejny przykład, który mógł nas zaboleć w przeszłości, jeśli programowaliśmy kiedyś w języku podobnym do C. Rozważmy poniższą instrukcję języka C:

```
if (x)
    if (y)
        instrukcja1;
else
    instrukcja2;
```

Z którym `if` powiązane jest `else`? Co może być zaskoczeniem, `else` jest dopełnieniem zagnieżdżonej instrukcji `if (y)`, choć wizualnie wydaje się przynależeć do zewnętrznej instrukcji `if (x)`. To klasyczna pułapka języka C, która może prowadzić do całkowicie niepoprawnej interpretacji kodu przez czytelnika i jego modyfikacji w niepoprawny sposób, co może nie zostać wykryte do momentu, gdy marsjański łazik rozbije się na wielkiej skale!

Takie coś nie może się jednak zdarzyć w Pythonie. Ponieważ indentacja jest znacząca, to, jak wygląda kod, przedstawia to, jak działa. Rozważmy odpowiednik powyższego kodu w Pythonie:

```
if x:
    if y:
        instrukcja1
    else:
        instrukcja2
```

W tym przykładzie `if`, z którym `else` wyrównane jest w pionie, to `if`, z którym `else` jest powiązane logicznie (jest to zewnętrzne `if x`). W pewnym sensie Python jest językiem typu WYSIWYG — to, co widzimy, jest tym, co otrzymujemy, ponieważ kod wykonywany jest tak, jak wygląda, bez względu na jego autora.

Jeśli te argumenty nie były w stanie przekonać kogoś o wyższości składni Pythona, podam jeszcze jedną anegdotę. Na początku mojej kariery zawodowej pracowałem w firmie rozwijającej oprogramowanie w języku C, w którym spójna indentacja nie jest wymagana. Mimo to, kiedy pod koniec dnia przesyliśmy kod do systemu kontroli wersji, wykorzystywano zautomatyzowany skrypt analizujący indentację w kodzie. Jeśli skrypt zauważał, że nie wcinamy kodu w sposób spójny, następnego dnia czekał na nas e-mail w tej sprawie, który trafiał również do naszych szefów.

Dlaczego o tym piszę? Nawet jeśli język programowania tego nie wymaga, dobrzy programiści wiedzą, że spójna indentacja kodu ma ogromny wpływ na jego czytelność i jakość. To, że Python przenosi tę kwestię na poziom składni, przez większość osób uznawane jest za wielką zaletę.

Wreszcie należy także pamiętać, że prawie każdy edytor tekstu dla programistów ma wbudowaną obsługę modelu składni Pythona. W IDLE wiersze kodu są wcinane automatycznie, kiedy piszemy blok zagnieżdzony. Naciśnięcie przycisku *Backspace* powraca o jeden poziom wcięcia i można również ustawić, jak daleko do prawej strony IDLE wciną instrukcje zagnieżdzonego bloku. Nie ma uniwersalnego standardu określającego sposób wcinania kodu. Często stosuje się cztery spacje lub jeden tabulator na poziom, jednak to każdy z nas decyduje, w jaki sposób i na jaką odległość wciąć kod. Dla bloków bardziej zagnieżdzonych odległość ta może być większa, dla bloków bliższych zewnętrznemu — mniejsza.

Jako zasadę należy przyjąć niemieszanie tabulatorów i spacji w tym samym bloku w kodzie Pythona, o ile takie formatowanie nie jest stosowane w spójny sposób. W jednym bloku powinny być używane albo tabulatory, albo spacje, ale nie jedno i drugie rozwiązanie jednocześnie (jak zobaczymy w rozdziale 12., Python 3.0 zwraca teraz błąd w przypadku niekonsekwentnego wykorzystywania tabulatorów i spacji). Spacji i tabulatorów nie powinno się tak naprawdę mieszać w żadnym ustrukturyzowanym języku programowania — taki kod powoduje znaczne problemy z czytelnością, jeśli kolejny programista ma w swoim edytorze ustawiony inny sposób wyświetlania tabulatorów. Programistom języków takich jak C może to ujść na sucho, choć nie powinno tak być — w ten sposób powstaje ogromny bałagan.

Muszę zatem powtórzyć: bez względu na to, w jakim języku się programuje, z uwagi na czytelność kodu należy wykonywać indentację w konsekwentny i spójny sposób. Jeśli ktoś nie został tego nauczony na początku swej kariery programistycznej, jego nauczyciele wykształcili mu niedźwiedzią przysługę. Większość programistów — a już zwłaszcza ci, którzy muszą czytać kod napisany przez inne osoby — uznają za zaletę to, że Python podnosi indentację na poziom składni. Co więcej, wstawianie tabulatorów zamiast nawiasów klamrowych nie jest w praktyce trudniejsze dla narzędzi zwracających kod w Pythonie. Generalnie wystarczy robić to samo co w językach podobnych do C — należy się tylko pozbyć nawiasów klamrowych, a kod będzie spełniał reguły składni Pythona.

Kilka przypadków specjalnych

Jak wspomniano wcześniej, w modelu składni Pythona:

- koniec wiersza kończy instrukcję znajdującą się w tym wierszu (bez konieczności użycia średników),
- instrukcje zagnieżdżone są łączone w bloki i wiązane ze sobą za pomocą fizycznego wcięcia (bez konieczności użycia nawiasów klamrowych).

Te reguły decydują o prawie całym kodzie napisanym w Pythonie, z jakim można się spotkać w praktyce. Python udostępnia również kilka reguł specjalnego przeznaczenia, które pozwalają na dostosowanie instrukcji i zagnieżdżonych bloków instrukcji do własnych potrzeb.

Przypadki specjalne dla reguły o końcu wiersza

Choć instrukcje normalnie pojawiają się po jednej na wiersz, można również umieścić w wierszu kodu Pythona więcej niż jedną instrukcję, rozdzielając je od siebie średnikami.

```
a = 1; b = 2; print(a + b) # Trzy instrukcje w wierszu
```

To jedyne miejsce, w którym w Pythonie wymagane są średniki — jako *separatory instrukcji*. Działa to jednak tylko wtedy, gdy połączone w ten sposób instrukcje nie są instrukcjami złożonymi. Innymi słowy, można połączyć ze sobą jedynie proste instrukcje, takie jak przypisywanie, wyświetlanie za pomocą `print` czy wywołania funkcji. Instrukcje złożone nadal muszą pojawiać się w osobnych wierszach (w przeciwnym razie w jednym wierszu można by było umieścić cały program, co nie przysporzyłoby nam popularności wśród współpracowników).

Druga reguła specjalna dla instrukcji jest odwrotna: jedna instrukcja może rozciągać się na kilka wierszy. By to zadziało, wystarczy umieścić część instrukcji w parze nawiasów — zwykłych `(())`, kwadratowych `[]` lub klamrowych `{ }`. Kod umieszczony w tych konstrukcjach

może znajdować się w kilku wierszach. Instrukcja nie kończy się, dopóki Python nie dojdzie do wiersza zawierającego zamkającą część nawiasu. Przykładem może być rozciągający się na kilka wierszy literal listy.

```
mlist = [111,  
         222,  
         333]
```

Ponieważ kod umieszczony jest w parze nawiasów kwadratowych, Python po prostu przechodzi do kolejnego wiersza aż do momentu napotkania nawiasu zamkającego. Nawiasy klamrowe otaczające słowniki (a także literaly zbiorów oraz słowniki i listy składane w wersji 3.0) także mogą się rozciągać na kilka wierszy, natomiast zwykłe nawiasy mogą mieścić krótki, wywołania funkcji i wyrażenia. Indentacja wiersza z kontynuacją nie ma znaczenia, choć zdrowy rozsądek nakazuje jakoś wyrównać ze sobą kolejne wiersze dla celów czytelności.

Nawiasy są wszechstronnym narzędziem. Ponieważ można w nich umieścić dowolne wyrażenie, samo wstawienie lewego nawiasu pozwala na przejście do kolejnego wiersza i kontynuowanie instrukcji tam.

```
X = (A + B +  
      C + D)
```

Ta technika działa zresztą również w przypadku instrukcji złożonych. Kiedy tylko potrzebujemy zakodować duże wyrażenie, wystarczy opakować je w nawiasy, by móc je kontynuować w kolejnym wierszu.

```
if (A == 1 and  
    B == 2 and  
    C == 3):  
    print('mielonka' * 3)
```

Starsza reguła pozwala również na kontynuację w następnym wierszu, kiedy poprzedni kończy się ukośnikiem lewym.

```
X = A + B + \  
    C + D
```

Podatne na błędy rozwiążanie alternatywne

Ta technika alternatywna jest jednak przestarzała i raczej już nielubiana, ponieważ trudno jest zauważać i utrzymywać ukośniki lewe, a do tego dość bezwzględna (po ukośniku nie może być spacji). Pominięcie ukośnika może także prowadzić do nieprzewidzianych rezultatów, jeśli kolejny wiersz zostanie wzięty za nową instrukcję. Jest to również kolejny powrót do języka C, gdzie technika ta jest często wykorzystywana w makrach „#define”. W świecie Pythona należy zachowywać się jak programista Pythona, a nie języka C.

Przypadki specjalne dla reguły o indentacji bloków

Jak wspomniano wcześniej, instrukcje w zagnieźdzonym bloku kodu są zazwyczaj wiązane ze sobą dzięki wcinaniu na tę samą odległość w prawą stronę. W specjalnym przypadku ciało instrukcji złożonej może pojawić się w tym samym wierszu co jej nagłówek, po znaku dwukropka.

```
if x > y: print(x)
```

Pozwala to na kodowanie jednowierszowych instrukcji if czy pętli. Tutaj jednak zadziała to tylko wtedy, gdy ciało instrukcji złożonej nie zawiera żadnych instrukcji złożonych. Mogą się tam znajdować jedynie proste instrukcje — przypisania, instrukcje print, wywołania funkcji i tym podobne. Większe instrukcje nadal muszą być umieszczane w osobnych wierszach.

Dodatkowe części instrukcji złożonych (na przykład część `else` z `if`, z którą spotkamy się później) również muszą znajdować się w osobnych wierszach. Ciało instrukcji może składać się z kilku prostych instrukcji rozdzielonych średnikami, jednak zazwyczaj nie jest to pochwalane.

Ogólnie rzecz biorąc — nawet jeśli nie zawsze jest to wymagane — jeżeli będziemy umieszczać każdą instrukcję w osobnym wierszu i zawsze wcinać zagnieżdżone bloki, nasz kod będzie łatwiejszy do odczytania i późniejszej modyfikacji. Co więcej, niektóre narzędzia służące do profilowania kodu i testów pokrycia mogą nie być w stanie rozróżnić kilku instrukcji umieszczonych w jednym wierszu czy też oddzielić od siebie nagłówka i ciała jednowierszowej instrukcji złożonej. W Pythonie prawie zawsze prostota daje najlepsze rezultaty.

By zobaczyć najważniejszy i najczęściej spotykany wyjątek od jednej z tych reguł (użycie jednowierszowej instrukcji `if` w celu wyjścia z pętli), przejdźmy do kolejnego podrozdziału i zajmijmy się pisaniem prawdziwego kodu.

Szybki przykład — interaktywne pętle

Wszystkie te reguły składni zobaczymy w działaniu, kiedy w kolejnych rozdziałach będziemy omawiać określone instrukcje złożone Pythona. Działają one w ten sam sposób w całym języku. Na początek zajmiemy się krótkim, realistycznym przykładem demonstrującym sposób łączenia składni i zagnieżdżania instrukcji, a także wprowadzającym przy okazji kilka instrukcji.

Prosta pętla interaktywna

Przypuśćmy, że poproszono nas o napisanie w Pythonie programu wchodzącego w interakcję z użytkownikiem w oknie konsoli. Być może będziemy przyjmować dane wejściowe w celu przesłania ich do bazy danych bądź odczytywać liczby wykorzystane w obliczeniach. Bez względu na cel potrzebna nam będzie pętla odczytująca dane wejściowe wpisywane przez użytkownika na klawiaturze i wyświetlająca dla nich wynik. Innymi słowy, musimy utworzyć klasyczną pętlę odczytaj-oblicz-wyświetl.

W Pythonie typowy kod takiej pętli interaktywnej może wyglądać jak poniższy przykład.

```
while True:  
    reply = input('Wpisz tekst: ')  
    if reply == 'stop': break  
    print(reply.upper())
```

Kod ten wykorzystuje kilka nowych koncepcji.

- W kodzie użyto pętli `while` — najbardziej ogólnej instrukcji pętli Pythona. Instrukcję `while` omówimy bardziej szczegółowo później. Mówiąc w skrócie, zaczyna się ona od słowa `while`, a po nim następuje wyrażenie, którego wynik interpretowany jest jako prawda lub fałsz. Później znajduje się zagnieżdżony blok kodu powtarzany, dopóki test znajdujący się na górze jest prawdą (słowo `True` z przykładu jest zawsze prawdą).
- Wbudowana funkcja `input`, z którą spotkaliśmy się już wcześniej, wykorzystana została tutaj do wygenerowania danych wejściowych z konsoli. Wyświetla ona w charakterze zachęty łańcuch znaków będący opcjonalnym argumentem i zwraca odpowiedź wpisaną przez użytkownika w postaci łańcucha znaków.

- W kodzie pojawia się również jednowierszowa instrukcja `if` wykorzystująca regułę specjalną dotyczącą zagnieźdzonych bloków. Ciało instrukcji `if` pojawia się po dwukropku w jej nagłówku, zamiast znajdować się w kolejnym, wciętym wierszu. Oba alternatywne sposoby zadziałają, jednak dzięki metodzie zastosowanej powyżej zaoszczędziliśmy jeden wiersz.
- Instrukcja `break` Pythona wykorzystywana jest do natychmiastowego wyjścia z pętli. Powoduje ona całkowite wyskoczenie z instrukcji pętli i program kontynuowany jest po pętli. Bez takiej instrukcji wyjścia pętla byłaby nieskończona, ponieważ wynik jej testu będzie zawsze prawdziwy.

W rezultacie takie połączenie instrukcji oznacza: „Wczytuj wiersze wpisane przez użytkownika i wyświetl je zapisane wielkimi literami, dopóki nie wpisze on słowa stop”. Istnieją inne sposoby zakodowania takiej pętli, jednak metoda zastosowana powyżej jest w Pythonie często spotykana.

Warto zauważyć, że wszystkie trzy wiersze zagnieździone pod wierszem nagłówka instrukcji `while` są wcinane na tę samą odległość. Ponieważ są one wyrównane w pionie jak jedna kolumna, są blokiem kodu powiązanego z testem `while` i powtarzanego. Blok ciała pętli zostaje zakończony albo przez koniec pliku źródłowego, albo przez umieszczenie mniej wciętej instrukcji.

Po wykonaniu kodu możemy otrzymać interakcję podobną do poniższej.

Wpisz tekst:**mielonka**

MIELONKA

Wpisz tekst:**42**

42

Wpisz tekst:**stop**



Uwaga na temat wersji: Przykład ten napisany jest dla Pythona 3.0. W Pythonie 2.6 i wcześniejszych wersjach kod ten działa tak samo, jednak zamiast funkcji `input` należy użyć `raw_input`, a także można pominąć zewnętrzne nawiasy instrukcji `print`. W wersji 3.0 funkcja ta zmieniła nazwę, a `print` jest funkcją wbudowaną, a nie instrukcją (więcej informacji na ten temat znajduje się w kolejnym rozdziale).

Wykonywanie obliczeń na danych użytkownika

Nasz skrypt działa, jednak teraz założmy, że zamiast zmiany tekstuowego łańcucha znaków na zapisany wielkimi literami wolelibyśmy wykonać jakieś obliczenia na danych liczbowych — na przykład podnosząc je do kwadratu. By osiągnąć zamierzony efekt, możemy spróbować z poniższymi instrukcjami.

```
>>> reply = '20'
>>> reply ** 2
...pominieto tekst bledu...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Takie coś nie zadziała w naszym skrypcie (zgodnie z informacjami z poprzedniej części książki), ponieważ Python nie przekształci typów obiektów w wyrażeniach, jeśli wszystkie one nie są typami liczbowymi — a tak nie jest, ponieważ dane wpisywane przez użytkownika są zawsze w skrypcie zwracane jako łańcuchy znaków. Nie możemy podnieść łańcucha cyfr do potęgi, dopóki ręcznie nie przekształcimy go na liczbę całkowitą.

```
>>> int(reply) ** 2  
400
```

Mając takie informacje, możemy teraz poprawić naszą pętlę w taki sposób, by wykonywała ona niezbędne obliczenia. By to sprawdzić, należy wpisać poniższy kod do pliku:

```
while True:  
    reply = input('Wpisz tekst:')  
    if reply == 'stop': break  
    print(int(reply) ** 2)  
print('Koniec')
```

Ten skrypt wykorzystuje jednowierszową instrukcję `if` do wyjścia z pętli w momencie wpisania słowa „stop”, jednak przy okazji konwertuje również dane wejściowe na postać liczbową w celu umożliwienia obliczeń. Wersja ta dodaje także komunikat końcowy umieszczony na dole. Ponieważ instrukcja `print` z ostatniego wiersza nie jest wcięta na tę samą odległość co zagnieżdżony blok kodu, nie jest ona uważana za część ciała pętli i zostanie wykonana tylko raz — po wyjściu z pętli.

```
Wpisz tekst:2  
4  
Wpisz tekst:40  
1600  
Wpisz tekst:stop  
Koniec
```

Jedna uwaga: zakładam tutaj, że powyższy kod zostanie przechowyany w pliku skryptu i stamtąd wykonany. Jeśli kod ten wpisuje się do interaktywnego wiersza poleceń, należy pamiętać o uwzględnieniu pustego wiersza (czyli dwa razy nacisnąć klawisz *Enter*) przed ostatnim wywołaniem `print` w celu zakończenia pętli. Ostatnie wywołanie `print` w trybie interaktywnym nie ma zbyt dużego sensu (będzie trzeba je wpisać po interakcji z pętlą).

Obsługa błędów za pomocą sprawdzania danych wejściowych

Jak na razie wszystko działa, ale co się stanie, kiedy dane wejściowe będą niepoprawne?

```
Wpisz tekst:xxx  
...pominieto tekst błędu...  
ValueError: invalid literal for int() with base 10: 'xxx'
```

Wbudowana funkcja `int` w momencie wystąpienia błędu zwraca tutaj wyjątek. Jeśli chcemy, by nasz skrypt miał większe możliwości, możemy z wyprzedzeniem sprawdzić zawartość łańcucha znaków za pomocą metody obiektu łańcucha znaków o nazwie `isdigit`.

```
>>> S = '123'  
>>> T = 'xxx'  
>>> S.isdigit(), T.isdigit()  
(True, False)
```

Daje nam to również pretekst do dalszego zagnieżdżenia instrukcji w naszym przykładzie. Poniższa nowa wersja naszego interaktywnego skryptu wykorzystuje pełną instrukcję `if` do obejścia problemu wyjątków pojawiających się w momencie wystąpienia błędu.

```
while True:  
    reply = input('Wpisz tekst:')  
    if reply == 'stop':  
        break  
    elif not reply.isdigit():  
        print('Niepoprawnie!' * 5)
```

```
else:  
    print(int(reply) ** 2)  
print('Koniec')
```

Instrukcję `if` przestudiujemy szczegółowo w rozdziale 12. Jest ona dość łatwym narzędziem służącym do kodowania logiki w skryptach. W pełnej formie składa się ze słowa `if`, po którym następuje test, powiązany blok kodu, jeden lub większa liczba opcjonalnych testów `elif` (od `else if`) i bloków kodu, a na dole opcjonalna część `else` z powiązanym blokiem kodu, który służy za wynik domyślny. Kiedy pierwszy test zwraca wynik będący prawdą, Python wykonuje blok kodu z nim powiązany — od góry do dołu. Jeśli wszystkie testy będą zwracały wyniki będące fałszem, wykonywany jest kod z części `else`.

Części `if`, `elif` i `else` w powyższym przykładzie są powiązanymi częściami tej samej instrukcji, ponieważ wszystkie są ze sobą wyrównane w pionie (to znaczy mają ten sam poziom wcięcia). Instrukcja `if` rozciąga się od słowa `if` do początku instrukcji `print` w ostatnim wierszu skryptu. Z kolei cały blok `if` jest częścią pętli `while`, ponieważ w całości wcięty jest pod wierszem nagłówka tej pętli. Zagnieżdżanie instrukcji z czasem stanie się dla każdego naturalne.

Kiedy wykonamy nasz nowy skrypt, jego kod przechwyci błąd przed jego wystąpieniem i wyświetli (dość głupi) komunikat w celu podkreślenia tego.

```
Wpisz tekst:5  
25  
Wpisz tekst:xyz  
Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!  
Wpisz tekst:10  
100  
Wpisz tekst:stop
```

Obsługa błędów za pomocą instrukcji `try`

Powyzsze rozwiązanie działa, jednak, jak zobaczymy w dalszej części książki, najbardziej uniwersalnym sposobem obsługi wyjątków w Pythonie jest przechwytywanie ich i poradzenie sobie z błędem za pomocą instrukcji `try`. Instrukcję tę omówimy bardziej dogłębnie w siódmej części książki, jednak już teraz możemy pokazać, że użycie tutaj `try` może sprawić, iż kod niektórym osobom wyda się prostszy od poprzedniej wersji.

```
while True:  
    reply = input('Wpisz tekst:')  
    if reply == 'stop': break  
    try:  
        num = int(reply)  
    except:  
        print('Niepoprawnie!' * 5)  
    else:  
        print(int(reply) ** 2)  
print('Koniec')
```

Ta wersja działa dokładnie tak samo jak poprzednia, jednak zastąpiliśmy dosłowne sprawdzanie błędu kodem zakładającym, że konwersja będzie działała, i opakowaliśmy go kodem z obsługą wyjątku, który zatroszczy się o przypadki, kiedy konwersja nie zadziała. Instrukcja `try` składa się ze słowa `try`, następującego po nim głównego bloku kodu (z działaniem, jakie próbujemy uzyskać), później z części `except` podającej kod obsługujący błędy i części `else`, która jest wykonywana, kiedy żaden wyjątek nie zostanie zgłoszony w części `try`. Python najpierw próbuje wykonać część `try`, a później albo część `except` (jeśli wystąpi wyjątek), albo `else` (jeśli wyjątek się nie pojawi).

Jeśli chodzi o zagnieździanie instrukcji, ponieważ poziom wcięcia try, except i else jest taki sam, wszystkie one uznawane są za część tej samej instrukcji try. Warto zauważyc, że część else powiązana jest tutaj z try, a nie z if. Jak widzieliśmy wcześniej, else może się w Pythonie pojawiać w instrukcjach if, ale także w instrukcjach try i pętlach — to indentacja informuje nas, której instrukcji jest częścią. W tym przypadku instrukcja try rozciąga się od słowa try do kodu wciętego pod słowem else, ponieważ else wcięte jest na tym samym poziomie co try. Instrukcja if w tym kodzie składa się z jednego wiersza i kończy się po break.

Do instrukcji try powrócimy w dalszej części książki. Na razie warto być świadomym tego, że ponieważ try można wykorzystać do przechwycenia dowolnego błędu, instrukcja ta redukuje ilość kodu sprawdzającego błędy, jaki musimy napisać. Jest także bardzo uniwersalnym sposobem radzenia sobie z niezwykłymi przypadkami. Gdybyśmy chcieli na przykład obsłużywać wprowadzanie liczb zmiennoprzecinkowych, zamiast ograniczać się do liczb całkowitych, użycie try byłoby o wiele łatwiejsze od ręcznego testowania błędów — wywołalibyśmy po prostu float i przechwycili wyjątki, zamiast próbować analizować całą potencjalną składnię liczb zmiennoprzecinkowych.

Kod zagnieżdżony na trzy poziomy głębokości

Przyjrzyjmy się teraz ostatniej mutacji naszego skryptu. Zagnieździanie może być jeszcze głębsze — możemy na przykład rozgałęzić jedną z alternatyw w oparciu o względową wielkość poprawnych danych wejściowych.

```
while True:
    reply = input('Wpisz tekst:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Niepoprawnie!' * 5)
    else:
        num = int(reply)
        if num < 20:
            print('mało')
        else:
            print(num ** 2)
print('Koniec')
```

Ta wersja zawiera instrukcję if zagnieżdżoną w części else innej instrukcji if, która z kolei jest zagnieżdżona w pętli while. Kiedy kod jest warunkowy lub powtarzany, tak jak ten, po prostu wcinamy go jeszcze dalej w prawo. W rezultacie otrzymujemy coś podobnego do poprzedniej wersji, ale dla liczb mniejszych od 20 wyświetlony zostanie komunikat „mało”.

```
Wpisz tekst:19
mało
Wpisz tekst:20
400
Wpisz tekst:mielonka
Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!Niepoprawnie!
Wpisz tekst:stop
Koniec
```

Podsumowanie rozdziału

Powyższe informacje kończą nasze krótkie omówienie podstaw składni instrukcji Pythona. W niniejszym rozdziale wprowadziliśmy ogólne reguły kodowania instrukcji oraz bloków kodu. Jak widzieliśmy, w Pythonie zazwyczaj umieszcza się jedną instrukcję na wiersz i wcina wszystkie instrukcje zagnieżdżonego bloku kodu na tę samą odległość (indentacja jest częścią składni Pythona). Przyjrzelismy się również kilku odstępstwom od tych reguł, w tym wierszom z kontynuacją oraz jednowierszowym testom i pętlom. Wreszcie zastosowaliśmy te koncepcje w praktyce w interaktywnym skrypcie, który zademonstrował kilka instrukcji i pokazał nam, jak tak naprawdę działa składnia instrukcji.

W kolejnym rozdziale zaczniemy zagłębiać się w instrukcje, omawiając bardziej szczegółowo każdą z podstawowych instrukcji proceduralnych Pythona. Jak już jednak widzieliśmy, wszystkie instrukcje zachowują się zgodnie z ogólnymi regułami zaprezentowanymi tutaj.

Sprawdź swoją wiedzę — quiz

1. Jakie trzy elementy wymagane są w językach podobnych do C, ale pomijane w Pythonie?
2. W jaki sposób w Pythonie normalnie kończy się instrukcja?
3. W jaki sposób instrukcje z zagnieżdżonego bloku kodu są zazwyczaj ze sobą wiązane w Pythonie?
4. W jaki sposób możemy rozciągnąć pojedynczą instrukcję na kilka wierszy?
5. W jaki sposób można utworzyć instrukcję złożoną zajmującą jeden wiersz?
6. Czy istnieje jakiś powód uzasadniający umieszczenie średnika na końcu instrukcji w Pythonie?
7. Do czego służy instrukcja `try`?
8. Co jest najczęściej popełnianym przez osoby początkujące błędem w kodowaniu w Pythonie?

Sprawdź swoją wiedzę — odpowiedzi

1. Języki podobne do C wymagają stosowania nawiasów wokół testów w niektórych instrukcjach, średników na końcu instrukcji i nawiasów klamrowych wokół zagnieżdżonych bloków kodu.
2. Koniec wiersza kończy instrukcję umieszczoną w tym wierszu. Alternatywnie, jeśli w tym samym wierszu znajduje się większa liczba instrukcji, można je zakończyć średnikami. W podobny sposób, kiedy instrukcja rozciąga się na wiele wierszy, trzeba ją zakończyć za pomocą zamknięcia pary nawiasów.
3. Instrukcje w bloku zagnieżdżonym są wcinane o taką samą liczbę tabulatorów lub spacji.
4. Instrukcja może rozciągać się na kilka wierszy dzięki umieszczeniu jej części w nawiasach zwykłych, kwadratowych lub klamrowych. Instrukcja taka kończy się, kiedy Python napotka wiersz zawierający zamykającą część pary nawiasów.

5. Ciało instrukcji złożonej można przesunąć do wiersza nagłówka po dwukropku, jednak tylko wtedy, gdy nie zawiera ono żadnych instrukcji złożonych.
6. Możemy to zrobić tylko wtedy, gdy chcemy zmieścić w jednym wierszu kodu więcej niż jedną instrukcję. Ale nawet wówczas działa to tylko w sytuacji, w której żadna z instrukcji nie jest złożona, i jest odradzane, ponieważ prowadzi do kodu trudniejszego w odczycie.
7. Instrukcja `try` wykorzystywana jest do przechwytywania wyjątków (błędów) i radzenia sobie z nimi w skrypcie Pythona. Zazwyczaj jest alternatywą dla ręcznego sprawdzania błędów w kodzie.
8. Błądem najczęściej popełnianym przez osoby poczatkujące jest zapominanie o dodaniu dwukropka na końcu wiersza nagłówka instrukcji złożonej. Jeśli ktoś tego błędu nigdy jeszcze nie popełnił, na pewno zrobi to wkrótce!

Przypisania, wyrażenia i wyświetlanie

Po krótkim wprowadzeniu do składni instrukcji Pythona niniejszy rozdział rozpoczyna nasze pogłębione omówienie poszczególnych instrukcji tego języka. Zaczniemy od podstaw — przypisania, instrukcji wyrażeń i wyświetlania za pomocą `print`. Ze wszystkimi już się spotkaliśmy, jednak teraz uzupełnimy szczegóły pominięte wcześniej. Choć wszystkie te operacje są stosunkowo proste, jak się niebawem okaże, istnieją opcjonalne odmiany każdego z tych typów instrukcji, które przydadzą się nam przy pisaniu w Pythonie prawdziwych programów.

Instrukcje przypisania

Instrukcji przypisania używamy już od jakiegoś czasu w celu przypisywania obiektów do nazw (zmiennych). W najbardziej podstawowej postaci piszemy *cel* przypisania po lewej stronie znaku równości (=) i *obiekt* przypisywany po prawej stronie. Cel znajdujący się z lewej strony może być nazwą lub komponentem obiektu, natomiast obiekt z prawej strony może być dowolnym wyrażeniem obliczającym obiekt. Najczęściej przypisania są proste, jednak należy pamiętać o kilku kwestiach.

- **Przypisania tworzą referencje do obiektów.** Jak wspomniano w rozdziale 6., w Pythonie przypisania przechowują referencje do obiektów w zmiennych lub komponentach struktur danych. Zawsze tworzą referencje do obiektów, a nie kopiąją te obiekty. Z tego powodu zmienne Pythona bardziej przypominają wskaźniki niż obszary przechowywania danych.
- **Zmienne tworzone są przy pierwszym przypisaniu.** Python tworzy nazwy zmiennych za pierwszym razem, gdy przypisujemy do nich jakąś wartość (na przykład referencję do obiektu), dlatego nie jest konieczne deklarowanie ich z wyprzedzeniem. Niektóre (ale nie wszystkie) miejsca w strukturach danych tworzone są w momencie przypisania (na przykład wpisy słownika, niektóre atrybuty obiektu). Po przypisaniu zmieniona zastępowana jest wartością, do której się odnosi, za każdym razem, gdy pojawią się ona w wyrażeniu.
- **Przed odniesieniem się do zmiennych trzeba je najpierw przypisać.** Błądem jest wykorzystywanie zmiennej, do której nie przypisaliśmy jeszcze wartości. Kiedy tak zrobimy, Python zwraca wyjątek, zamiast próbować zwracać jakiś rodzaj niejednoznacznej wartości domyślnej. Gdyby zwracał wartość domyślną, o wiele trudniej byłoby znaleźć w kodzie błędy literowe.

- Przypisania niejawne.** W tej części zajmowaliśmy się przede wszystkim instrukcją `=`, jednak przypisania mają miejsce w wielu kontekstach. Jak zobaczymy później, importowanie modułów, definicje funkcji i klas, zmienne pętli `for` i argumenty funkcji są przypisaniami niejawnymi. Ponieważ przypisanie odbywa się w ten sam sposób w każdym miejscu, wszystkie te konteksty po prostu łączą nazwy z referencjami do obiektów w momencie wykonania.

Formy instrukcji przypisania

Choć przypisanie jest w Pythonie ogólną i wszechobecną koncepcją, w tym rozdziale interesują nas przede wszystkim *instrukcje przypisania*. W tabeli 11.1 zilustrowano różne formy instrukcji przypisania w Pythonie.

Tabela 11.1. Formy instrukcji przypisania

Operacja	Interpretacja
<code>spam = 'Mielonka'</code>	Forma podstawowa
<code>spam, ham = 'mniam', 'MNIAM'</code>	Przypisanie krotki (pozycyjne)
<code>[spam, ham] = ['mniam', 'MNIAM']</code>	Przypisanie listy (pozycyjne)
<code>a, b, c, d = 'mielonka'</code>	Przypisanie sekwencji, uogólnione
<code>a, *b = 'mielonka'</code>	Rozszerzone rozpakowanie sekwencji (Python 3.0)
<code>spam = ham = 'lunch'</code>	Przypisanie z wieloma celami
<code>spam += 42</code>	Przypisanie rozszerzone (odpowiednik <code>spam = spam + 42</code>)

Pierwsza forma z tabeli 11.1 jest bez wątpienia najczęściej spotykana i polega na połączeniu nazwy (lub komponentu struktury danych) z pojedynczym obiektem. Pozostałe wpisy z tabeli reprezentują formy specjalne i opcjonalne, które w praktyce często przydają się programistom.

Przypisania rozpakowujące krotki i listy

Druga i trzecia forma z tabeli są ze sobą powiązane. Kiedy kodujemy krotki czy listy po lewej stronie znaku `=`, Python łączy obiekty z prawej strony z celami z lewej strony w pary zgodnie z ich pozycją i przypisuje je od lewej do prawej strony. W drugim wierszu tabeli 11.1 nazwa `spam` przypisywana jest do łańcucha znaków `'yum'`, a nazwa `ham`łączona jest z łańcuchem `'YUM'`. Wewnętrznie Python najpierw z elementów z prawej strony robi krotkę, dlatego często nazywa się tę operację przypisaniem rozpakowującym krotkę.

Przypisania sekwencji

W nowszych wersjach Pythona przypisania krotki i listy zostały uogólnione jako instancje czegoś, co możemy teraz nazwać wywołaniem *przypisania sekwencji*. Dowolną sekwencję nazw można przypisać do dowolnej sekwencji wartości, a Python przypisuje elementy po jednym na raz zgodnie z ich pozycją. Tak naprawdę możemy mieszać i dopasowywać typy wykorzystywanych sekwencji. W czwartym wierszu tabeli 11.1 w parę połączono krotkę nazw i łańcuch znaków — a zostaje przypisane do `'s'`, `b` do `'p'` i tak dalej.

Rozszerzone rozpakowanie sekwencji

W Pythonie 3.0 wprowadzono nową formę przypisania sekwencji, która daje więcej możliwości wyboru fragmentu sekwencji do przypisania. W przykładzie z piątego wiersza

tabeli 11.1 zmiennej a zostanie przypisana litera "m", natomiast zmiennej b — ciąg znaków "ielonka". Rozszerzona składnia rozpakowania sekwencji upraszcza tego typu operacje, pozwalając uniknąć stosowania wycinków w przypisaniach.

Przypisania z wieloma celami

Piąty wiersz tabeli 11.1 prezentuje formę przypisania z wieloma celami. W tej formie Python przypisuje referencję do jednego obiektu (znajdującego się najdalej na prawo) do wszystkich celów znajdujących się po lewej stronie. W tabeli do nazw spam i ham przypisane są referencje do tego samego obiektu łańcucha znaków, 'lunch'. Rezultat jest taki sam, jakbyśmy w kodzie napisali ham = 'lunch', a potem spam = ham, ponieważ ham ma wartość oryginalnego obiektu łańcucha znaków (a nie osobnej kopii tego obiektu).

Przypisania rozszerzone

Ostatni wiersz tabeli 11.1 jest przykładem *przypisania rozszerzonego* — skrótu łączącego wyrażenie i przypisanie w zwięzły sposób. Przypisanie spam += 42 ma na przykład ten sam efekt co spam = spam + 42, jednak forma rozszerzona wymaga mniej pisania i jest generalnie szybsza do wykonania. Dodatkowo w przypadku, gdy obiekt jest mutowalny i obsługuje daną operację, przypisania rozszerzone mogą działać szybciej dzięki zastosowaniu modyfikacji obiektu w miejscu zamiast jego kopiowania. W Pythonie istnieje jedno przypisanie rozszerzone dla każdego operatora wyrażenia binarnego.

Przypisanie sekwencji

W książce korzystaliśmy już z podstawowej formy przypisania. Poniżej znajduje się kilka prostych przykładów przypisania rozpakowującego sekwencje.

```
% python
>>> nudge = 1
>>> wink = 2
>>> A, B = nudge, wink
>>> A, B
(1, 2)
# Przypisanie krotki
# Jak A = nudge; B = wink
>>> [C, D] = [nudge, wink]
>>> C, D
(1, 2)
# Przypisanie listy
```

W trzecim wierszu tego kodu tak naprawdę tworzymy dwie krotki — pomijamy tylko ich nawiasy. Python łączy w pary wartości z krotki z prawej strony operatora przypisania z wartościami z krotki z lewej strony i przypisuje po jednej wartości na raz.

Przypisanie krotek prowadzi do często stosowanej sztuczki wprowadzonej w rozwiązańach do ćwiczeń z drugiej części książki. Ponieważ Python tworzy tymczasową krotkę, która zapisuje oryginalne wartości zmiennych z prawej strony w czasie wykonywania instrukcji, przypisania tego typu służą również do *zamiany* wartości dwóch zmiennych bez tworzenia osobnej zmiennej tymczasowej — krotka z prawej strony automatycznie pamięta poprzednie wartości zmiennych.

```
>>> nudge = 1
>>> wink = 2
>>> nudge, wink = wink, nudge
# Krotki: zamiana wartości
# Jak T = nudge; nudge = wink; wink = T
>>> nudge, wink
(2, 1)
```

Tak naprawdę oryginalne formy przypisania krotek i list w Pythonie zostały z czasem uogólnione w taki sposób, by przyjmowały dowolny typ sekwencji po prawej stronie, pod warunkiem, że

długość sekwencji będzie taka sama. Krotkę wartości można na przykład przypisać do listy zmiennych, a łańcuch znaków do krotki zmiennych. W każdym przypadku Python przypisuje elementy z sekwencji po prawej stronie do zmiennych z sekwencji po lewej stronie zgodnie z ich pozycją, od lewej do prawej.

```
>>> [a, b, c] = (1, 2, 3)          # Przypisanie krotki wartości do listy nazw
>>> a, c
(1, 3)
>>> (a, b, c) = "ABC"            # Przypisanie łańcucha znaków do krotki
>>> a, c
('A', 'C')
```

Z technicznego punktu widzenia przypisanie sekwencji tak naprawdę przyjmuje po prawej stronie dowolny obiekt, po którym można iterować, nie tylko sekwencję. Jest to bardziej uniwersalna koncepcja, którą omówimy w rozdziałach 14. oraz 20.

Zaawansowane wzorce przypisywania sekwencji

Jedna uwaga: choć możemy mieszać i dopasowywać typy sekwencji znajdujące się wokół symbolu `=`, nadal musimy mieć tę samą liczbę elementów po prawej stronie co zmiennych po lewej stronie. W przeciwnym razie otrzymamy błąd. Python 3.0 pozwala zachować wyższy poziom uogólnienia dzięki rozszerzonej składni rozpakowania, opisanej w następnym punkcie rozdziału. Z reguły jednak (a w 2.X zawsze) liczba zmiennych po obydwu stronach równości powinna się zgadzać.

```
>>> string = 'JAJO'
>>> a, b, c, d = string           # Ta sama liczba elementów po obu stronach
>>> a, d
('J', 'O')

>>> a, b, c = string             # Błąd: różna liczba elementów
...pominieto tekst błędu...
ValueError: too many values to unpack
```

W celu uzyskania bardziej uniwersalnego rozwiązania musimy skorzystać z wycinków. Istnieje kilka sposobów zastosowania wycinków w taki sposób, by ostatni przykład działał.

```
>>> a, b, c = string[0], string[1], string[2:] # Indeksowanie i wycinek
>>> a, b, c
('J', 'A', 'JO')

>>> a, b, c = list(string[:2]) + [string[2:]] # Wycinek i konkatenacja
>>> a, b, c
('J', 'A', 'JO')

>>> a, b = string[:2]                  # To samo w prostszej formie
>>> c = string[2:]
>>> a, b, c
('J', 'A', 'JO')

>>> (a, b), c = string[:2], string[2:]      # Zagnieżdżone sekwencje
>>> a, b, c
('J', 'A', 'JO')
```

W ostatnim przykładzie tego kodu widać, że możemy nawet przypisać sekwencje *zagnieżdżone*, ponieważ Python, zgodnie z oczekiwaniemi, rozpakowuje ich części, uwzględniając ich kształt. W tym przypadku przypisujemy krotkę dwuelementową dokładnie w taki sposób, w jaki byśmy ją zakodowali; pierwszym elementem jest zagnieżdzona sekwencja (łańcuch znaków).

```
>>> ((a, b), c) = ('JA', 'JO')          # Połączone w pary zgodnie z kształtem i pozycją
>>> a, b, c
('J', 'A', 'JO')
```

Python łączy pierwszy łańcuch znaków z prawej strony ('JA') z pierwszą krotką z lewej strony ((a, b)) i przypisuje po jednym znaku na raz przed przypisaniem całego drugiego łańcucha znaków ('JO') do zmiennej c za jednym razem. W takiej sytuacji kształt zagnieźdżenia sekwencji z lewej strony musi odpowiadać obiekowi z prawej strony. Takie przypisania zagnieźdzonych sekwencji są nieco bardziej zaawansowane i rzadziej się je spotyka, jednak mogą być wygodne dla celów wyboru części struktur danych o znany kształcie.

Technika ta działa na przykład również w listach argumentów funkcji, ponieważ argumenty funkcji przekazywane są poprzez przypisanie (co zobaczymy w rozdziale 13. książki).

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...
for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ...      # Zagnieżdżone przypisanie do krotki
```

W uchwadze w rozdziale 18. napisałem, że zagnieździone rozpakowanie do krotki (a dokładniej do sekwencji) działa również w przypadku list argumentów funkcji w wersji 2.6 (ale nie działa w 3.0), ponieważ również argumenty funkcji są przekazywane przez przypisania.

```
def f(((a, b), c)):
    f(((1, 2), 3))      # Również dla argumentów w Python 2.6, ale nie 3.0
```

Przypisania rozpakowujące sekwencje spowodowały również pojawienie się kolejnej sztuczki programistycznej w Pythonie — przypisania serii liczb całkowitych do zbioru zmiennych.

```
>>> red, green, blue = range(3)
>>> red, blue
(0, 2)
```

Powyższy kod inicjalizuje trzy zmienne i nadaje im wartości liczb całkowitych 0, 1 oraz 2 (to odpowiednik *wyliczeniowych* typów danych obecnych w innych językach programowania). By zrozumieć ten przykład, trzeba wiedzieć, że wbudowana funkcja `range` generuje listę kolejnych liczb całkowitych.

```
>>> range(3)          # W Pythonie 3.0 można spróbować list(range(3))
[0, 1, 2]
```

Ponieważ funkcja `range` jest często wykorzystywana w pętlach `for`, więcej powiemy o niej w rozdziale 13.

Kolejnym miejscem, w którym wykorzystuje się przypisanie krotek, jest dzielenie sekwencji w pętlach na jej przód i resztę, jak w poniższym kodzie.

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, L = L[0], L[1:]           # Wersja dla 3.0 przedstawiona w następnym punkcie
...     print front, L
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Przypisanie krotki w pętli mogłoby również zostać zakodowane w poniższych dwóch wierszach, jednak na ogół wygodniej jest połączyć je ze sobą.

```
...     front = L[0]
...     L = L[1:]
```

Warto zwrócić uwagę na to, że ten kod wykorzystuje listę jako swoistą strukturę danych stosu — coś, co często uzyskujemy za pomocą metod `append` i `pop` obiektu listy. Tutaj `front = L.pop(0)` miałoby mniej więcej ten sam efekt co instrukcja z przypisaniem krotki, jednak byłaby to modyfikacja w miejscu. W rozdziale 13. dowiemy się więcej na temat pętli `while` i innych (często lepszych) sposobów przechodzenia sekwencji za pomocą pętli `for`.

Rozszerzona składnia rozpakowania sekwencji w 3.0

W poprzednim punkcie zademonstrowałem, w jaki sposób używać wycinania fragmentów sekwencji, aby uogólnić operacje ich przypisania. W Pythonie 3.0 (ale nie w 2.6) operacje przypisania sekwencji zostały uogólnione, co upraszcza to zadanie. W skrócie: w sekwencji celu przypisania można użyć pojedynczej *nazwy z gwiazdką* (`*x`), której zostanie przypisana lista elementów nieprzypisanych innym zmiennym. Taka możliwość przydaje się szczególnie w przypisaniach typu „pierwsze X elementów i reszta”, jak w ostatnim przykładzie poprzedniego punktu.

Rozszerzona składnia rozpakowania w działaniu

Rzućmy okiem na przykład. Jak widzieliśmy, przypisania sekwencji wymagają z reguły tej samej liczby zmiennych po lewej stronie przypisania, ile elementów zawiera sekwencja z prawej strony. Jeśli liczby elementów nie są zgodne, przypisanie zakończy się błędem, co widzieliśmy również w poprzednim punkcie.

```
C:\misc> c:\python30\python
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4
>>> a, b = seq
ValueError: too many values to unpack
```

W Pythonie 3.0 jedna z nazw po lewej stronie przypisania może być poprzedzona gwiazdką, co daje dodatkowe możliwości. Poniższy listing stanowi kontynuację rozpoczętej wyżej sesji interaktywnej, czyli rozpakowujemy tę samą sekwencję. Zmiennej `a` przypisujemy pierwszy element sekwencji, zmiennej `b` listę pozostałych elementów:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

W przypadku użycia nazwy z gwiazdką liczba zmiennych po lewej stronie przypisania nie musi zgadzać się z liczbą elementów sekwencji po prawej. W rzeczywistości nazwa z gwiazdką może wystąpić w dowolnym miejscu sekwencji. Na przykład w następnej iteracji zmiennej `b` przypisujemy ostatni element sekwencji, a zmiennej `a` wszystkie jej elementy oprócz ostatniego:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

Jeśli nazwa z gwiazdką wystąpi w środku, zostaną jej przypisane wszystkie elementy z pominięciem tych, które zostały przypisane zmiennym bez gwiazdki. W poniższym przykładzie a i c otrzymują odpowiednio pierwszy i ostatni element sekwencji, zmiennej b zostaje przypisana kopia sekwencji z pominięciem pierwszego i ostatniego elementu.

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

A więc w przypadku, gdy w przypisaniu sekwencji występuje nazwa z gwiazdką, zostaną jej przypisane wszystkie elementy, które nie zostały przypisane do pozostałych zmiennych.

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Oczywiście podobnie jak zwykłe przypisanie sekwencji rozszerzona składnia rozpakowania sekwencji działa z dowolnymi typami sekwencyjnymi, nie tylko z listami. Poniższy listing prezentuje przykład rozpakowania ciągu znaków:

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])

>>> a, *b, c = 'spam'
>>> a, b, c
('s', ['p', 'a'], 'm')
```

Reguła ta przypomina tworzenie wycinków, ale to nie jest dokładnie to samo: przypisanie rozpakowujące sekwencję zawsze zwraca listę, natomiast tworzenie wycinków zwraca obiekt tego samego typu, co obiekt poddany operacji wycinania.

```
>>> S = 'spam'

>>> S[0], S[1:] # Wycinki są zależne od typu, przypisanie * zawsze zwraca listę
('s', 'pam')

>>> S[0], S[1:3], S[3]
('s', 'pa', 'm')
```

Dzięki temu rozszerzeniu Pythona 3.0 ostatni przykład poprzedniego punktu staje się jeszcze prostszy, ponieważ nie musimy ręcznie przypisywać wycinka, aby uzyskać listę nieprzypisanych elementów sekwencji.

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, *L = L           # Pobieramy pierwszy element i listę pozostałych, bez wycinania
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Przypadki brzegowe

Składnia rozszerzonego rozpakowania sekwencji jest elastycznym mechanizmem, ale warto pamiętać o szczególnych przypadkach. Po pierwsze, nazwa z gwiazdką może otrzymać pojedynczy element, ale zawsze będzie to lista:

```
>>> seq
[1, 2, 3, 4]

>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]
```

Po drugie, w sytuacji, gdy nic nie zostanie, nazwie z gwiazdką zostaje przypisana pusta lista, niezależnie od tego, w którym miejscu listy zmiennych występuje. W poniższym przykładzie zmiennym `a`, `b`, i `c` przypisano wszystkie elementy sekwencji, ale Python przypisuje pustą listę zmiennej `e` i nie wywołuje błędu.

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

Błędy mogą jednak wystąpić, jeśli w przypisaniu występuje większa liczba nazw z gwiazdką, albo jeśli wystąpi za duża liczba zmiennych bez gwiazdki (czyli analogicznie do zwykłego przypisania sekwencji) oraz w przypadku, gdy nazwa z gwiazdką zostanie użyta poza wyrażeniem przypisania.

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

Wygodny gadżet

Należy pamiętać, że rozszerzona składnia rozpakowania sekwencji służy jedynie zwiększeniu wygody. Można sobie bez niej poradzić (a w 2.X trzeba), wykorzystując odczyt po indeksie i tworzenie wycinków, ale składnia rozpakowująca jest łatwiejsza w użyciu. Opisany wzorzec „pierwszy element i reszta” jest dość powszechny w programowaniu i można go zaimplementować na wiele sposobów, ale użycie wycinków wymaga dodatkowych wierszy kodu:

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq
>>> a, b
# Pierwszy, pozostałe
(1, [2, 3, 4])
```

```
>>> a, b = seq[0], seq[1:] # Pierwszy, pozostałe: metoda tradycyjna
>>> a, b
(1, [2, 3, 4])
```

Równie powszechny wzorzec „początek, ostatni element” implementuje się podobnie, ale ponownie składania rozpakowująca wymaga mniejszej ilości kodu:

```
>>> *a, b = seq # Początek, ostatni
>>> a, b
([1, 2, 3], 4)

>>> a, b = seq[:-1], seq[-1] # Początek, ostatni: metoda tradycyjna
>>> a, b
([1, 2, 3], 4)
```

Użycie rozszerzonej składni rozpakowania sekwencji jest prostsze i wydaje się bardziej naturalne, co wróży tej nowości w Pythonie zdobycie z czasem dużej popularności.

Zastosowanie w pętli for

Zmienna użyta w pętli `for` może być określona z użyciem dowolnego przypisania, zatem i w tym przypadku działa przypisanie sekwencji. W części drugiej spotkaliśmy się już z iteracjami w pętli `for`, ale pełnej ich analizie poświęcimy rozdział 13. W Pythonie 3.0 składania rozszerzonego przypisania może wystąpić po słowie kluczowym `for`, gdzie najczęściej stosuje się zwykłe przypisanie do pojedynczej zmiennej:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    ...
```

W tym kontekście w każdej iteracji Python przypisuje zmiennym `a`, `b` i `c` kolejną krotkę wartości. Na przykład w pierwszym przebiegu pętli zmienne te otrzymują takie wartości, jak w wyniku następującego wywołania:

```
a, *b, c = (1, 2, 3, 4) # b ma wartość [2, 3]
```

Nazwy `a`, `b` i `c` mogą być użyte w kodzie pętli, udostępniając rozpakowane elementy krotki. W rzeczywistości takie użycie nie jest specjalnym przypadkiem, ponieważ stanowi jeden z przypadków wyrażeń przypisania. Jak widzieliśmy wcześniej w tym rozdziale, możemy dokonać podobnego przypisania do krotek, które działa w Pythonie 2.X i 3.X:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: # a, b, c = (1, 2, 3), ...
```

Oczywiście i w tym kontekście w Pythonie 2.6 możemy zaemulować rozszerzoną składnię rozpakowania sekwencji dostępną w 3.0, używając wycinków:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:
    a, b, c = all[0], all[1:3], all[3]
```

Na razie nie posiadamy wystarczającej wiedzy na temat tego typu składni w pętlach `for`, zatem odłożymy omawianie tego tematu do rozdziału 13.

Przypisanie z wieloma celami

Przypisanie z wieloma celami po prostu przypisuje wszystkie podane zmienne do obiektu znajdującego się najbardziej po prawej stronie. Poniższy kod przypisuje na przykład trzy zmienne (`a`, `b` oraz `c`) do łańcucha znaków 'mielonka'.

```
>>> a = b = c = 'mielonka'  
>>> a, b, c  
('mielonka', 'mielonka', 'mielonka')
```

Forma ta jest (łatwiejszym do zapisania) odpowiednikiem poniższych trzech instrukcji przypisania.

```
>>> c = 'mielonka'  
>>> b = c  
>>> a = b
```

Przypisanie z wieloma celami a współdzielone referencje

Należy pamiętać o tym, że ciągle mamy tu jeden obiekt — współdzielony przez wszystkie trzy zmienne (wszystkie one wskazują na ten sam obiekt w pamięci). Takie zachowanie jest poprawne w przypadku typów niezmiennych — na przykład inicjalizacji zbioru liczników do wartości zero (przypomnijmy, że przed użyciem zmienne muszą zostać w Pythonie przypisane, dlatego przed dodaniem czegoś do liczników trzeba je najpierw zainicjalizować do wartości zero).

```
>>> a = b = 0  
>>> b = b + 1  
>>> a, b  
(0, 1)
```

Tutaj zmiana `b` modyfikuje jedynie `b`, ponieważ liczby nie obsługują modyfikacji w miejscu. Dopóki przypisywany obiekt jest niezmienny, to, że większa liczba zmiennych zawiera referencje do niego, nie ma większego znaczenia.

Jak zawsze jednak musimy być ostrożni, kiedy inicjalizuje się zmienne do pustych obiektów zmiennych, takich jak lista czy słownik.

```
>>> a = b = []  
>>> b.append(42)  
>>> a, b  
([42], [42])
```

Tym razem, ponieważ zmienne `a` i `b` zawierają referencje do tego samego obiektu, dodanie do niego jakiegoś elementu w miejscu za pośrednictwem zmiennej `b` będzie miało również wpływ na to, co zobaczymy w `a`. To tak naprawdę kolejny przykład zjawiska współdzielonych referencji, z którym pierwszy raz spotkaliśmy się w rozdziale 6. By uniknąć tego problemu, należy zamiast tego inicjalizować obiekty zmienne w osobnych instrukcjach, wykonując oddzielne wyrażenie z literałem, tak by każdy z nich utworzył osobny pusty obiekt.

```
>>> a = []  
>>> b = []  
>>> b.append(42)  
>>> a, b  
([], [42])
```

Przypisania rozszerzone

Od Pythona 2.0 udostępniony został zbiór dodatkowych formatów instrukcji przypisania wymieniony w tabeli 11.2. Znane jako *przypisania rozszerzone* i zapożyczone z języka C, formaty te są generalnie skrótami. Są one kombinacją wyrażenia binarnego i przypisania. Poniższe dwa formaty są na przykład mniej więcej równoważne.

<code>X = X + Y</code>	# Forma tradycyjna
<code>X += Y</code>	# Nowsza forma rozszerzona

Tabela 11.2. Rozszerzone instrukcje przypisania

X += Y	X &= Y	X -= Y	X = Y
X *= Y	X ^= Y	X /= Y	X >= Y
X %= Y	X <= Y	X **= Y	X //= Y

Przypisania rozszerzone działają na dowolnym typie obiektów obsługujących wyrażenia binarne. Poniżej widać dwa sposoby dodania liczby 1 do zmiennej.

```
>>> x = 1
>>> x = x + 1
>>> x
2

>>> x += 1
# Forma rozszerzona
>>> x
3
```

Po zastosowaniu do łańcuchów znaków forma rozszerzona wykonuje zamiast tego konkatenację. Z tego powodu drugi wiersz poniższego kodu jest odpowiednikiem wpisania dłuższego S = S + "MIELONKA".

```
>>> S = "mielonka"
>>> S += "MIELONKA"
>>> S
'mielonkaMIELONKA'
```

Jak widać w tabeli 11.2, istnieją analogiczne formy rozszerzonego przypisania dla każdego operatora wyrażenia binarnego Pythona (czyli każdego operatora z wartościami po lewej i prawej stronie). Przykładowo X *= Y mnoży i przypisuje, a X >= Y przesuwa w prawo i przypisuje. Instrukcja X //= Y (dzielenie bez reszty) została dodana w Pythonie 2.2.

Przypisania rozszerzone mają trzy zalety:¹

- Mamy mniej kodu do wpisania. Czy muszę dodawać coś więcej?
- Lewa strona musi być obliczona tylko raz. W X += Y zmienna X może być skomplikowanym wyrażeniem obiektu. W formie rozszerzonej wyrażenie to musi być obliczone tylko raz. W dłuższej formie (X = X + Y) zmienna X pojawia się dwa razy i musi też być dwa razy wykonana. Z tego powodu przypisania rozszerzone są zazwyczaj szybsze.
- Optymalna technika wybierana jest automatycznie. W przypadku obiektów obsługujących modyfikację w miejscu formy rozszerzone automatycznie wykonują operacje modyfikacji w miejscu zamiast wolniejszych kopii.

Ostatni z powyższych punktów zasługuje na słowo wyjaśnienia. W przypadku przypisania rozszerzonego w obiektach zmiennych w celu optymalizacji mogą być zastosowane operacje modyfikujące obiekt w miejscu. Warto sobie przypomnieć, że listy mogą być rozszerzane na kilka sposobów. By dodać pojedynczy element na końcu listy, możemy skorzystać z konkatenacji lub metody append.

```
>>> L = [1, 2]
>>> L = L + [3]
# Konkatenacja: wolniej
>>> L
```

¹ Uwaga dla programistów języków C i C++: choć Python obsługuje teraz instrukcje takie, jak X += Y, nadal nie ma operatorów automatycznej inkrementacji i dekrementacji z języka C (czyli X++ i --X). Nie do końca odpowiadają one modelowi obiektów Pythona, ponieważ Python nie pozwala na modyfikacje w miejscu obiektów niezmiennych, takich jak liczby.

```
[1, 2, 3]
>>> L.append(4)                                     # Szybciej, ale w miejscu
>>> L
[1, 2, 3, 4]
```

By dodać zbiór elementów na końcu, możemy albo znowu skorzystać z konkatenacji, albo wywołać metodę listy extend.²

```
>>> L = L + [5, 6]                                # Konkatenacja: wolniej
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])                             # Szybciej, ale w miejscu
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

W obu przypadkach konkatenacja jest mniej podatna na efekty uboczne współdzielonych referencji do obiektu, jednak zazwyczaj będzie działała wolniej od swojego odpowiednika wykonyującego modyfikację w miejscu. Operacje konkatenacji muszą utworzyć nowy obiekt, skopiować listę z lewej strony, a następnie skopiować listę z prawej strony. Modyfikacje w miejscu za pomocą wywołania metod po prostu dodają elementy na końcu bloku pamięci.

Kiedy do rozszerzenia listy wykorzystamy rozszerzoną formę przypisania, możemy zapomnieć o tych szczegółach. Python automatycznie wywołuje na przykład szybszą metodę extend, zamiast korzystać z wolniejszej operacji konkatenacji — co sugerowałoby użycie operatora +.

```
>>> L += [9, 10]                                    # Odwzorowane na L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Przypisania rozszerzone a współdzielone referencje

Zazwyczaj takiego zachowania właśnie sobie życzymy, jednak warto zauważyc, że sugeruje ono, iż += jest modyfikacją listy w miejscu. Z tego powodu nie do końca przypomina konkatenację z operatorem +, która zawsze tworzy nowy obiekt. Jeśli chodzi o referencje współdzielone, różnica ta może mieć znaczenie, gdy różne zmienne wskazują na modyfikowany obiekt.

```
>>> L = [1, 2]
>>> M = L                                         # L i M są referencjami do jednego obiektu
>>> L = L + [3, 4]                               # Konkatenacja tworzy nowy obiekt
>>> L, M                                         # Zmiana L, ale nie M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]                                 # Ale += tak naprawdę oznacza extend
>>> L, M                                         # Również w M zmiana jest widoczna!
([1, 2, 3, 4], [1, 2, 3, 4])
```

Ma to znaczenie jedynie w przypadku obiektów zmiennych, takich jak listy i słowniki, i jest stosunkowo egzotycznym przypadkiem (dopóki, oczywiście, nie zacznie nam przysparzać problemów w kodzie). Jak zawsze, kiedy musimy złamać strukturę referencji współdzielonych, należy wykonać kopie obiektów zmiennych.

² Zgodnie z sugestią z rozdziału 6., można również wykorzystać przypisanie do wycinków (na przykład `L[:len(L)]:= [11, 12, 13]`), jednak działa to mniej więcej tak samo jak prostsza metoda listy extend.

Reguły dotyczące nazw zmiennych

Skoro omówiliśmy już instrukcje przypisania, czas zająć się nazwami zmiennych z bardziej formalnego punktu widzenia. W Pythonie zmienne pojawiają się, kiedy przypisuje się do nich wartości, jednak istnieje kilka reguł dotyczących wyboru nazw dla komponentów naszego programu.

Składnia: (znak _ lub litera) + (dowolna liczba liter, cyfr i znaków _)

Nazwy zmiennych muszą rozpoczynać się od liter lub znaków _, po których może następować dowolna liczba liter, cyfr lub znaków _. W Pythonie poprawnymi nazwami są _mielonka, mielonka i Mielonka_1, natomiast 1_Mielonka, mielonka\$ i @#! są niepoprawne.

Wielkość liter ma znaczenie — MIELONKA to nie to samo co mielonka

W programach Python zawsze zwraca uwagę na wielkość liter — zarówno w tworzonych nazwach, jak i w słowach zarezerwowanych. Nazwy X i x odnoszą się do dwóch różnych zmiennych. W przypadku przenośności wielkość liter ma znaczenie również w nazwach importowanych plików modułów nawet na platformach, w których systemach plików wielkość liter nie ma znaczenia.

Słowa zarezerwowane nie mogą być stosowane

Definiowane nazwy nie mogą być takie same jak słowa zarezerwowane mające w Pythonie specjalne znaczenie. Jeśli na przykład spróbujemy użyć nazwy zmiennej takiej, jak class, Python zwróci błąd składni; klass i Class będą poprawne. W tabeli 11.3 wymieniono aktualne słowa zarezerwowane Pythona.

Tabela 11.3. Słowa zarezerwowane Pythona 3.0

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Tabela 11.3 zawiera słowa kluczowe Pythona 3.0. W Pythonie 2.6 zestaw zarezerwowanych słów kluczowych odróżni się różni:

- print jest zarezerwowanym słowem, ponieważ jest instrukcją, nie funkcją wbudowaną (więcej na ten temat w dalszej części niniejszego rozdziału).
- exec jest zarezerwowanym słowem, ponieważ jest instrukcją, a nie funkcją wbudowaną.
- nonlocal nie jest zarezerwowanym słowem, ponieważ ta instrukcja nie jest dostępna.

W starszych wersjach Pythona jest podobnie, z kilkoma różnicami:

- with oraz as nie były zarezerwowane do wersji 2.6, kiedy wprowadzono mechanizm menedżera kontekstu.
- yield nie było nazwą zarezerwowaną do wersji 2.3, kiedy wprowadzono funkcje generatorów.

- `yield` w wersji 2.5 z instrukcji przekształcono w wyrażenie, ale wciąż jest słowem zarezerwowanym, a nie wbudowaną funkcją.

Jak łatwo zauważyc, słowa zarezerwowane Pythona pisane są małymi literami i naprawdę są one zarezerwowane. W przeciwieństwie do nazw z wbudowanego zakresu, z którymi spotkamy się w kolejnej części książki, nie możemy redefiniować słów zarezerwowanych za pomocą przypisania (na przykład kod `and = 1` spowoduje zwrócenie błędu składni).³

Pierwsze trzy słowa kluczowe przedstawione w tabeli 11.3 są szczególnie nie tylko dlatego, że rozpoczynają się wielką literą. Są one bowiem elementami wbudowanego zakresu nazw Pythona, co jest szczegółowo opisane w rozdziale 17., i z technicznego punktu widzenia są nazwami związanymi z konkretnymi obiektami. Są jednak nazwami zarezerwowanymi i nie można ich użyć w innym kontekście niż przewidziany dla związań z nimi obiektów. Pozostałe słowa kluczowe są natomiast elementami składni języka Python i mogą wystąpić tylko w kontekstach instrukcji, do których zostały przeznaczone.

Co więcej, ponieważ nazwy modułów w instrukcjach `import` stają się zmiennymi w skrypcie, ograniczenie to rozciąga się również na *nazwy plików modułów* — możemy utworzyć pliki o nazwach `and.py` i `my-code.py`, jednak nie możemy ich zimportować. Ponieważ ich nazwy bez rozszerzenia `.py` stają się *zmiennymi* w kodzie, muszą być zgodne z regułami przedstawionymi dla zmiennych (słowa zarezerwowane nie mogą zostać użyte, a myślniki nie działają — zamiast nich można jednak użyć znaków `_`). Powrócimy do tej kwestii w piątej części książki.

Konwencje dotyczące nazewnictwa

Oprócz powyższych reguł istnieje również zbiór *konwencji* dotyczących nazewnictwa — reguł, które nie są obowiązkowe, ale zazwyczaj się ich przestrzega. Na przykład ponieważ nazwy z dwoma znakami `_` na początku i końcu (jak `__name__`) zazwyczaj mają dla interpretera Pythona specjalne znaczenie, powinniśmy unikać tego wzorca we własnych nazwach zmiennych. Poniżej znajduje się lista konwencji dotyczących nazw zmiennych używanych w Pythonie.

- Nazwy rozpoczynające się od jednego znaku `_` (jak `_X`) nie są importowane za pomocą instrukcji `from moduł import *` (opisano to w rozdziale 22.).
- Nazwy z dwoma początkowymi i końcowymi znakami `_` (jak `__X__`) są nazwami zdefiniowanymi przez system, które mają specjalne znaczenie dla interpretera.
- Nazwy rozpoczynające się od dwóch znaków `_` i niekończące się dwoma kolejnymi takimi znakami (jak `__X`) są lokalne dla zawierających je klas (opisano to w rozdziale 30.).
- Nazwa będąca pojedynczym znakiem `_` zachowuje w sesji interaktywnej wynik ostatniego wyrażenia.

Oprócz powyższych konwencji interpretera Pythona istnieją różne inne konwencje, których zazwyczaj przestrzegają programiści tego języka. W dalszej części książki zobaczymy na przykład, że nazwy klas najczęściej zaczynają się od wielkiej litery, a także przekonamy się, że nazwa `self` — choć nie jest zarezerwowana — zazwyczaj pełni w klasach pewną specjalną rolę. W rozdziale 17. zajmiemy się również inną, większą kategorią nazw zwanych *wbudowanymi*, które także są z góry zdefiniowane, jednak nie zarezerwowane (dlatego można je przypisać ponownie: `open = 42` zadziała, choć czasami wolelibyśmy, żeby tak nie było!).

³ W implementacji Jython opartej na Javie nazwy zmiennych zdefiniowanych przez użytkownika mogą czasami być takie same jak zarezerwowane słowa Pythona. Opis systemu Jython znajduje się w rozdziale 2.

Protokół przedawnienia w Pythonie

Interesująca jest obserwacja procesu stopniowego wprowadzania zarezerwowanych słów kluczowych do języka. Gdy nowa funkcja może popsuć istniejący kod, w Pythonie z reguły zmianę wprowadza się stopniowo z zastosowaniem mechanizmu „przedawnienia” (*deprecation*), czyli wyświetlania ostrzeżeń w przypadku użycia takich konstrukcji, które mają być wycofane z języka. Mechanizm przedawnienia opiera się na założeniu, że programista powinien mieć czas na zauważenie ostrzeżeń i aktualizację kodu przed dokonaniem migracji do nowego wydania Pythona. W przypadku dużych wydań, jak 3.0, mechanizm ten nie jest stosowany, przez co ryzyko niekompatybilności jest niemałe, ale jest stosowany od wielu lat w przypadku zwykłych aktualizacji wersji.

Na przykład `yield`: w Pythonie 2.2 był opcjonalnym rozszerzeniem, a w 2.3 stał się standardowym słowem kluczowym. To słowo kluczowe jest stosowane w funkcjach generatorów. Jest to jeden z licznych przykładów, gdy Python psuł zgodność ze starym kodem. Jednak `yield` był wprowadzany do Pythona stopniowo: w 2.2 użycie go jako nazwy zmiennej generowało ostrzeżenia o przedawnieniu, a w 2.3 `yield` przestał być akceptowany jako nazwa i został wprowadzony jako słowo kluczowe.

Podobnie w Pythonie 2.6 nowymi słowami kluczowymi stały się `with` i `as` używane przez menedżera kontekstu (służące między innymi do nowej formy obsługi wyjątków). Te dwa słowa kluczowe do wersji 2.5 nie były zarezerwowane, chyba że programista włączył obsługę menedżera kontekstów za pomocą instrukcji `from __future__ import` (co zostało omówione w dalszej części książki). W przypadku użycia `with` i `as` jako nazw zmiennych w 2.5 Python generuje ostrzeżenia o zbliżającej się zmianie. Wyjątkiem jest tu IDLE dla Pythona 2.5, w którym autorzy włączyli użycie tej funkcji (co w IDLE powoduje wyświetlenie błędu użycia słowa kluczowego, zamiast ostrzeżenia o przedawnieniu).

Nazwy nie mają typu, typ mają obiekty

To przede wszystkim przypomnienie, jednak pamiętanie o rozróżnieniu nazw i obiektów Pythona jest kwestią kluczową. Jak pisaliśmy w rozdziale 6., obiekty mają typ (na przykład liczby całkowitej czy listy) i mogą być zmienne lub niezmienne. Nazwy (inaczej zmienne) są zawsze jedynie referencjami do obiektów. Nie jest z nimi powiązana koncepcja zmienności i niezmienności ani informacja o typie — poza typem obiektu, do którego w określonym momencie są referencją.

Można przypisać tę samą nazwę do różnych rodzajów obiektów w różnym czasie.

```
>>> x = 0                                # x powiązane z obiektem liczby całkowitej
>>> x = "Troll"                            # Teraz jest łańcuchem znaków
>>> x = [1, 2, 3]                          # A teraz lista
```

W późniejszych przykładach zobaczymy, że uniwersalna natura zmiennych może być dużą zaletą programowania w Pythonie. W rozdziale 17. książki dowiemy się, że zmienne istnieją w czymś o nazwie *zakres*, co definiuje miejsca, w których mogą one być użyte. Miejsce, w którym przypisujemy zmienną, określa to, gdzie będzie ona widoczna.⁴

⁴ Osoby używające w przeszłości języka C++ zainteresować może to, że w Pythonie nie istnieje koncepcja deklaracji `const` z tego języka. Pewne obiekty mogą być *niezmienne*, jednak nazwy można przypisywać zawsze. Python potrafi również ukrywać nazwy w klasach i modułach, jednak nie jest to to samo co deklaracja z C++. (jeśli ukrywanie atrybutów jest dla Czytelnika istotne powinien zajrzeć do rozdziału 24 w temacie nazw typu `_x` w modułach, rozdziału 30 w temacie nazw typu `_x` w klasach oraz rozdziału 38 w temacie dekoratorów prywatnych i publicznych atrybutów klas).



Dodatkowe sugestie dotyczące nazewnictwa, oprócz przekazanych w punkcie „Konwencje dotyczące nazewnictwa”, można znaleźć w półoficjalnych zaleceniach dotyczących stylu programowania, znanych jako PEP 8. Ten dokument jest dostępny pod adresem <http://www.python.org/dev/peps/pep-0008>, warto też przeszukać Internet pod kątem wyrażenia „Python PEP 8”. Z technicznego punktu widzenia ten dokument formalizuje standardy tworzenia kodu biblioteki Pythona.

Sformalizowane standardy kodowania są bardzo użyteczne, ale warto traktować je z rezerwą. Po pierwsze, PEP 8 zawiera więcej szczegółów, niż większość Czytelników jest w stanie przyswoić sobie na tym etapie studiowania niniejszej książki. Szczerze mówiąc, reguły zdefiniowane w PEP 8 są zbyt skomplikowane, sztywne i subiektywne, niż można by oczekwać od dokumentu tego typu. Niektóre sugestie w nim zawarte są wręcz powszechnie kwestionowane lub wręcz ignorowane przez programistów Pythona. Co więcej, wiele firm używających Pythona zaadaptowało własne standardy, które różnią się z PEP 8 w wielu kwestiach.

PEP 8 jednak stanowi ważny punkt odniesienia w zakresie standardu tworzenia kodu w Pythonie i jest doskonałą lekturą dla początkujących programistów Pythona, jednak należy traktować go jako zbiór zaleceń, nie prawd objawionych.

Instrukcje wyrażeń

W Pythonie można również użyć wyrażenia jako instrukcji (to znaczy w osobnym wierszu). Ponieważ jednak wynik wyrażenia nie zostanie zapisany, ma to sens jedynie wtedy, gdy efekt uboczny działania wyrażenia będzie przydatny. Wyrażenia są często używane w charakterze instrukcji w dwóch sytuacjach.

W wywołaniach funkcji i metod

Niektóre funkcje oraz metody wykonują dużo pracy bez zwracania wartości. Takie funkcje czasami nazywane są w innych językach *procedurami*. Ponieważ nie zwracają wartości, które moglibyśmy chcieć zachować, możemy je wywoływać za pomocą instrukcji wyrażeń.

Do wyświetlania wartości w sesji interaktywnej

Python zwraca wyniki wyrażenia wpisanego w interaktywnym wierszu poleceń. Z technicznego punktu widzenia są one również instrukcjami wyrażeń — służą jako skrót następujący wpisywanie instrukcji print.

W tabeli 11.4 wymieniono często spotykane formy instrukcji wyrażeń w Pythonie. Wywołania funkcji oraz metod kodowane są z zerem lub większą liczbą argumentów (a tak naprawdę wyrażeń zwracających obiekty) w nawiasach, po nazwie funkcji lub metody.

Tabela 11.4. Często wykorzystywane instrukcje wyrażeń Pythona

Operacja	Interpretacja
spam(eggs, ham)	Wywołanie funkcji
spam.ham(eggs)	Wywołanie metody
spam	Wyświetlanie zmiennych w interpreterze interaktywnym
print(a, b, c, sep='')	Funkcja wypisywania ciągów znaków w Pythonie 3.0
yield x ** 2	Zwracanie wyniku częściowego

Dwa ostatnie wiersze w tabeli 11.4 mają specjalną formę; jak przekonamy się w dalszej części niniejszego rozdziału, wyświetlanie tekstów w Pythonie 3.0 jest realizowane przez funkcję, natomiast wyrażenie `yield` w funkcjach generatorów (o tym dowiemy się więcej w rozdziale 20.) jest często zapisywane jak instrukcja. Jedno i drugie jest natomiast specyficzną formą instrukcji wyrażeń.

Wywołanie funkcji `print` jest z reguły wywoływanie w osobnym wierszu kodu bez przypisania wyniku, choć funkcja `print` zwraca wynik jak każda inna (a dokładniej: zwraca `None`, co jest zupełnie normalne w przypadku funkcji, których zadanie nie polega na zwracaniu wyniku).

```
>>> x = print('spam')      # print jest w 3.0 wyrażeniem wywołania funkcji  
spam  
>>> print(x)           # najczęściej jest jednak stosowana jako instrukcja wyrażenia  
None
```

Należy jednak być świadomym, że choć wyrażenia mogą się w Pythonie pojawić w postaci instrukcji, instrukcje nie mogą być używane jako wyrażenia. Python nie pozwala na przykład osadzać instrukcji przypisania (`=`) w innych wyrażeniach. Uzasadnienie jest takie, że pozwala to uniknąć błędów w kodowaniu. Nie możemy przypadkowo zmodyfikować zmiennej, wpisując `=`, kiedy tak naprawdę chcielibyśmy użyć testu równości `==`. Zobaczmy, jak obejść to ograniczenie, kiedy w rozdziale 13. spotkamy się z pętlą `while`.

Instrukcje wyrażeń i modyfikacje w miejscu

W ten sposób dochodzimy do błędu często popełnianego w pracy z Pythonem. Instrukcje wyrażeń są często wykorzystywane w celu wykonywania metod list modyfikujących te listy w miejscu.

```
>>> L = [1, 2]                # append jest metodą modyfikującą listę w miejscu  
>>> L.append(3)  
>>> L  
[1, 2, 3]
```

Osoby rozpoczynające swoją znajomość z Pythonem często kodują taką operację jako instrukcję przypisania, zamierzając przypisać `L` do większej listy.

```
>>> L = L.append(4)          # append zwraca None, a nie L  
>>> print L                # Tracimy więc naszą listę!  
None
```

To jednak nie działa — wywołanie operacji modyfikującej listę w miejscu, takiej jak `append`, `sort` czy `reverse`, zawsze zmienia tę listę w miejscu, jednak metody te nie zwracają samej zmodyfikowanej listy. Tak naprawdę zwracają one obiekt `None`. Jeśli przypiszemy wynik takiej operacji z powrotem do zmiennej, w efekcie całkowicie stracimy listę (i najprawdopodobniej zostaje ona w międzyczasie wyczyszczona z pamięci).

Moral z tej historii jest taki, że nie należy tego robić. Z tym zjawiskiem spotkamy się raz jeszcze w części z ostrzeżeniami „Często spotykane problemy programistyczne” znajdującej się na końcu tej części książki, ponieważ może się ono pojawić również w kontekście niektórych instrukcji pętli omawianych w kolejnych rozdziałach.

Polecenia print

Polecenia `print` wyświetlają (drukują) różne rzeczy — to po prostu przyjazny programiście interfejs do standardowego strumienia wyjścia.

Z technicznego punktu widzenia można powiedzieć, że przekształca on obiekt na jego reprezentację tekstową, dodaje formatowanie i przesyła do standardowego wyjścia. Nieco bardziej szczegółowo — `print` jest silnie związany z plikami i strumieniami wyjścia w Pythonie:

Metody plikowe

W rozdziale 9. można się było zapoznać z informacjami na temat obiektów plikowych służących do zapisu tekstu (`file.write(str)`). Wyświetlanie informacji działa podobnie, ale w sposób bardziej specjalizowany: metody zapisu do plików pozwalają zapisywać dane w dowolnych plikach, natomiast funkcja `print` wypisuje teksty na standardowym wyjściu z możliwością określenia formatowania. W przeciwieństwie do operacji plikowych, przy użyciu funkcji `print` nie ma konieczności przekształcania obiektów na ciągi znaków.

Standardowy strumień wyjściowy

Standardowy strumień wyjściowy (znany jako `stdout`) jest podstawowym mechanizmem systemowym do wypisywania wyników przez programy. Wraz ze standardowym strumieniem wejścia oraz strumieniem błędów jest jednym z trzech kanałów komunikacyjnych tworzonych podczas uruchamiania skryptu. Standardowy strumień wyjściowy jest z reguły mapowany na ekran (terminal), z którego został uruchomiony program, chyba że w poleceniu wywołania zostanie przekierowany do potoku lub pliku.

Standardowy strumień wyjściowy jest dostępny w Pythonie jako obiekt `stdout` wchodzący w skład modułu `sys` biblioteki standardowej (`sys.stdout`). Istnieje możliwość zaemulowania funkcji `print` z użyciem standardowych mechanizmów plikowych. Jednak funkcja `print` jest znacznie prostsza w użyciu, jak również pozwala zapisywać teksty w plikach i innych strumieniach.

Wyświetlanie tekstów to jedno z zagadnień, w których wystąpiły najwyraźniejsze różnice między Pythonem 3.0 a 2.6. W rzeczywistości ta różnica między wersjami jest z reguły pierwszą przyczyną braku możliwości uruchomienia starego kodu w Pythonie 3.0. Sposób wywołania operacji wyświetlania tekstu rożni się bowiem w różnych wersjach Pythona:

- w Pythonie 3.X do wyświetlania tekstu stosuje się funkcję wbudowaną `print`, obsługującą argumenty ze słowami kluczowymi pozwalające na zastosowanie specjalnych trybów wyświetlania;
- w Pythonie 2.6 wyświetlanie tekstów jest instrukcją wykorzystującą własną, specyficzną składnię.

Ponieważ niniejsza książka opisuje Pythona 2.6 i 3.0, w kolejnych podrozdziałach omówimy obydwa te sposoby wyświetlania. Jeśli ktoś tworzy kod tylko w jednej z omawianych wersji Pythona, może z powodzeniem pominać punkty, które dotyczą nieużywanej wersji. Jednak z pewnością nie zaszkodzi zapoznać się z obydwoma technikami.

Funkcja print Pythona 3.0

Wyświetlanie tekstów w 3.0 nie wykorzystuje instrukcji. Zamiast tego mamy do dyspozycji wyrażenie wykorzystujące *wywołanie funkcji*, wspomniane w poprzednim punkcie.

Wbudowana funkcja `print` jest najczęściej wywoływana w osobnym wierszu kodu bez przypisywania zwracanej wartości (ponieważ `print` zwraca `None`). Z uwagi na to, że w 3.0 `print` jest funkcją, mamy do dyspozycji standardową składnię wywołania funkcji, a nie specjalną formę instrukcji jak w starszych wersjach języka. Funkcja `print` obsługuje różne tryby działania obsługiwane za pomocą argumentów ze słowami kluczowymi, dzięki czemu jej użycie jest bardziej uogólnione i pozwala na proste wprowadzanie nowych mechanizmów w przeszłość.

Stosowana w starszych wersjach Pythona (również w 2.6) instrukcja `print` wykorzystuje specjalną składnię pozwalającą na pominięcie automatycznego przejścia do nowego wiersza lub przekierowanie wyniku do pliku. Co więcej, stara forma w ogóle nie pozwala na określenie separatorów, zamiast tego w wersji 2.6 programista musi zbudować ciągi znaków, które następująco wyświetla. W 3.0 taka konieczność występuje dużo rzadziej.

Format wywołania

Składniowo wywołanie funkcji `print` w Pythonie 3.0 ma następującą formę:

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])
```

W tej formalnej notacji elementy w nawiasach kwadratowych są opcjonalne i można je pominąć w wywoaniu, wartości po znaku = określają domyślne wartości argumentów. W skrócie: ta funkcja wbudowana wypisuje do pliku `file` reprezentację tekstową jednego lub większej liczby obiektów, oddzielając je użyciem separatora `sep` i na końcu wyświetlając ciąg znaków `end`.

Argumenty `sep`, `end` i `file` muszą być podane jako *argumenty ze słowami kluczowymi*, to znaczy w celu ich przekazania musi być użyta specjalna składnia nazwa=wartość. Argumenty ze słowami kluczowymi zostały omówione szczegółowo w rozdziale 18., ale ich użycie jest proste. Argumenty ze słowami kluczowymi przekazywane do tej funkcji mogą mieć dowolną kolejność w wywołaniu, a ich działanie jest następujące:

- `sep` określa ciąg znaków wstawiany pomiędzy wyświetlonymi obiektami, domyślnie to jedna spacja; w celu rezygnacji z separatorów należy przekazać pusty ciąg znaków.
- `end` określa ciąg znaków, jaki będzie wyświetlony na końcu wyświetlonego tekstu, domyślnie jest to sekwencja końca wiersza (`\n`). Przekazanie pustego ciągu znaków spowoduje, że na końcu wyświetlonego tekstu kurSOR nie zostanie przeniesiony do nowego wiersza, czyli następna operacja drukowania rozpocznie się na końcu bieżącego wyświetlania.
- `file` określa plik, strumień standardowy lub inny obiekt typu plikowego, do którego zostanie wysłany wyświetlany tekst. Domyślną wartością tego argumentu jest `sys.stdout`, czyli standardowy strumień wyjściowy. W tym argumencie można przekazać dowolny obiekt plikowy (lub emulujący ten interfejs) obsługujący metodę `write`. W przypadku użycia rzeczywistych plików muszą być one już otwarte do zapisu.

Tekstowa reprezentacja każdego obiektu jest pozyskiwana przez przekazanie go do funkcji standardowej `str`. Jak widzieliśmy wcześniej, ta funkcja zwraca „przyjazną dla użytkownika” reprezentację tekstową każdego obiektu.⁵ W przypadku wywołania funkcji `print` bez argumentów w standardowym strumieniu wyjściowym zostanie wyświetlony pusty wiersz.

⁵ Dokładniej: funkcja `print` wykorzystuje wewnętrzny odpowiednik funkcji `str`, ale efekt jest dokładnie taki sam. Funkcja `str` nie tylko zwraca tekstową reprezentację obiektów, ale też jest stosowana jako funkcja przekształcająca dane do typu tekstopiowego i może być użyta do dekodowania ciągów znaków Unicode z użyciem dodatkowego argumentu `encoding`. Więcej informacji na ten temat podano w rozdziale 36., ale to zaawansowana wiedza i możemy ją zignorować w kontekście niniejszego rozdziału.

Funkcja print w działaniu

Wyświetlanie ciągów znaków w 3.0 jest znacznie prostsze, niż może sugerować nagłówek funkcji print. Przeanalizujmy kilka prostych przykładów. Poniższy listing prezentuje wyświetlanie obiektów różnych typów z domyślnym separatorem i zakończeniem (te wartości są zastosowane jako domyślne, ponieważ są najczęściej stosowane).

```
C:\misc> c:\python30\python
>>>
>>> print()                                     # Wyświetla pustą linię

>>> x = 'mielonka'
>>> y = 99
>>> z = ['jajka']
>>>
>>> print(x, y, z)                             # Wyświetla 3 obiekty z domyślnym formatowaniem
mielonka 99 ['jajka']
```

Nie ma potrzeby przekształcania obiektów na ciągi znaków, co jest konieczne w przypadku zapisu obiektów do pliku. Domyślnie funkcja print rozdziela reprezentacje obiektów pojedynczą spacją. Aby tego uniknąć, należy przekazać pusty ciąg znaków w argumencie sep, można oczywiście podać własny separator.

```
>>> print(x, y, z, sep='')                      # Pominięcie separatora
mielonka99['jajka']
>>>
>>> print(x, y, z, sep=', ')
mielonka, 99, ['jajka']                         # Niestandardowy separator
```

Również domyślnie funkcja print na końcu dodaje znak nowego wiersza. I to zachowanie można zmienić, przekazując w argumencie end pusty ciąg znaków lub inną sekwencję znaków, która ma być wyświetlona na końcu (znak końca wiersza jest reprezentowany przez sekwencję \n).

```
>>> print(x, y, z, end='')                      # Pominięcie przejścia do nowego wiersza
mielonka 99 ['jajka']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)       # Dwukrotne wypisanie tego samego wiersza
mielonka 99 ['jajka']mielonka 99 ['jajka']
>>> print(x, y, z, end='...\n')                  # Niestandardowa sekwencja zakończenia wiersza
mielonka 99 ['jajka']...>>>
```

Można również zastosować różne kombinacje argumentów ze słowami kluczowymi, definiując własne separatory i zakończenia: argumenty mogą występować w dowolnej kolejności po liście wyświetlanych obiektów.

```
>>> print(x, y, z, sep='...', end='!\n')        # Wiele argumentów ze słowami kluczowymi
mielonka...99...['jajka']!
>>> print(x, y, z, end='!\n', sep='...')          # Kolejność nie ma znaczenia
mielonka...99...['jajka']!
```

Poniższy listing prezentuje użycie argumentu file. W przypadku przekazania otwartego obiektu plikowego obiekty zostaną zapisane w tym pliku, zamiast być wyświetcone na ekranie; taka zamiana obowiązuje tylko w tym wywołaniu funkcji print (to jedna z form przekierowania strumienia wyjściowego, tym zagadnieniem zajmiemy się szerzej za chwilę).

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w'))    # Wypisanie do pliku
>>> print(x, y, z)                                              # Powrót do standardowego wyjścia
mielonka 99 ['jajka']
>>> print(open('data.txt').read())                                # Wyświetlenie pliku tekstowego
mielonka...99...['jajka']
```

Należy pamiętać, że argumenty sep i end są dostępne jedynie dla wygody. Jeśli chcemy zastosować bardziej skomplikowane formatowanie, można zupełnie z nich zrezygnować i ręcznie budować ciągi znaków, korzystając z zaawansowanego formatowania i wykorzystując narzędzia poznane w rozdziale 7., a następnie wyświetlać takie gotowe ciągi znaków w jednym wywołaniu funkcji print.

```
>>> text = '%s: %-4f, %05d' % ('Wynik', 3.14159, 42)
>>> print(text)
Wynik: 3.1416, 00042
>>> print('%s: %-4f, %05d' % ('Wynik', 3.14159, 42))
Wynik: 3.1416, 00042
```

Jak się przekonamy w następnym punkcie, prawie wszystko, co wiemy o funkcji print z Pythona 3.0, ma zastosowanie do instrukcji print z Pythona 2.6, co ma sens, biorąc pod uwagę fakt, że funkcja ma za zadanie emulować stary mechanizm, uzupełniając go o nowe możliwości.

Instrukcja print w Pythonie 2.6

Jak wspominałem wcześniej, do wyświetlania ciągów znaków w Pythonie 2.6 wykorzystywana jest instrukcja print z własną, specjalizowaną składnią. W praktyce jednak wyświetlanie w 2.6 jest wariacją poznanego mechanizmu wyświetlania z Pythona 3.0, z pominięciem możliwości definiowania separatorów i zakończenia (które są dostępne w 3.0, ale niedostępne w 2.6). Oprócz tego wszystko, co da się zrobić za pomocą funkcji print w 3.0, można przekształcić w instrukcję print Pythona 2.6.

Formy instrukcji

Tabela 11.5 przedstawia formy wywołania instrukcji print w Pythonie 2.6 oraz ich odpowiedniki w funkcji print Pythona 3.0. Warto zwrócić uwagę na to, że *przecinek* ma znaczenie w instrukcji print: służy do oddzielania wyświetlanych obiektów. Przecinek na końcu instrukcji zapobiega wyświetleniu znaku końca wiersza na końcu (nie należy mylić takiego zapisu ze składnią krotki!). Składnia >> (przesunięcie bitowe w prawo) w kontekście instrukcji print jest używana do przekierowania wyniku do innego strumienia niż standardowy sys.stdout.

Tabela 11.5. Formy wywołania instrukcji print w Pythonie 2.6

Instrukcja w Pythonie 2.6	Odpowiednik w Pythonie 3.0	Interpretacja
print x, y	print(x, y)	Wypisuje na standardowym wyjściu reprezentacje tekstowe obiektów, oddzielając je spacją i umieszczając znak końca wiersza na końcu wyniku.
print x, y, print >> afile, x, y	print(x, y, end='')	To samo, bez znaku końca wiersza.
	print(x, y, file=afile)	Przekierowanie ciągu znaków do pliku zamiast do sys.stdout.

Instrukcja print Pythona 2.6 w działaniu

Choć w 2.6 instrukcja print posiada dziwniejszą składnię od funkcji print w 3.0, jej użycie jest bardzo podobne. Ponownie przeanalizujmy kilka prostych przykładów. Domyślnie instrukcja print w 2.6 dodaje spację między wyświetlonymi obiektami oddzielonymi przecinkami, a na końcu wyświetla znak końca wiersza.

```
C:\misc> c:\python26\python
>>>
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

To formatowanie jest domyślne i można z niego nie skorzystać. Aby uniknąć wyświetlania znaku końca wiersza na końcu wyniku, należy instrukcję zakończyć przecinkiem, jak w drugim wierszu tabeli 11.5 (w poniższym listingu dwie instrukcje `print` zostały umieszczone w jednym wierszu, oddzielone średnikiem).

```
>>> print x, y,; print x, y
a b a b
```

Aby uniknąć spacji między obiektami, nie należy oddzielać obiektów przecinkami. Zamiast tego można wynikowy ciąg znaków zbudować w całości przed wyświetleniem, używając narzędzi formatowania ciągów znaków poznanych w rozdziale 7., po czym wyświetlić utworzony ciąg znaków.

```
>>> print x + y
ab
>>> print '%s...%s' % (x, y)
a...b
```

Jak widzimy — pomijając dziwną składnię — instrukcje `print` Pythona 2.6 są tak proste w użyciu, jak funkcja `print` z Pythona 3.0. Następny punkt wyjaśni, w jaki sposób można wymusić zapis do pliku w Pythonie 2.6, co jest odpowiednikiem argumentu `file` funkcji `print`.

Przekierowanie strumienia wyjściowego

W Pythonie 3.0 i 2.6 użycie funkcji lub instrukcji `print` powoduje wyświetlenie ciągu znaków na ekranie. Często jednak użytkownicy mogą okazać się wysłanie wyniku w inne miejsce: na przykład do pliku tekstowego, w celu przeanalizowania wyników po zakończeniu działania programu. Choć tego typu przekierowanie wyniku do pliku można zrealizować poza Pythonem dzięki narzędziom powłoki systemowej, to czasem konieczne bywa zastosowanie kontroli strumienia wyjściowego bezpośrednio w skrypcie w Pythonie.

Program „witaj świecie”

Na początek znany, powszechnie stosowany do testowania wydajności (i zupełnie bezużyteczny) kod „witaj świecie”. Oto wersje wywołań dla różnych wersji Pythona:

```
>>> print('witaj świecie')                      # Wyświetlenie ciągu znaków w Pythonie 3.0
witaj świecie

>>> print 'witaj świecie'                      # Wyświetlenie ciągu znaków w Pythonie 2.6
witaj świecie
```

Wyniki wyrażeń są wysyłane do wyjścia konsoli interaktywnej, ale w tym przypadku nie ma konieczności używania polecenia `print`, wystarczy wpisać wyrażenie, które chcemy wyświetlić, a jego wartość zostanie wyświetlona:

```
>>> 'hello world'                                # Interaktywne wyświetlanie wartości
'witaj świecie'
```

Powyższy kod raczej nie stanowi zachwycającego przykładu mistrzowskiego programowania, ale dobrze ilustruje zasadę działania mechanizmu wyświetlania. W rzeczywistości operacja `print` jest jedynie ergonomiczną wersją zapisu do pliku z prostym mechanizmem formatującym. Jeśli ktoś lubi zbędne wyzwania, może również zakodować wyświetlanie, właśnie korzystając z operacji plikowych.

```
>>> import sys  
>>> sys.stdout.write('witaj świecie\n')  
witaj świecie
```

Wyświetlanie metodą "plikową"

Powyższy kod wywołuje metodę `write` obiektu plikowego `sys.stdout`, czyli obiektu podłączonego do standardowego strumienia wyjściowego inicjalizowanego podczas uruchamiania interpretera Pythona. Operacja `print` działa w zbliżony sposób, ale ukrywa przed programistą te szczegóły, dając mu proste narzędzie do realizacji prostych zadań związanych z wyświetleniem tekstów.

Ręczne przekierowanie strumienia wyjścia

Po co zatem pokazałem trudniejszy sposób wyświetlania obiektów? Użycie `sys.stdout` okazuje się podstawą pewnej często stosowanej w Pythonie techniki. Generalnie `print` i `sys.stdout` powiązane są ze sobą w następujący sposób. Poniższa instrukcja:

```
print(X, Y) # albo w Pythonie 2.6: print X, Y
```

jest odpowiednikiem dłuższej:

```
import sys  
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

która ręcznie wykonuje konwersję łańcucha znaków za pomocą `str`, dodaje nowy wiersz za pomocą `\n` i wywołuje metodę `write` strumienia wyjścia. Której wersji wolałbyś użyć (piszę to w nadziei zasugerowania piękna prostoty funkcji `print`)?

Oczywiście dłuższa forma nie jest szczególnie użytkowa dla zwykłego wyświetlania. Dobrze jest jednak wiedzieć, że instrukcje `print` robią dokładnie to, ponieważ można *przypisać* `sys.stdout` do czegoś innego od standardowego strumienia wyjścia. Innymi słowy, równoważność tych dwóch rozwiązań umożliwia tworzenie własnych instrukcji `print` przesyłających tekst do innych miejsc, jak w poniższym kodzie.

```
import sys  
sys.stdout = open('log.txt', 'a') # Przekierowuje print do pliku  
...  
print x, y, x # Pokazuje się w log.txt
```

Tutaj zmieniamy `sys.stdout`, tak by ręcznie otworzyć obiekt pliku wyjścia w trybie do dodawania. Po tej zmianie każda instrukcja `print` w dowolnym miejscu programu będzie zapisywała tekst na końcu pliku `log.txt`, a nie w oryginalnym strumieniu wyjścia. Instrukcje `print` z radością wywołują metodę `write` obiektu `sys.stdout` bez względu na to, do czego w danej chwili odnosi się `sys.stdout`. Ponieważ w procesie istnieje tylko jeden moduł `sys`, przypisanie `sys.stdout` w ten sposób przekieruje każdą instrukcję `print` znajdująjącą się w dowolnym miejscu programu.

Tak naprawdę — co zostanie wyjaśnione w ramce dotyczącej `print` i `stdout` — możemy nawet ustawić `sys.stdout` na obiekt niebędący plikiem, o ile obsługuje on oczekiwany protokół (metodę `write`). Kiedy ten obiekt jest *klasą*, wyświetlany tekst można dowolnie przekierować i przetwarzać.

Sztuczka z przestawianiem strumienia wyjścia jest szczególnie użyteczna w programach oryginalnie napisanych z instrukcjami `print`. Jeśli od początku wiemy, że dane wyjściowe powinny trafić do pliku, zawsze możemy wywołać zamiast tego metody zapisu do pliku. By jednak przekierować dane wyjściowe programu opartego na instrukcjach `print`, nie musząc jednocześnie modyfikować każdej z tych instrukcji lub używać składni powłoki systemowej, wygodnie jest skorzystać z możliwości przestawienia `sys.stdout`.

Automatyczne przekierowanie strumienia

Sztuczka z przekierowaniem zapisywanej tekstu za pomocą przypisania `sys.stdout` jest w praktyce używana bardzo często. Jednym z problemów związanych z poprzednim kodem jest jednak to, że nie istnieje bezpośrednia metoda przywrócenia oryginalnego strumienia wyjścia, gdybyśmy musieli przełączyć się z powrotem po zapisaniu tekstu do pliku. Ponieważ `sys.stdout` jest normalnym obiektem pliku, zawsze możemy go jednak zapisać i w razie konieczności przywrócić.⁶

```
C:\misc> c:\python30\python
>>> import sys
>>> temp = sys.stdout
>>> sys.stdout = open('log.txt', 'a')
>>> print('mielonka')
>>> print(1, 2, 3)
>>> sys.stdout.close()
>>> sys.stdout = temp
# Zapisanie na później
# Przekierowanie do pliku
# Wypisuje do pliku, nie na ekran
# Opróżnienie bufora zapisu
# Przywrócenie oryginalnego strumienia

>>> print('znów jestem')
znów jestem
# Wynik znów pojawi się na ekranie
>>> print(open('log.txt').read())
mielonka
# Rezultat wcześniejszego wypisywania
1 2 3
```

Jak pokazano w powyższym przykładzie, ręczne zapisywanie i przywracanie oryginalnego strumienia wyjścia jest dość skomplikowane. Taka sytuacja zdarza się dość często, dlatego w 2.6 zaimplementowano możliwość przekierowania strumienia wyjściowego do pliku.

W 3.0 do tego samego celu służy argument ze słowem kluczowym `file` funkcji `print`. Pozwala on na zapisanie tekstu do wskazanego pliku bez modyfikowania obiektu `sys.stdout`. Dzięki temu, że to przekierowanie jest tymczasowe, inne wywołania funkcji `print` nadal wysyłają teksty na oryginalny strumień wyjściowy. W 2.6 takie samo działanie ma instrukcja `print`, w której zastosuje się operator przekierowania `>>`. Poniższy przykład zapisuje teksty do pliku `log.txt`.

```
log = open('log.txt', 'a')          # 3.0
print(x, y, z, file=log)           # Zapis do obiektu podobnego do pliku
print(a, b, c)                    # Zapis do oryginalnego stdout

log = open('log.txt', 'a')          # 2.6
print >> log, x, y, z             # Zapis do obiektu podobnego do pliku
print a, b, c                      # Zapis do oryginalnego stdout
```

⁶ W obu wersjach Pythona, 2.6 oraz 3.0, można również skorzystać z względnie nowego atrybutu `__stdout__` modułu `sys`, który odnosi się do oryginalnej wartości `sys.stdout` z początku programu. By jednak wrócić do oryginalnej wartości strumienia wyjścia, trzeba będzie przywrócić `sys.stdout` do `sys.__stdout__`. Więcej informacji na ten temat można znaleźć w dokumentacji modułu `sys`.

Tego typu przekierowania są szczególnie użyteczne w sytuacjach, gdy w tym samym kodzie musimy zapisywać dane do pliku i na ekran naprzemiennie. Stosując je, należy pamiętać, aby używać obiektu otwartego pliku (lub obiektu emulującego interfejs plikowy), nie ciągu znaków określającego nazwę pliku. Poniższy listing prezentuje tę technikę w działaniu:

```
C:\misc> c:\python30\python
>>> log = open('log.txt', 'w')
>>> print(1, 2, 3, file=log)           # 2.6: print >> log, 1, 2, 3
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)                  # 2.6: print 7, 8, 9
7 8 9
>>> print(open('log.txt').read())
1 2 3
4 5 6
```

Ta rozszerzona forma instrukcji `print` jest również często wykorzystywana do zapisywania komunikatów o błędach do standardowego strumienia wyjścia błędów, `sys.stderr`. Można albo skorzystać z metod zapisu pliku i ręcznie sformatować dane wyjściowe, albo użyć instrukcji `print` ze składnią przekierowującą.

```
>>> import sys
>>> sys.stderr.write(''Zle!'' * 8 + '\n')
Zle!Zle!Zle!Zle!Zle!Zle!Zle!
>>> print (''Zle!'' * 8, file=sys.stderr,)      # 2.6: print >> sys.stderr, ''Zle!'' * 8
Zle!Zle!Zle!Zle!Zle!Zle!Zle!
```

Po poznaniu tajników wyświetlania znaków powiązanie tej operacji z operacjami plikowymi powinno stać się oczywiste. Poniższy listing prezentuje sesję konsoli interaktywnej w 3.0 i wyświetlanie znaków obydwoma metodami (zwykłą i plikową), po czym wynik zostaje przekierowany do pliku zewnętrznego w celu weryfikacji prawidłowości.

```
>>> X = 1; Y = 2
>>> print(X, Y)                      # Wyświetlanie: metoda standardowa
1 2
>>> import sys
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')          # Wyświetlanie: metoda plikowa
1 2
4
>>> print(X, Y, file=open('temp1', 'w'))                     # Przekierowanie do pliku
>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n')    # Bezpośredni zapis do pliku
4
>>> print(open('temp1', 'rb').read())                          # Binarny tryb odczytu
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'
```

Jak widać, funkcja lub instrukcja `print` jest z reguły najlepszym sposobem wyświetlania tekstów, chyba że ktoś lubi pisać dużo kodu. Kolejne przykłady równoważności operacji wyświetlania i zapisu do plików poznamy w rozdziale 18. przy okazji implementacji odpowiednika funkcji `print`. Zaimplementujemy wówczas odpowiednik funkcji `print` z Pythona 2.3 do użycia w Pythonie 2.6.

Wyświetlanie niezależne od wersji

Jeśli pisane programy muszą działać pod różnymi wersjami Pythona, ale chcemy, aby wyświetlanie było kompatybilne z 3.0, mamy kilka możliwości poradzenia sobie z tym problemem. Po pierwsze, można użyć instrukcji `print` Pythona 2.6 i użyć skryptu `2to3`, który automatycznie

przekształci skrypty do postaci zgodnej z Pythonem 3.0. Więcej szczegółów na temat tego narzędzia można znaleźć w dokumentacji Pythona. W skrócie: skrypt *2to3* próbuje zmodyfikować kod napisany dla 2.X w taki sposób, aby działał w wersjach 3.X.

Inna opcja polega na użyciu funkcji `print` Pythona 3.0 w kodzie Pythona 2.6. W tym celu należy aktywować tę opcję za pomocą następującej instrukcji:

```
from __future__ import print_function
```

Ta instrukcja powoduje, że w 2.6 będzie dostępna funkcja `print` w takiej wersji, jaka jest w 3.0. Dzięki temu mamy możliwość używania funkcji Pythona 3.0 bez konieczności modyfikowania kodu dla różnych wersji Pythona.

Należy pamiętać, że proste instrukcje wyświetlania, jak prezentowane w tabeli 11.5, będą działać w *dowolnej* wersji Python: dzięki temu, że wyrażenia mogą być ujęte w nawiasy okrągłe, możemy „udawać”, że wywołujemy funkcję Pythona 3.0, dodając nawiasy po instrukcji `print` w 2.6. Jedyna wada tego podejścia polega na tym, że w przypadku próby wyświetlenia kilku obiektów ujęcie ich w nawiasy przekształca je w krotkę, a więc na ekranie pojawią się dodatkowe nawiasy okrągłe. W wersji 3.0 na przykład można podać kilka obiektów w funkcji `print`:

```
C:\misc> c:\python30\python
>>> print('mielonka')                                # Wywołanie funkcji print w 3.0
mielonka
>>> print('mielonka', 'szynka', 'jajka')      # Użycie kilku argumentów
mielonka szynka jajka
```

Pierwsze wywołanie zadziała tak samo, jak w Pythonie 2.6 z użyciem „symulacji” funkcji po dodaniu nawiasów, ale drugie wywołanie wygeneruje krotkę, która następnie zostanie wyświetlona:

```
C:\misc> c:\python26\python
>>> print('mielonka')                                # Instrukcja print w 2.6 z użyciem nawiasów
mielonka
>>> print('mielonka', 'szynka', 'jajka')      # To w rzeczywistości obiekt krotki!
('mielonka', 'szynka', 'jajka')
```

Aby kod był w pełni przenośny, należy wyświetlany tekst sformatować wcześniej jako pojedynczy ciąg znaków, wykorzystując wyrażenia formatowania ciągów znaków lub inne narzędzia tekstowe, które poznaliśmy w rozdziale 7.

```
>>> print('%s %s %s' % ('mielonka', 'szynka', 'jajka'))
mielonka szynka jajka
>>> print('{0} {1} {2}'.format('mielonka', 'szynka', 'jajka'))
mielonka szynka jajka
```

Oczywiście jeśli możemy pozwolić sobie na używanie wyłącznie wersji 3.0, można zapomnieć o tych komplikacjach, ale wielu programistów Pythona może napotkać kod w Pythonie 2.X, warto więc wiedzieć, jak sobie z nim radzić.



W książce w wielu miejscach wykorzystuję wywołanie funkcji `print` Pythona 3.0. Z reguły uprzedzam Czytelnika w przypadku, gdy wynik jakiejś operacji, który chcemy wyświetlić, jest krotką i w wyniku pojawią się dodatkowe nawiasy. Jednak jeśli uruchomisz przykładowy kod w Pythonie 2.6 i zobaczysz nawiasy, których nie ma w książce, to znak, że należy usunąć nawiasy z instrukcji `print` lub przepisać kod zgodnie z opisany wyżej regułami pisania przenośnego kodu wyświetlającego teksty, albo polubić dodatkowe nawiasy.

Znaczenie print i stdout

Równoważność instrukcji `print` i zapisu do `sys.stdout` ma duże znaczenie. Umożliwia przypisanie `sys.stdout` do obiektu zdefiniowanego przez użytkownika udostępniającego te same metody co pliki (na przykład `write`). Ponieważ instrukcja `print` po prostu przesyła tekst do metody `sys.stdout.write`, możemy przechwycić tekst zapisywany w naszych programach, przypisując `sys.stdout` do obiektu, którego metoda `write` w dowolny sposób przetworzy ten tekst.

Można na przykład przesyłać wyświetlany tekst do okna graficznego interfejsu użytkownika lub wysłać go do kilku miejsc docelowych, definiując obiekt z metodą `write` wymagającą przekierowania. Przykład takiej sztuczki zobaczymy przy okazji omawiania klas w szóstej części książki, jednak w skrócie będzie on przypominał poniższe rozwiązanie.

```
class FileFaker:  
    def write(self, string):  
        # Coś robi z łańcuchem znaków  
  
    import sys  
    sys.stdout = FileFaker()  
    print(someObjects)                                # Przesyła do metody write klasy
```

Takie rozwiązanie działa, ponieważ `print` jest czymś, co w następnej części książki nazwiemy operacją *polimorficzną* — dla instrukcji tej nie ma znaczenia, czym jest `sys.stdout`, ważne jest tylko, by obiekt ten miał metodę (a właściwie interfejs) `write`. W nowszych wersjach Pythona takie przekierowanie do obiektów jest jeszcze łatwiejsze dzięki rozszerzonej formie `print` ze znakami `>>` — nie musimy już w jawnym sposobie przestawiać `sys.stdout`.

```
myobj = FileFaker()                                # 3.0: Przekierowanie do obiektu na tę jedną operację  
print(someObjects, file= myobj)                   # Nie przestawia sys.stdout  
  
myobj = FileFaker()                                # 2.6: taki sam efekt  
print >> myobj, someObjects                      # Nie przestawia sys.stdout
```

Wbudowana funkcja Pythona `input` odczytuje plik `sys.stdin`, dzięki czemu w podobny sposób można przechwytywać żądania odczytu, wykorzystując do tego klasy implementujące metody `read` podobne do tych używanych w obiektach plików. Więcej informacji na ten temat można znaleźć w przykładzie z `input` i pętlą `while` z rozdziału 10.

Warto zauważyć, że ponieważ wyświetlany tekst trafia do strumienia `stdout`, w ten sposób można wyświetlać HTML w skryptach CGI. Pozwala nam to również na przekierowanie danych wejściowych i wyjściowych skryptu Pythona w systemowym wierszu poleceń w normalny sposób.

```
python script.py < plik_wejściowy > plik_wyjściowy  
python script.py | filterProgram
```

Narzędzia przekierowania strumieni wbudowane w funkcję i instrukcję `print` Pythona to w zasadzie pythonowa implementacja tych narzędzi powłoki systemowej.

Podsumowanie rozdziału

W niniejszym rozdziale rozpoczęliśmy nasze pogłębione omówienie instrukcji Pythona od zapoznania się z przypisaniem, wyrażeniami i instrukcjami `print`. Choć wszystkie one są stosunkowo proste w użyciu, mają pewne opcjonalne formy alternatywne, które w praktyce często się przydają. Rozszerzone instrukcje przypisania i forma instrukcji `print` z przekierowaniem pozwalają na przykład uniknąć ręcznego kodowania pewnych kwestii. Przy okazji

omówiliśmy składnię nazw zmiennych, techniki przekierowywania strumienia wyjścia i wiele często popełnianych błędów, takich jak przypisanie wyniku wywołania metody `append` z powrotem do zmiennej.

W kolejnym rozdziale będziemy kontynuować omawianie instrukcji, uzupełniając brakujące szczegóły dotyczące instrukcji `if` — najważniejszego narzędzia służącego do dokonywania wyboru w kodzie napisanym w Pythonie. Powróćmy również do modelu składni Pythona i przyjrzymy się zachowaniu wyrażeń Boolean. Zanim jednak do tego dojdzie, quiz podsumowujący rozdział pozwoli sprawdzić wiedzę nabyczą przy okazji lektury tekstu.

Sprawdź swoją wiedzę — quiz

1. Należy podać trzy sposoby przypisania tej samej wartości do trzech zmiennych.
2. Dlaczego trzeba uważać, kiedy przypisuje się zmienny obiekt do trzech zmiennych?
3. Co złego jest w kodzie `L = L.sort()`?
4. W jaki sposób można wykorzystać instrukcję `print` do przesłania tekstu do pliku zewnętrznego?

Sprawdź swoją wiedzę — odpowiedzi

1. Można skorzystać z przypisania z wieloma celami (`A = B = C = 0`), przypisania sekwencji (`A, B, C = 0, 0, 0`) lub kilku osobnych instrukcji przypisania w odrębnych wierszach (`A = 0, B = 0, C = 0`). W przypadku tej ostatniej techniki (zgodnie z informacjami z rozdziału 10.) można również połączyć trzy odrębne instrukcje w jeden wiersz, rozdzielając je średnikami (`A = 0; B = 0; C = 0`).

2. Jeśli przypiszemy je w poniższy sposób:

```
A = B = C = []
```

wszystkie trzy zmienne będą się odnosiły do tego samego obiektu, dlatego modyfikacja jednej z nich w miejscu (na przykład `A.append(99)`) będzie miała wpływ na pozostałe dwie zmienne. Ma to znaczenie wyłącznie w przypadku modyfikacji obiektów zmiennych (jak listy i słowniki) w miejscu; w przypadku obiektów niezmiennych, takich jak liczby i łańcuchy znaków, kwestia ta nie ma znaczenia.

3. Metoda listy `sort` jest podobna do metody `append`, ponieważ modyfikuje listę będącą jej podmiotem w miejscu — zwraca `None`, a nie zmodyfikowaną listę. Przypisanie jej wyniku z powrotem do `L` daje ustwienie `L` na `None`, a nie posortowaną listę. Jak zobaczymy niebawem w tej części książki, nowsza funkcja wbudowana `sorted` sortuje dowolną sekwencję i zwraca nową listę z wynikiem posortowania. Ponieważ nie jest ona modyfikacją w miejscu, jej wynik może być przypisany z powrotem do zmiennej.
4. Można przypisać `sys.stdout` do ręcznie otwartego pliku przed użyciem instrukcji `print` lub skorzystać z rozszerzonej formy instrukcji `print >>` w celu zapisania tekstu do pliku w tej jednej instrukcji. Można również przekierować całość wyświetlanego tekstu programu do pliku za pomocą specjalnej składni powłoki systemowej, jednak wykracza to poza kwestie związane z Pythonem.

Testy if i reguły składni

Niniejszy rozdział wprowadza instrukcję `if`, będącą w Pythonie podstawowym narzędziem wykorzystywanym w wyborze alternatywnych działań w oparciu o wyniki testu. Ponieważ będzie to nasze pierwsze szersze omówienie *instrukcji złożonych* — instrukcji z osadzonymi innymi instrukcjami — bardziej szczegółowo niż w rozdziale 10. zajmiemy się również bardziej uniwersalnymi koncepcjami stojącymi za modelem składni instrukcji Pythona. Ponieważ instrukcja `if` wprowadza pojęcie testów, rozdział ten będzie również omawiał wyrażenia typu Boolean i uzupełni pominięte dotychczas informacje dotyczące testów prawdy.

Instrukcje if

W uproszczeniu instrukcja `if` Pythona wybiera działanie, które należy wykonać. To podstawowe narzędzie wyboru w tym języku, reprezentujące dużą część *logiki programu* napisanego w Pythonie. Podobnie do innych instrukcji złożonych Pythona, może ona zawierać inne instrukcje, w tym kolejne `if`. Tak naprawdę Python pozwala nam łączyć instrukcje w programie w sposób sekwencyjny (tak by były wykonywane jedna po drugiej) i dowolnie zagnieżdżony (tak by wykonywane były jedynie pod pewnymi warunkami).

Ogólny format

Instrukcja `if` w Pythonie jest typowym przykładem instrukcji `if` występujących w większości języków proceduralnych. Przybiera formę testu `if`, po którym następuje jeden lub większa liczba testów `elif` (od „`else if`”) oraz końcowy opcjonalny blok `else`. Testy oraz część `else` zawierają powiązane bloki zagnieżdżonych instrukcji, wciętych w stosunku do wiersza nagłówka. Kiedy instrukcja `if` jest wykonywana, Python wykonuje blok kodu powiązany z pierwszym testem zwracającym wynik będący prawdą lub blok `else`, jeśli wszystkie testy zwracają wynik będący fałszem. Ogólna forma instrukcji `if` przedstawia się w następujący sposób:

```
if <test1>:                                # Test if
    <instrukcje1>                            # Powiązany blok
elif <test2>:                                # Opcjonalne testy elif
    <instrukcje2>
else:                                         # Opcjonalne else
    <instrukcje3>
```

Proste przykłady

By zademonstrować działanie instrukcji `if`, przyjrzyjmy się kilku krótkim przykładom. Wszystkie części są opcjonalne, z wyjątkiem początkowego testu `if` oraz powiązanych z nim instrukcji. Tym samym w najprostszym przypadku pozostałe części są pomijane.

```
>>> if 1:  
...     print('prawda')  
...  
prawda
```

Warto zwrócić uwagę, jak znak zachęty zmienia się na `...` w wierszach kontynuacji podstawowego interfejsu wykorzystywanego w sesji interaktywnej (w IDLE zamiast tego przechodzimy po prostu do kolejnego, wciętego wiersza — by powrócić do początku wiersza, należy nacisnąć przycisk *Backspace*). Pusty wiersz (który uzyskuje się za pomocą dwukrotnego naciśnięcia przycisku *Enter*) kończy i wykonuje całą instrukcję. Należy pamiętać, że `1` to inaczej wartość Boolean `True` oznaczająca prawdę, dlatego test powyższej instrukcji zawsze się powieźmie. By obsługiwać wynik będący fałszem, należy utworzyć część `else`.

```
>>> if not 1:  
...     print('prawda')  
... else:  
...     print('fałsz')  
...  
fałsz
```

Rozgałęzienia kodu

Poniżej znajduje się przykład bardziej skomplikowanej instrukcji `if`, w której obecne są wszystkie opcjonalne części.

```
>>> x = 'zabójczy królik'  
>>> if x == 'roger':  
...     print("jak się ma jessica?")  
... elif x == 'bugs':  
...     print("co słychać, doktorku?")  
... else:  
...     print('Uciekaj! Uciekaj!')  
...  
Uciekaj! Uciekaj!
```

Ta wielowierszowa instrukcja rozciąga się od wiersza z `if` aż do bloku `else`. Po uruchomieniu Python wykonuje instrukcje zagnieżdżone pod pierwszym testem zwracającym prawdę lub pod częścią `else`, jeśli wszystkie z testów zwracają fałsz (tak, jak w powyższym przykładzie). W praktyce zarówno `elif`, jak i `else` można pominąć, a w każdej z osadzonych części może być więcej instrukcji. Warto zwrócić uwagę na to, że słowa `if`, `elif` i `else` są ze sobą powiązane za pomocą indentacji — są wyrównane w pionie.

Osoby znające języki programowania takie, jak C lub Pascal, może zainteresować fakt, iż w Pythonie nie ma instrukcji `switch` czy `case` wybierającej działanie w oparciu o wartość zmiennej. Zamiast tego *rozgałęzienia kodu* zapisywane są albo jako seria testów `if` i `elif`, jak w przykładzie wyżej, albo za pomocą indeksowania słowników lub przeszukiwania list. Ponieważ słowniki i listy mogą być budowane w momencie wykonywania, czasami są bardziej elastyczne od zakodowanej na stałe logiki `if`.

```

>>> choice = 'szynka'
>>> print({'mielonka': 1.25,
...         'szynka': 1.99,
...         'jajka': 0.99,
...         'boczek': 1.10}[choice])
1.99

```

Choć zrozumienie takiego rozwiązania może przy pierwszej styczności z nim zająć chwilę, słownik ten jest tak naprawdę rozgałęzieniem. Indeksowanie po kluczu choice rozgałęzia kod na jedną wartość ze zbioru, podobnie jak instrukcja switch z języka C. Prawie równoważne, acz nieco bardziej rozwlekłe rozwiązanie z wykorzystaniem instrukcji Pythona if może z kolei wyglądać jak poniższy kod.

```

>>> if choice == 'mielonka':
...     print(1.25)
... elif choice == 'szynka':
...     print(1.99)
... elif choice == 'jajka':
...     print(0.99)
... elif choice == 'boczek':
...     print(1.10)
... else:
...     print('Zły wybór')
...
1.99

```

Warto zwrócić uwagę na to, że część else znajdująca się po if zajmuje się przypadkiem domyślnym, w którym klucz nie pasuje do niczego. Jak widzieliśmy w rozdziale 8., wartości domyślne słownika można zapisać za pomocą wyrażeń in, wywołań metod get lub przechwytywania wyjątków. Te same techniki można wykorzystać tutaj do zapisania domyślnego działania w rozgałęzieniu kodu opartym na słowniku. Poniżej widać, jak będzie to wyglądało przy użyciu metody get.

```

>>> branch = {'mielonka': 1.25,
...             'szynka': 1.99,
...             'jajka': 0.99}

>>> print(branch.get('mielonka', 'Zły wybór'))
1.25
>>> print(branch.get('boczek', 'Zły wybór'))
Zły wybór

```

Test przynależności in umieszczony w instrukcji if może mieć ten sam domyślny efekt.

```

>>> choice = 'boczek'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Zły wybór')
...
Zły wybór

```

Słowniki dobrze nadają się do łączenia wartości z kluczami, jednak co z bardziej skomplikowanymi działaniami, jakie możemy zapisać w blokach instrukcji powiązanych z instrukcjami if? W czwartej części książki zobaczymy, że słowniki mogą również zawierać *funkcje* reprezentujące bardziej skomplikowane działania i implementujące uniwersalne tablice skoków (ang. *jump table*). Takie funkcje pojawiają się jako wartości słownika, mogą być tworzone w postaci nazw funkcji lub za pomocą wyrażenia lambda i wywoływane są przez dodanie nawiasów rozpoczętujących ich działanie. Więcej informacji na ten temat znajdzie się w rozdziale 19.

Instrukcja 'switch' oparta na słowniku
Wartość domyślna dzięki has_key lub get

Choć rozgałęzianie oparte na słownikach przydaje się w programach obsługujących bardziej dynamiczne dane, większość programistów zauważa zapewne, że najprostszym sposobem wykonania rozgałęzienia kodu jest skorzystanie z instrukcji `if`. Uniwersalna reguła dotycząca programowania mówi, że kiedy mamy wątpliwości, lepiej jest postawić na prostotę i czytelność — będzie to najbardziej „pythonowy” wybór.

Reguły składni Pythona

Model składni Pythona został wprowadzony w rozdziale 10. Teraz, gdy przechodzimy do większych instrukcji, takich jak `if`, czas na przypomnienie i rozszerzenie przedstawionych wcześniej koncepcji związanych ze składnią. Python ma prostą składnię opartą na instrukcjach. Istnieje jednak kilka jej właściwości, o których należy wiedzieć.

- **Instrukcje wykonywane są jedna po drugiej, o ile nie wskażemy innego sposobu.** Python normalnie wykonuje instrukcje w pliku lub osadzonym bloku w kolejności od pierwszej do ostatniej, jednak instrukcje takie, jak `if` (i jak zobaczymy — również pętle) sprawiają, że interpreter przeskakuje w kodzie. Ponieważ ścieżka Pythona w programie nazywana jest *przebiegiem sterowania* (ang. *control flow*), instrukcje takie, jak `if`, wpływające na ten przebieg, nazywane są *instrukcjami sterującymi przebiegiem programu*.
- **Granice bloków i instrukcji wykrywane są w sposób automatyczny.** Jak widzieliśmy, wokół bloku kodu napisanego w Pythonie nie ma nawiasów klamrowych czy innych ograniczników typu `begin/end`. Zamiast tego Python wykorzystuje indentację instrukcji pod nagłówkiem do pogrupowania instrukcji w zagnieżdżony blok. W podobny sposób instrukcje Pythona nie są kończone średnikami. Zamiast tego koniec wiersza oznacza zazwyczaj koniec instrukcji umieszczonej w tym wierszu.
- **Instrukcje złożone składają się z wiersza nagłówka, znaku dwukropka i wciętych instrukcji.** Wszystkie instrukcje złożone w Pythonie tworzone są zgodnie z tym samym wzorcem. Najpierw występuje wiersz nagłówka zakończony dwukropkiem, po nim jedna lub większa liczba zagnieżdżonych instrukcji, zazwyczaj wciętych pod nagłówkiem. Wcięte instrukcje nazywane są *blokiem*. W instrukcji `if` części `elif` i `else` są jej elementami, ale jednocześnie również wierszami nagłówka dla własnych zagnieżdżonych bloków.
- **Puste wiersze, spacje i komentarze są zazwyczaj ignorowane.** Puste wiersze są ignorowane w plikach (jednak nie w sesji interaktywnej, gdzie kończą instrukcje złożone). Spacje wewnętrz instrukcji i wyrażeń są prawie zawsze ignorowane (z wyjątkiem literałów łańcuchów znaków i kiedy użyte są w indentacji). Komentarze ignorowane są zawsze — zaczynają się od znaku `#` (nie wewnętrz literała łańcucha znaków) i rozciągają aż do końca bieżącego wiersza.
- **Łańcuchy znaków dokumentacji są ignorowane, ale zapisywane i wyświetlane przez narzędzia.** Python obsługuje dodatkową formę komentarzy znaną jako *łańcuchy znaków dokumentacji* (ang. *docstring*), które — w przeciwieństwie do komentarzy ze znakiem `#` — są zachowywane w czasie wykonywania do późniejszego przejrzenia. Są to po prostu łańcuchy znaków pokazujące się na górze plików programów i niektórych instrukcji. Python ignoruje ich zawartość, jednak są one automatycznie dołączane do obiektów w czasie wykonywania i mogą być wyświetlane za pomocą narzędzi do dokumentacji. Łańcuchy znaków dokumentacji są częścią większej strategii dokumentacji Pythona i zostaną omówione w ostatnim rozdziale tej części książki.

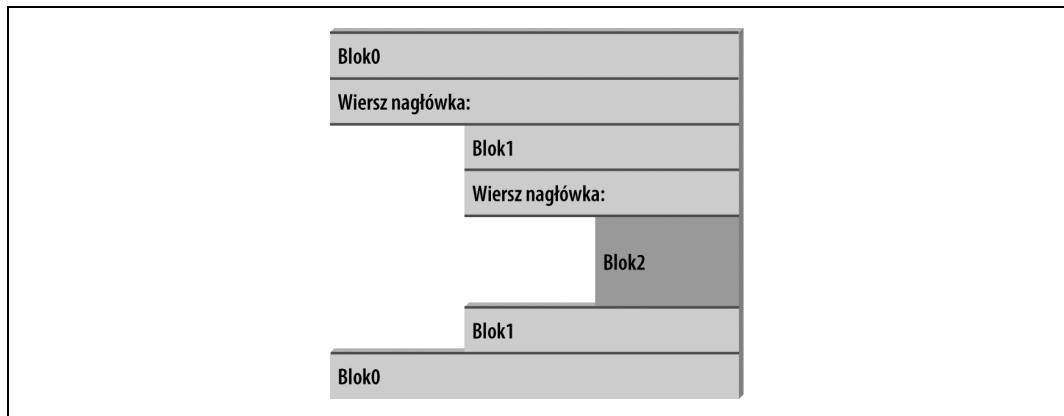
Jak widzieliśmy, w Pythonie nie ma deklaracji typu zmiennych. Już ten fakt sprawia, że składnia tego języka jest o wiele prostsza niż to, do czego możemy być przyzwyczajeni. Dla większości nowych użytkowników brak nawiasów klamrowych i średników oznaczających w wielu innych językach programowania bloki i instrukcje wydaje się jednak największą nowością składniową w Pythonie, dlatego spróbujmy wyjaśnić tę kwestię nieco bardziej szczegółowo.

Ograniczniki bloków — reguły indentacji

Python automatycznie wykrywa granice bloków dzięki *indentacji* wierszy — czyli pustej przestrzeni po lewej stronie kodu. Wszystkie instrukcje wcięte na tę samą odległość w prawą stronę należą do tego samego bloku kodu. Innymi słowy, instrukcje jednego bloku są ze sobą wyrównane w pionie, jak kolumna. Blok kończy się na końcu pliku lub po napotkaniu mniej wciętego wiersza. Bloki głębiej zagnieżdżone są po prostu wcinane na większą odległość w prawą stronę w stosunku do instrukcji z bloku je zawierającego.

Przykładowo na rysunku 12.1 zademonstrowano strukturę bloków poniższego kodu.

```
x = 1
if x:
    y = 2
    if y:
        print('blok2')
    print('blok1')
print('blok0')
```



Rysunek 12.1. Zagnieżdżone bloki kodu. Blok zagnieżdżony rozpoczyna się od instrukcji wciętej na większą odległość w prawo i kończy na instrukcji wciętej na mniejszą odległość lub na końcu pliku

Powyższy kod zawiera trzy bloki. Pierwszy z nich (najwyższy poziom pliku) nie jest w ogóle wcięty, drugi (wewnętrzny zewnętrznej instrukcji `if`) jest wcięty o trzy spacje, a trzeci (instrukcja `print` pod zagnieżdżonym `if`) wcięty jest o sześć spacji.

Zazwyczaj kod najwyższego poziomu (niezagnieżdżony) musi rozpoczynać się w pierwszej kolumnie. Bloki zagnieżdżone mogą zaczynać się w dowolnej kolumnie. Indentacja może zawierać dowolną liczbę spacji i tabulatorów, dopóki dla wszystkich instrukcji w jednym bloku wcięcie będzie takie samo. Pythona nie interesuje, *w jaki sposób* będziemy wcinać kod. Ważne jest jedynie to, by odbywało się to w sposób spójny. Popularnymi konwencjami są cztery spacje lub jeden tabulator na poziom indentacji, jednak w świecie Pythona nie istnieje bezwzględny standard.

Indentacja kodu jest w praktyce bardzo naturalna. Przykładowo poniższy (dość niemądry) fragment kodu demonstruje często spotykane błędy indentacji w kodzie napisanym w Pythonie.

```
x = 'MIELONKA'  
if 'plot' in 'żywoplot':  
    print(x * 8)  
    x += 'NI'  
    if x.endswith('NI'):  
        x *= 2  
    print(x)                                # Błąd — wciecie pierwszego wiersza  
                                              # Błąd — nieoczekiwana indentacja  
                                              # Błąd — niespójna indentacja
```

Wersja tego kodu z poprawną indentacją będzie wyglądać następująco — nawet w przypadku tak naciąganego przykładu poprawne wciecie kodu sprawia, że jego zawartość jest o wiele bardziej zrozumiała:

```
x = 'MIELONKA'  
if 'plot' in 'żywoplot':  
    print(x * 8)  
    x += 'NI'  
    if x.endswith('NI'):  
        x *= 2  
    print(x)                                # Wyświetla "MIELONKANIMIELONKANI"
```

Jednym istotnym miejscem w Pythonie, w którym białe znaki mają znaczenie, jest użycie ich po lewej stronie kodu — do indentacji. We wszystkich innych kontekstach spacji można użyć lub nie. Indentacja jest tak naprawdę częścią składni Pythona, a nie tylko wskazówką stylistyczną. Wszystkie instrukcje w jednym bloku muszą być wcięte na tę samą odległość, gdyż inaczej Python zwróci nam błąd składni. Jest to celowe — ponieważ nie musimy w jawnym sposobie oznaczać końca zagnieżdzonego bloku kodu, część zbędnych elementów składniowych występująca w innych językach programowania mogła zostać w Pythonie pominięta.

Zgodnie z informacjami z rozdziału 10. to, że indentacja stała się częścią modelu składni, wymusza również spójność kodu — kluczowy element czytelności w ustrukturyzowanych językach programowania, takich jak Python. Składnia Pythona czasami opisywana jest jako „to, co widzisz, jest tym, co otrzymujesz” (WYSIWYG) — wciecie każdego wiersza kodu jednoznacznie informuje użytkownika, z czym kod ten jest powiązany. Ten spójny wygląd sprawia, że kod napisany w Pythonie łatwiej jest utrzymywać i wykorzystywać powtórnie.

Indentacja jest o wiele bardziej naturalna, niż mogłyby to sugerować podane tu informacje, a dzięki niej kod odzwierciedla swoją strukturę logiczną. Spójne wcinanie kodu zawsze spełnia wymagania składni Pythona. Co więcej, większość edytorów tekstu (włącznie z IDLE) ułatwia przestrzeganie reguł modelu składni Pythona poprzez automatyczne wcinanie kodu w miarę pisania.

Unikanie mieszania tabulatorów i spacji — nowa opcja sprawdzania błędów w Pythonie 3.0

Zasada numer jeden: choć do indentacji kodu można wykorzystać spacje lub tabulatory, mieszanie obu tych opcji w jednym bloku kodu nie jest najlepszym pomysłem. Lepiej jest konsekwentnie używać albo jednego, albo drugiego rozwiązania. Tabulatory zawierają wystarczającą liczbę spacji, by przesunąć początek kodu do ośmiu znaków spacji, a kod będzie działał, jeśli tabulatory i spacje będą się łączyć w spójny sposób. Taki kod może jednak być trudniejszy do modyfikacji. Co gorsza, pomieszanie ze sobą spacji i tabulatorów sprawia, że kod trudniej będzie się czytało — tabulatory mogą wyglądać zupełnie inaczej w edytorech tekstu innych programistów.

Z tego właśnie powodu Python 3.0 zwraca błąd, jeśli w skrypcie tabulatory i spacje wykorzystane do indentacji kodu użyte są w ramach jednego bloku w sposób niespójny (to znaczy w sposób uzależniony od odpowiednika tabulatora w spacjach). Python 2.6 pozwala na wykonanie takich skryptów, jednak zawiera także opcję wiersza polecen -t, która ostrzega przed niespójnym użyciem tabulatorów, oraz opcję -tt, zwracającą błędy dla takiego kodu (przełączniki te można zastosować w wierszu polecen, na przykład: `python -t main.py` w oknie powłoki systemowej). Zwracanie błędów w Pythonie 3.0 jest odpowiednikiem przełącznika -tt z Pythona 2.6.

Ograniczniki instrukcji — wiersze i kontynuacje

Normalnie instrukcja w Pythonie kończy się z końcem wiersza, w którym się znajduje. Kiedy instrukcja jest zbyt dłuża, by zmieścić się w jednym wierszu, można skorzystać z kilku specjalnych reguł umożliwiających rozcięcie jej na kilka wierszy.

- **Instrukcje mogą rozciągać się na kilka wierszy, jeśli kontynuujemy otwartą parę znaków składniowych.** Python pozwala na kontynuację instrukcji w kolejnym wierszu, jeśli kod umieszczony jest w parze nawiasów zwykłych (()), klamrowych ({ }) lub kwadratowych ([]). Wyrażenia w nawiasach czy literałы słowników i list mogą się na przykład rozciągać na dowolną liczbę wierszy; instrukcja nie kończy się, dopóki interpreter Pythona nie dojdzie do końca wiersza, w którym wpisaliśmy zamkającą część nawiasów (znak), } lub]). Wiersze kontynuacji (czyli wiersze od drugiego i dalej) mogą rozpoczynać się od dowolnego poziomu zagnieżdżenia, jednak z uwagi na czytelność powinno się starać wyrównywać je w pionie. Reguła dotycząca otwierania par nawiasów dotyczy także zbiorów i słowników składanych z Pythona 3.0.
- **Instrukcje mogą rozciągać się na kilka wierszy, jeśli kończą się znakiem ukośnika lewego (\).** To nieco przestarzała opcja, ale jeśli instrukcja ma się rozciągać na kilka wierszy, można również dodać znak ukośnika lewego (znak \ nieosadzony w literale łańcucha znaków lub komentarzu) na końcu poprzedniego wiersza, by wskazać, że będzie on kontynuowany w następnym. Ponieważ można osiągnąć to samo za pomocą pary nawiasów umieszczonych wokół większości konstrukcji, ukośniki lewe w zasadzie nie są już stosowane. Takie rozwiązanie jest podatne na błędy — przypadkowe zapomnienie o znaku ukośnika generuje błąd składni i może nawet powodować, że kolejny wiersz zostanie po cichu błędnie wzięty za nową instrukcję, co daje nieoczekiwane rezultaty.
- **Specjalne reguły dotyczące literałów łańcuchów znaków.** Jak wiemy z rozdziału 7., bloki łańcuchów znaków opatrzone potrójnymi apostrofami lub cudzysłowami normalnie służą do rozciągania długich łańcuchów znaków na kilka wierszy. Z rozdziału tego wiemy również, że przylegające do siebie literaly łańcuchów znaków są w sposób niejawny poddawane konkatenacji. W połączeniu ze wspomnianą wcześniej regułą dotyczącą otwartych par nawiasów opakowanie takiej konstrukcji w nawiasy pozwala na rozcięcie łańcucha znaków na kilka wierszy.
- **Inne reguły.** Istnieje jeszcze kilka kwestii związanych z ogranicznikami instrukcji. Choć jest to stosunkowo rzadkie, instrukcję możemy kończyć średnikiem — taki zapis pozwala na umieszczenie większej liczby prostych (niezłożonych) instrukcji w jednym wierszu. Również komentarze i puste wiersze mogą pojawiać się w dowolnym miejscu pliku. Komentarze (rozpoczynające się od znaku #) kończą się na końcu wiersza, w którym występują.

Kilka przypadków specjalnych

Poniżej widać, jak może wyglądać wiersz kontynuacji w przypadku zastosowania reguły skla-dniowej z otwartymi parami nawiasów. Konstrukcje oznaczone w ten sposób, takie jak listy umieszczone w nawiasach kwadratowych, mogą się rozciągać na dowolną liczbę wierszy.

```
L = ["Dobry",
      "Zły",
      "Paskudny"] # Para nawiasów rozciąga się na kilka wierszy
```

Takie rozwiązanie będzie także działało dla dowolnej zawartości nawiasów zwykłych (wyrażeń, argumentów funkcji, nagłówków funkcji, krotek i wyrażeń generatorów), a także zawartości nawiasów klamrowych (słowników, a w Pythonie 3.0 również literałów zbiorów oraz słowników składanych). Część tych narzędzi omówimy w kolejnych rozdziałach, jednak reguła ta w sposób naturalny obejmuje większość konstrukcji, które w praktyce rozciągają się na kilka wierszy.

W celu kontynuowania wiersza można również użyć znaków ukośnika lewego, ale nie jest to w Pythonie często spotykane.

```
if a == b and c == d and \
   d == e and f == g:
    print('stary') # Ukośniki lewe pozwalają na kontynuację
```

Ponieważ w nawiasach można umieścić dowolne wyrażenie, zazwyczaj to z tej techniki korzysta się, kiedy kod ma się rozciągać na kilka wierszy. Wystarczy tylko opakować część instrukcji w nawiasy:

```
if (a == b and c == d and
    d == e and e == f):
    print('nowy') # To samo robią nawiasy
```

Tak naprawdę użycie ukośników lewych jest krytykowane, ponieważ zbyt łatwo można ich nie zauważyc, a także całkowicie je pominiąć. W poniższym kodzie po użyciu znaku ukośnika do zmiennej `x` przypisana zostaje wartość 10, zgodnie z zamierzeniami. Jeśli jednak znak \ przypadkowo pominiemy, do zmiennej `x` przypisana zostanie zamiast tego wartość 6 i nie zostanie zgłoszony żaden błąd (+4 jest samo w sobie poprawną instrukcją wyrażenia).

W prawdziwym programie z bardziej skomplikowaną operacją przypisania mogłoby to być źródłem wyjątkowo paskudnego błędu.¹

```
x = 1 + 2 + 3 \
+4 # Pominięcie znaku \ znacznie zmienia kod
```

W innym specjalnym przypadku Python pozwala na zapis większej liczby prostych instrukcji (instrukcji bez zagnieżdżonych innych instrukcji) w tym samym wierszu, rozzielonych średnikami. Niektórzy programiści wykorzystują tę formę do zaoszczędzenia miejsca w pliku programu, jednak zazwyczaj kod jest czytelniejszy, kiedy przestrzegamy reguły umieszczania jednej instrukcji w wierszu.

¹ Mówiąc szczerze, aż dziwne, że opcja ta nie została usunięta w Pythonie 3.0, biorąc pod uwagę niektóre inne wprowadzone w tej wersji zmiany! W tabeli P.2 w „Przedmowie” znajduje się lista elementów usuniętych w wersji 3.0; część z nich wydaje się stosunkowo nieszkodliwa w porównaniu z niebezpieczeństwem związanym z kontynuacjami kodu opartymi na znakach ukośników lewych. Z drugiej strony celem tej książki jest przedstawienie Pythona, a nie populistyczna krytyka, więc najprostszą radą, jakiej mogę udzielić, jest: „Nie należy z tego korzystać”.

```
x = 1; y = 2; print(x) # Większa liczba prostych instrukcji
```

Jak wiemy z rozdziału 7., literały łańcuchów znaków ujęte w potrójne znaki apostrofów także mogą się rozciągać na kilka wierszy. Dodatkowo jeśli dwa literały łańcuchów znaków pojawiają się obok siebie, zostają połączone w procesie konkatenacji, tak jakby pomiędzy nie wstawiony został znak +. W połączeniu z regułą dotyczącą otwartych par nawiasów oznacza to, że opakowanie w nawiasy pozwala na rozciąganie się tej formy na kilka wierszy. Pierwszy z poniższych przykładów kodu wstawia znaki nowego wiersza w miejscu złamania wiersza i przypisuje S do '\naaaa\nbbbb\ncccc', natomiast drugi przykład dokonuje niejawnej konkatenacji i przypisuje S do 'aaaabbbbcccc'. Komentarze są ignorowane w drugim przykładzie, jednak dołączane do łańcucha znaków w pierwszym:

```
S = """
aaaa
bbbb
cccc"""

S = ('aaaa'
      'bbbb'
      'cccc') # Tutaj komentarze są ignorowane
```

Wreszcie Python pozwala również na przeniesienie ciała instrukcji złożonej do wiersza nagłówka, pod warunkiem że ciało jest pojedynczą, prostą instrukcją. Najczęściej można to zobaczyć na przykładzie prostych instrukcji if z pojedynczym testem i działaniem.

```
if 1: print('witaj') # Prosta instrukcja w wierszu nagłówka
```

Niektóre z tych przypadków specjalnych można ze sobą łączyć i w rezultacie otrzymać kod trudny do odczytania, jednak nie polecam takiego działania. Należy trzymać się reguły jednej instrukcji na wiersz i wcinać wszystkie bloki z wyjątkiem tych najprostszych. Po sześciu miesiącach będziemy szczęśliwi, że tak robiliśmy.

Testy prawdziwości

Pojęcia porównania, równości oraz wartości prawdy zostały wprowadzone w rozdziale 9. Ponieważ instrukcja if jest pierwszą omawianą instrukcją, która tak naprawdę wykorzystuje wynik testu, rozszerzymy nieco koncepcje przedstawione wcześniej. Operatory Boolean Pythona nieznacznie się różnią od swoich odpowiedników w językach takich, jak C. W Pythonie:

- Dowolna liczba niebędąca zerem i dowolny niepusty obiekt są prawdą.
- Liczby o wartości zero, puste obiekty i specjalny obiekt None uznawane są za fałsz.
- Porównania i testy równości stosowane są do struktur danych w sposób rekurencyjny.
- Porównania i testy równości zwracają True i False (odpowiedniki liczb 1 i 0).
- Operatory Boolean and i or zwracają obiekt argumentu prawdy lub fałszu.

Mówiąc w skrócie, operatory Boolean wykorzystuje się do łączenia wyników innych testów. W Pythonie istnieją trzy operatory wyrażeń Boolean:

X and Y
jest prawdziwe, kiedy zarówno X, jak i Y są prawdziwe.
X or Y
jest prawdziwe, kiedy X lub Y jest prawdziwe.

```
not X
```

jest prawdziwe, kiedy X jest fałszywe (wyrażenie zwraca True lub False).

X i Y mogą mieć dowolną wartość prawdziwości lub mogą być dowolnym wyrażeniem zwracającym wartość prawdziwości (na przykład testem równości lub porównaniem zakresu). Operatory Boolean wpisywane są w Pythonie jako słowa (zamiast zapisu &&, || i ! z języka C). Operatory and oraz or zwracają w Pythonie obiekty (obliczone do True dla or i False dla and), a nie wartości True lub False. Przyjrzyjmy się kilku przykładom w celu przekonania się, jak to działa.

```
>>> 2 < 3, 3 < 2 # Mniejszy od — zwraca True lub False (1 lub 0)
(True, False)
```

Porównania wielkości, jak powyższy przykład, zwracają True i False — wartości, które (jak wiemy z rozdziałów 5. i 9.) są tylko inaczej zapisanymi wersjami liczb całkowitych 1 oraz 0. Są one jedynie wyświetlane w inny sposób, jednak poza tym niczym się nie różnią.

Z drugiej strony, operatory and oraz or zawsze zwracają obiekt — albo obiekt znajdujący się po lewej stronie operatora, albo ten z prawej strony. Jeśli sprawdzimy ich wyniki w if lub innych instrukcjach, będą one zgodne z oczekiwaniami (należy pamiętać, że każdy obiekt jest albo prawda, albo fałszem), jednak nie otrzymamy z powrotem prostego True lub False.

W przypadku testów or Python analizuje obiekty argumentów od lewej do prawej strony i zwraca pierwszy będący prawdą. Co więcej, po odnalezieniu pierwszego prawdziwego argumentu Python zatrzymuje wyszukiwanie — zazwyczaj znane jako skrótna analiza typu *short-circuit* (czyli w przybliżeniu: po linii najmniejszego oporu — nawiązanie do tego, że ustalone wyniku kończy resztę wyrażenia).

```
>>> 2 or 3, 3 or 2 # Zwrócenie lewego argumentu, jeśli jest prawdą
(2, 3) # Inaczej zwrócenie prawego argumentu (prawdy lub fałszu)
>>> [] or 3
3
>>> [] or {}
{}
```

W pierwszym wierszu powyższego przykładu oba argumenty (2 oraz 3) są prawdziwe (czyli niebędące zerem), dlatego Python zawsze zatrzymuje się i zwraca obiekt z lewej strony. W dwóch kolejnych testach lewy argument jest pusty, dlatego Python analizuje i zwraca obiekt z prawej strony (który może po przetestowaniu zwracać wartość będącą prawdą lub fałszem).

Operacje and również zatrzymują się, kiedy tylko odnaleziony zostanie ich wynik. W tym przypadku Python oblicza argumenty od lewej do prawej strony i zatrzymuje się na pierwszym obiekcie będącym fałszem.

```
>>> 2 and 3, 3 and 2 # Zwrócenie lewego argumentu, jeśli jest fałszem
(3, 2) # Inaczej zwrócenie prawego argumentu (prawdy lub fałszu)
>>> [] and {}
[]
>>> 3 and []
[]
```

W powyższym przykładzie oba argumenty z pierwszego wiersza są prawdą, dlatego Python analizuje obie strony i zwraca obiekt znajdujący się z prawej. W drugim teście lewy argument jest fałszem ([]), dlatego Python zatrzymuje się i zwraca go jako wynik testu. W ostatnim teście lewa strona jest prawdą (3), dlatego Python analizuje i zwraca obiekt z prawej (którym okazuje się [] będące fałszem).

Rezultat jest taki sam jak w C i większości innych języków programowania — otrzymujemy wartość, która jest logiczną prawdą lub fałszem, kiedy przetestuje się ją w `if` lub `while`. W Pythonie wartości Boolean zwracają jednak obiekty (lewy lub prawy), a nie po prostu liczbowe odpowiedniki prawdy lub fałszu.

Takie zachowanie `and` i `or` może się na pierwszy rzut oka wydawać dziwne, jednak w ramce „Znaczenie operatorów Boolean” na końcu niniejszego rozdziału można zobaczyć przykłady sytuacji, w których może ono być przydatne dla programistów Pythona. W kolejnym podrozdziale pokażemy również często spotykany sposób wykorzystania tego zachowania, a także jego odmianę w nowszych wersjach Pythona.

Wyrażenie trójargumentowe `if/else`

Jedną z często spotykanych ról omówionych wyżej operatorów Boolean jest zapisywanie w kodzie wyrażeń działających tak samo jak instrukcje `if`. Rozważmy poniższą instrukcję, ustawiającą zmienną `A` na `Y` lub `Z` w zależności od wartości prawdy `X`.

```
if X:  
    A = Y  
else:  
    A = Z
```

Czasami jednak elementy takiej instrukcji są tak proste, że wydaje się przesadą rozciąganie ich na cztery wiersze. Innym razem konieczne może się okazać zagnieźdżenie takiej konstrukcji w większej zamiast przypisywania jej wyniku do zmiennej. Z tego powodu (a poza tym dlatego, że podobne narzędzie znajduje się również w języku C²) w Pythonie 2.5 wprowadzono nowy format instrukcji pozwalający na napisanie tego samego w jednym wyrażeniu:

```
A = Y if X else Z
```

Wyrażenie to ma dokładnie ten sam efekt jak poprzednia czterowierszowa instrukcja `if`, jednak jest łatwiejsze do zapisania. Tak jak w instrukcji `if`, Python wykonuje wyrażenie `Y` tylko wtedy, gdy `X` okazuje się prawdą, i wykonuje wyrażenie `Z` tylko wtedy, gdy `X` okazuje się fałszem. Jest to zatem skrót — charakterystyczny dla opisanych wcześniej operacji wykonywanych przez operatory Boolean. Poniżej widać przykłady zastosowania tego wyrażenia.

```
>>> A = 't' if 'mielonka' else 'f'          # Niepusty, czyli prawda  
>>> A  
't'  
>>> A = 't' if '' else 'f'  
>>> A  
'f'
```

Przed wersją 2.5 (i nawet już po jej opublikowaniu, jeśli koniecznie chcemy) ten sam efekt można często osiągnąć po uważnym połączeniu operatorów `and` oraz `or`, ponieważ zwracają one albo obiekt z lewej strony, albo ten z prawej.

```
A = ((X and Y) or Z)
```

² Tak naprawdę wyrażenie `X if Y else Z` ma w Pythonie nieco inną kolejność od `Y ? X : Z` z języka C. Podobno zmianę tę wprowadzono w oparciu o analizę wzorców użycia w kodzie Pythona. Wieść głosi, że kolejność ta została wybrana po części również dlatego, by zniechęcić byłych programistów języka C do nadużywania tego wzorca. Należy pamiętać, że proste jest lepsze od skomplikowanego, zarówno w Pythonie, jak i w każdej innej dziedzinie.

Taki kod działa, jednak jest w nim pewna pułapka — musimy móc zakładać, że `Y` będzie prawdą. W tym przypadku rezultat będzie taki sam — najpierw działa operator `and` zwracający `Y`, jeśli `X` jest prawdą. Jeśli tak nie jest, operator `or` zwraca po prostu `Z`. Innymi słowy, otrzymujemy „jeśli `X`, to `Y`; w przeciwnym razie `Z`”.

Połączenie `and` i `or` wydaje się również wymagać chwili zastanowienia, by zrozumieć je za pierwszym razem, kiedy je zobaczymy. Od wersji 2.5 nie ma już takiej potrzeby — jeśli potrzebne jest nam w postaci wyrażenia, wystarczy użyć równoważnego i łatwiejszego do zapamiętania `Y if X else Z`; można również użyć pełnej instrukcji `if`, jeśli jej części są bardziej skomplikowane.

Na marginesie można dodać, że podobny efekt da w Pythonie użycie poniższego wyrażenia, ponieważ funkcja `bool` tłumaczy `X` na odpowiednik liczb całkowitych 1 lub 0, który można następnie wykorzystać do wybrania wartości prawdy lub fałszu z listy.

```
A = [Z, Y][bool(X)]
```

Obrazuje to poniższy przykład.

```
>>> ['f', 't'][bool('')]  
'f'  
>>> ['f', 't'][bool('mielonka')]  
't'
```

Nie jest to jednak dokładnie to samo, ponieważ Python nie zastosuje w tym przypadku skrótu — zawsze wykonane zostaną zarówno `Z`, jak i `Y`, bez względu na wartość `X`. Ze względu na te komplikacje od wersji 2.5 lepiej jest użyć prostszego i łatwiejszego do zrozumienia wyrażenia `if/else`. Nawet z tego wyrażenia lepiej jest jednak korzystać oszczędnie i tylko wtedy, gdy jego części są stosunkowo proste. W innym przypadku zaleca się napisanie pełnej instrukcji `if`, co ułatwi wprowadzanie modyfikacji do kodu w przyszłości. Nasi współpracownicy również ucieszą się z takiego wyboru.

W kodzie napisanym dla wersji Pythona starszych od 2.5 (a także napisanym przez programistów języka C, którzy nie przestawili się jeszcze na nowy, lepszy tryb programowania) nadal można jeszcze spotkać wersje z operatorami `and` oraz `or`.

Podsumowanie rozdziału

W niniejszym rozdziale przedstawiliśmy instrukcję `if` Pythona. Co więcej, ponieważ była to pierwsza omawiana przez nas złożona instrukcja logiczna, przejrzaliśmy również ogólne reguły składni Pythona i pogłębiliśmy naszą wiedzę na temat wykonywania testów prawdziwości. Przy okazji przyjrzaliśmy się również sposobowi zapisu rozgałęzień kodu w Pythonie i dowiedzieliśmy się czegoś na temat wyrażenia `if/else` wprowadzonego w Pythonie 2.5.

W kolejnym rozdziale będziemy kontynuować omawianie instrukcji proceduralnych, poszerzając naszą wiedzę na temat pętli `for` oraz `while`. Nauczymy się alternatywnych sposobów tworzenia pętli w Pythonie; niektóre z nich będą lepsze od pozostałych. Przedtem jednak pora wykonać quiz podsumowujący niniejszy rozdział.

Znaczenie operatorów Boolean

Jednym z często spotykanych zastosowań nieco nietypowego zachowania operatorów Boolean Pythona jest wybór ze zbioru obiektów za pomocą operatora `or`. Instrukcja podobna do poniższej:

```
X = A or B or C or None
```

ustawia `X` na pierwszy niepusty (będący prawdą) obiekt spośród `A`, `B` i `C` lub na `None`, jeśli wszystkie obiekty są puste. Powyższy kod działa, ponieważ operator `or` zwraca jeden z dwóch swoich obiektów. Technika ta okazuje się stosunkowo często stosowana w Pythonie. By wybrać niepusty obiekt ze zbioru o stałym rozmiarze, wystarczy połączyć obiekty w jednym wyrażeniu `or`. W prostszej postaci jest to często wykorzystywane do oznaczenia wartości domyślnej — poniższy kod ustawia `X` na `A`, jeśli `A` jest prawdziwe (lub niepuste), a w innym przypadku na `default`:

```
X = A or default
```

Zrozumienie skrótoowej analizy obiektów jest dość istotne, ponieważ wyrażenia znajdujące się po prawej stronie operatora Boolean mogą wywoływać funkcje wykonujące znaczące działania lub dawać efekty uboczne, które nie wystąpią, jeśli zastosowana zostanie analiza skrócona.

```
if f1() or f2(): ...
```

W powyższym kodzie, jeśli `f1` zwraca wartość będącą prawdą (niepustą), Python nigdy nie zwróci `f2`. By zagwarantować, że wykonane zostaną obie funkcje, musimy je wywołać przed `or`.

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

W niniejszym rozdziale widzieliśmy już zastosowanie takiego zachowania. Ze względu na sposób działania operatorów Boolean wyrażenie `((A and B) or C)` można wykorzystać do prawie dokładnego emulowania instrukcji `if/else` (bardziej szczegółowy opis działania tej postaci znajduje się w treści rozdziału).

Dodatkowe przypadki użycia operatorów Boolean widzieliśmy już w poprzednich rozdziałach. Zgodnie z informacjami z rozdziału 9., ponieważ każdy obiekt jest albo prawdą, albo fałszem, w Pythonie często łatwiej jest testować bezpośrednio obiekt (`if x:`), niż porównywać go z pustą wartością (`if x != ''`). W przypadku łańcuchów znaków te dwa testy są równoważne. Jak wiemy także z rozdziału 5., odgórnie zdefiniowane wartości Boolean `True` oraz `False` są tym samym co liczby całkowite `1` oraz `0` i przydadają się do inicjalizowania zmiennych (`X = False`), testów pętli (`while True:`), a także wyświetlania wyników w sesji interaktywnej.

Warto także zwrócić uwagę na omówienie przeciążania operatorów w szóstej części książki. Kiedy definiujemy nowe typy obiektów za pomocą klas, możemy określić ich naturę Boolean za pomocą metod `__bool__` oraz `__len__` (`__bool__` w Pythonie 2.6 nosi nazwę `__nonzero__`). Druga z tych metod jest sprawdzana, jeśli nie ma pierwszej z nich, i przypisuje fałsz, zwracając wartość o długości zero — pusty obiekt uznawany jest za fałsz.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób można zapisać w Pythonie rozgałęzienie kodu?
2. W jaki sposób można w Pythonie zapisać instrukcję `if/else` w postaci wyrażenia?
3. Jak można sprawić, by jedna instrukcja rozciągała się na kilka wierszy?
4. Co oznaczają słowa `True` i `False`?

Sprawdź swoją wiedzę — odpowiedzi

1. Instrukcja `if` z kilkoma częściami `elif` jest często najlepszym sposobem zapisania kodu rozgałęzionego, choć niekoniecznie jest przy tym najbardziej zwięzłą metodą. Podobny rezultat może często dać indeksowanie słowników, w szczególności kiedy słownik zawiera wywoływalne funkcje zapisane za pomocą instrukcji `def` czy wyrażeń `lambda`.
2. W Pythonie 2.5 i późniejszych wersjach wyrażenie w postaci `Y if X else Z` zwraca `Y`, jeśli `X` jest prawdą; w przeciwnym razie zwraca `Z`. Jest ono odpowiednikiem czterowierszowej instrukcji `if`. Połączenie operatorów `and` oraz `or` — `((X and Y) or Z)` — może działać w ten sam sposób, jednak jest mniej oczywiste i wymaga, by część `Y` była prawdą.
3. Należy opakować instrukcję w parę nawiasów `()`, `[]` lub `{ }`), dzięki czemu może ona rozciągać się na dowolną liczbę wierszy. Instrukcja kończy się, kiedy Python zobaczy zamkającą, prawą połowę pary nawiasów, a wiersze instrukcji od drugiego i dalej mogą się rozpoczynać na dowolnym poziomie indentacji.
4. `True` i `False` są zapisanymi w innej postaci wersjami liczb całkowitych `1` i `0`. Zawsze oznaczają one w Pythonie wartości Boolean prawdy i fałszu. Są one dostępne do użycia w testach prawdziwości oraz inicjalizacji zmiennych i wyświetlane jako wyniki wyrażeń w sesji interaktywnej.

Pętle while i for

Niniejszy rozdział kończy nasze omówienie instrukcji proceduralnych Pythona, przedstawiając dwie najważniejsze konstrukcje *pętli* Pythona — instrukcji powtarzających jakieś działanie. Pierwsza z nich, instrukcja `while`, umożliwia zapisywanie w kodzie uniwersalnych pętli. Druga, instrukcja `for`, pozwala przechodzić elementy w obiekcie sekwencji i wykonywać blok kodu dla każdego elementu.

Z obydwoema instrukcjami spotkaliśmy się już w sposób nieformalny wcześniej, tutaj natomiast uzupełnimy informacje na temat ich stosowania. A skoro już przy tym jesteśmy, omówimy także kilka mniej popularnych instrukcji stosowanych wewnątrz pętli, takich jak `break` oraz `continue`, i przedstawimy kilka funkcji wbudowanych wykorzystywanych najczęściej w połączeniu z pętlami — jak `range`, `zip` czy `map`.

W Pythonie istnieją również inne operacje pętli, jednak te dwie instrukcje są podstawową składnią służącą do tworzenia kodu dla powtarzanych działań. Z tego powodu w kolejnym rozdziale zapoznamy się z powiązaną koncepcją *protokołu iteracji* Pythona (wykorzystywanego przez pętlę `for`) i uzupełnimy pewne szczegóły dotyczące *list składanych* — bliskiego kuzyna tej pętli. W dalszych rozdziałach będziemy omawiać jeszcze bardziej egzotyczne narzędzia iteracyjne, takie jak *generatory*, *filter* oraz *reduce*. Na razie jednak pozostaniemy przy nieco prostszych zagadnieniach.

Pętle while

Instrukcja `while` Pythona jest najbardziej uniwersalną konstrukcją iteracyjną tego języka. W uproszczeniu powtarza ona wykonywanie bloku (normalnie wciętych) instrukcji, dopóki test znajdujący się na górze zwraca wartość będącą prawdą. Nazywana jest „*pętlą*”, ponieważ sterowanie powraca ciągle do początku tej instrukcji, dopóki test nie zwróci fałszu. Kiedy tak się stanie, sterowanie przechodzi do instrukcji następującej po bloku `while`. W rezultacie ciało pętli wykonywane jest raz za razem, dopóki test znajdujący się w jej nagłówku zwraca prawdę. Jeśli test od początku będzie zwracał fałsz, ciało pętli nigdy nie zostanie wykonane.

Ogólny format

W najbardziej złożonej postaci instrukcja while składa się z wiersza nagłówka z wyrażeniem testowym, ciała zawierającego jedną lub większą liczbę wciętych instrukcji oraz opcjonalną część else wykonywaną, kiedy sterowanie opuszcza pętlę bez napotkania instrukcji break. Python powtarza analizowanie testu z nagłówka i wykonywanie instrukcji zagnieżdżonych w ciele pętli, dopóki test nie zwróci fałszu.

```
while <test>:  
    <instrukcje1>  
else:  
    <instrukcje2>  
  
# Test pętli  
# Ciało pętli  
# Opcjonalne else  
# Wykonane, jeśli pętli nie zakończyło break
```

Przykłady

By zilustrować działanie pętli while, przyjrzyjmy się kilku przykładom jej zastosowania. Pierwszy, składający się z instrukcji print zagnieżdżonej w pętli while, po prostu w nieskończoność wyświetla komunikat. Warto przypomnieć, że True to inaczej zapisana wersja liczby całkowitej 1 i zawsze oznacza wartość Boolean dla prawdy. Ponieważ test zawsze będzie zwracał prawdę, Python będzie wykonywał ciało pętli w nieskończoność — lub dopóki mu nie przerwiemy. Takie zachowanie nazywa się często *nieskończoną pętlą*.

```
>>> while True:  
...     print('Wpisz Ctrl+C, by mnie zatrzymać!')
```

Kolejny przykład odcina ostatni znak łańcucha, dopóki łańcuch ten nie stanie się pusty i tym samym nie będzie fałszem. Zazwyczaj obiekty testuje się w sposób bezpośredni, zamiast korzystać z bardziej rozwlekłego odpowiednika while `x != ''`. W dalszej części rozdziału zobaczymy inne sposoby bezpośredniego przechodzenia elementów w łańcuchu znaków za pomocą pętli for.

```
>>> x = 'mielonka'  
>>> while x:  
...     print(x, end=' ')  
...     x = x[1:]  
...  
mielonka ielonka elonka lonka onka nka ka a
```

Warto tutaj zwrócić uwagę na argument ze słowem kluczowym `end=' '`, który pozwala na wyświetlenie wszystkich danych wyjściowych w jednym wierszu. Wracając do rozdziału 11., można dowiedzieć się, dlaczego tak się dzieje. Poniższy kod odlicza z kolei wartości od a do b (bez samego b). Później zobaczymy, że to samo można w łatwiejszy sposób uzyskać za pomocą pętli for i wbudowanej funkcji range.

```
>>> a=0; b=10  
>>> while a < b:  
...     print(a, end=' ')  
...     a += 1  
...  
0 1 2 3 4 5 6 7 8 9
```

Wreszcie warto zwrócić uwagę na to, że w Pythonie nie ma instrukcji pętli typu „rób to, dopóki...”. Możemy jednak taką instrukcję symulować za pomocą testu i instrukcji break na końcu ciała pętli.

```
while True:  
    ...ciało pętli...  
    if exitTest(): break
```

By w pełni zrozumieć działanie tej struktury, musimy przejść do kolejnego podrozdziału, w którym dowiemy się więcej o instrukcjach `break`.

Instrukcje `break`, `continue`, `pass` oraz `else` w pętli

Skoro już zobaczyliśmy kilka przykładów działających pętli Pythona, czas przyjrzeć się dwóm prostym instrukcjom, które mają sens dopiero wtedy, gdy zagnieżdżone są w pętlach — mowa tu o instrukcjach `break` oraz `continue`. A skoro już mowa o osobliwościach, omówimy przy okazji również blok `else` z pętli, ponieważ przeplata się on z `break`. Zajmiemy się także pustym pojemnikiem instrukcji, czyli `pass` (który nie jest powiązany z pętlami per se, ale mieści się w ogólnej kategorii prostych instrukcji składających się z jednego słowa). W Pythonie:

`break`

Wychodzi z najbliższej obejmującej daną instrukcję pętli (omija całą instrukcję pętli).

`continue`

Przechodzi na góre najbliższej obejmującej daną instrukcję pętli (do jej wiersza nagłówka).

`pass`

Nic nie robi. Jest pustym pojemnikiem instrukcji.

Blok pętli else

Wykonywany jest wtedy (i tylko wtedy), gdy pętla kończy się normalnie — bez trafienia na instrukcję `break`.

Ogólny format pętli

Uwzględniając instrukcje `break` i `continue`, ogólny format pętli `while` będzie wyglądał następująco.

```
while <test1>:  
    <instrukcje1>  
    if <test2>: break  
    if <test3>: continue  
    else:  
        <instrukcje2>  
        # Wyjście z pętli, pominięcie reszty  
        # Przejście do góry pętli, do testu1  
        # Wykonywane, jeśli nie trafiliśmy na break
```

Instrukcje `break` i `continue` mogą się pojawiać w dowolnym miejscu ciała pętli `while` (lub `for`), jednak zazwyczaj umieszczane są jako zagnieżdżone w teście `if` i wykonują jakieś działanie w odpowiedzi na określony warunek.

W celu przekonania się, jak w praktyce łączą się ze sobą te instrukcje, warto przyjrzeć się kilku prostym przykładom.

Instrukcja `pass`

Na początek najprostsze. Instrukcja `pass` jest pojemnikiem bez działania, wykorzystywany wtedy, gdy składnia wymaga instrukcji, ale nie mamy nic użytecznego do powiedzenia. Często wykorzystywana jest do zapisywania w kodzie pustego ciała instrukcji złożonej. Jeśli na przykład

chcemy umieścić w kodzie nieskończoną pętlę, która przy każdym przejściu nic nie robi, możemy skorzystać z instrukcji `pass`.

```
while True: pass # Użyj Ctrl+C, by mnie zatrzymać!
```

Ponieważ ciało pętli jest pustą instrukcją, Python zacina się w tej pętli. Instrukcja `pass` jest w instrukcjach mniej więcej tym, czym `None` dla obiektów — jawnie wyrażonym niczym. Warto zauważyc, że w powyższym przykładzie ciało pętli `while` umieszczone jest w jednym wierszu z jej nagłówkiem, po dwukropku. Tak jak w przypadku instrukcji `if`, taki zapis działa wyłącznie wtedy, gdy ciało nie jest instrukcją złożoną.

Powyższy przykład w nieskończoność nic nie robi. Nie jest to raczej najbardziej użyteczny program w historii programowania w Pythonie (o ile nie wykorzystamy go do rozgrzania naszego laptopa w chłodny, zimowy dzień). Szczerze mówiąc, po prostu nie umiałem wymyślić lepszego przykładu z `pass` na tym etapie książki.

Później zobaczymy inne sytuacje, w których `pass` ma większy sens — na przykład w ignorowaniu wyjątków przechwyconych za pomocą instrukcji `try`, definiowaniu pustych obiektów `class` z atrybutami zachowującymi się jak struktury i rekordy z innych języków programowania. Instrukcja `pass` czasami oznacza w kodzie miejsce, które zostanie uzupełnione później, a także służy do tymczasowego wyróżnienia ciała funkcji.

```
def func1():
    pass # Wstawić tu później prawdziwy kod!

def func2():
    pass
```

Nie możemy pozostawić ciała funkcji pustego bez otrzymania błędu składni, dlatego zamiast tego wstawiamy do niego instrukcję `pass`.



Uwaga na temat wersji: Python 3.0 (ale wersja 2.6 już nie) pozwala na wpisywanie w kodzie elips w postaci `...` (dosłownie: trzech następujących po sobie kropek) we wszystkich miejscach, w których może się pojawić wyrażenie. Ponieważ elipsy same z siebie nic nie robią, mogą służyć jako alternatywa dla instrukcji `pass`, zwłaszcza w przypadku kodu, który chcemy uzupełnić później:

```
def func1():
    ...
def func2():
    ...

func1() # Po wywołaniu nic nie robi
```

Elipsy mogą się także pojawiać w tym samym wierszu co nagłówek instrukcji i można je wykorzystywać do inicjalizacji nazw zmiennych, jeśli nie jest wymagany ich określony typ:

```
def func1(): ...
def func2(): ...

>>> X = ...
>>> X
Ellipsis # Działają także w tym samym wierszu
```

Powyższy zapis jest nowością w Pythonie 3.0 (i wychodzi znacznie poza początkowe zastosowanie znaków `...` w rozszerzeniach związanych z wycinkami). Z czasem okaże się, czy stanie się na tyle popularny, by zagrozić pozycji `pass` oraz `None` w tych zastosowaniach.

Instrukcja continue

Instrukcja `continue` powoduje natychmiastowe przejście na górę pętli. Czasami pozwala nam uniknąć zagnieźdzania instrukcji. Poniższy przykład wykorzystuje instrukcję `continue` do pomijania nieparzystych liczb. Kod ten wyświetla wszystkie parzyste liczby mniejsze od 10 i większe od lub równe 0. Należy pamiętać, że 0 oznacza fałsz, a `%` jest operatorem reszty z dzielenia, dlatego pętla ta odlicza od 10 do 0, pomijając liczby niebędące wielokrotnościami 2 (wyświetla zatem 8 6 4 2 0).

```
x = 10
while x:
    x = x-1
    if x % 2 != 0: continue
    print(x, end=' ')
# Lub x -= 1
# Nieparzyste? Pomijamy!
```

Ponieważ `continue` powoduje przeskoczenie na górę pętli, nie musimy zagnieźdzać instrukcji `print` wewnętrz testu `if`. Do instrukcji `print` dochodzimy dopiero wtedy, gdy nie zostanie wykonana instrukcja `continue`. Jeśli brzmi to podobnie do formy `goto` z innych języków programowania, to dobrze — tak właśnie powinno być. W Pythonie nie ma instrukcji `goto`, jednak ponieważ `continue` pozwala na przeskoki w programie, wiele z ostrzeżeń dotyczących czytelności i możliwości utrzymywania kodu dotyczących `goto` ma zastosowanie również tutaj. Instrukcji `continue` powinno się używać oszczędnie, w szczególności na początku naszej przygody z Pythonem. Przykład wyżej mógłby być nieco jaśniejszy, gdyby instrukcja `print` została zagnieźdzona pod `if`.

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:
        print(x, end=' ')
# Parzyste? Wyświetlamy!
```

Instrukcja break

Instrukcja `break` powoduje natychmiastowe wyjście z pętli. Ponieważ kod następujący po niej w pętli nie zostanie wykonany, dzięki umieszczeniu w kodzie instrukcji `break` czasami można uniknąć zagnieźdzania. Poniżej znajduje się prosta pętla interaktywna (wariant większego przykładu omawianego w rozdziale 10.), której dane wejściowe uzyskuje się za pomocą funkcji `input` (w Pythonie 2.6 znanej pod nazwą `raw_input`) i która kończy się, kiedy użytkownik w odpowiedzi na żądanie podania imienia wpisze „stop”.

```
>>> while True:
...     name = input('Podaj imię:')
...     if name == 'stop': break
...     age = input('Podaj wiek: ')
...     print('Witaj,', name, '=>', int(age) ** 2)
...
Podaj imię:Amadeusz
Podaj wiek: 40
Witaj, Amadeusz => 1600
Podaj imię:Maurycy
Podaj wiek: 30
Witaj, Maurycy => 900
Podaj imię:stop
```

Warto zwrócić uwagę na sposób konwersji danych wejściowych z wiekiem na liczbę całkowitą przed podniesieniem ich do kwadratu. Jak pamiętamy, jest to konieczne, ponieważ funkcja

input zwraca dane od użytkownika w postaciłańcucha znaków. W rozdziale 35. zobaczymy, że input zwraca również wyjątek na końcu pliku (na przykład kiedy użytkownik skorzysta ze skrótu *Ctrl+Z* lub *Ctrl+D*). Jeśli będzie to mogło mieć wpływ na nasz kod, należy umieścić tę funkcję w instrukcjach try.

Instrukcja else

W połączeniu z częścią pętli else instrukcja break może często wyeliminować konieczność używania opcji (flag) statusu wyszukiwania z innych języków programowania. Poniższy fragment kodu ustala na przykład, czy dodatnia liczba całkowita y jest liczbą pierwszą, wyszukując czynniki większe od 1.

```
x = y // 2                                # Dla jakiegoś y > 1
while x > 1:
    if y % x == 0:                         # Reszta
        print(y, 'ma czynnik', x)
        break
    x -= 1
else:                                         # Normalne wyjście
    print(y, 'jest liczbą pierwszą')
```

Zamiast ustawiać opcję statusu, która ma być sprawdzana przy wyjściu z pętli, wystarczy wstawić instrukcję break w miejsce, gdzie odnaleziony zostaje czynnik. W ten sposób część else zakłada, że będzie wykonana tylko wtedy, jeśli żaden czynnik nie zostanie odnaleziony. Jeśli nie trafimy na break, liczba jest liczbą pierwszą.

Część else wykonywana jest również wtedy, gdy ciało pętli nigdy nie zostanie wykonane, gdyż w takim przypadku również nie wykonamy instrukcji break. W pętli while może się tak stać, jeśli test z nagłówka od początku będzie fałszem. W rezultacie w przykładzie tym otrzymamy komunikat „jest liczbą pierwszą”, jeśli x początkowo będzie mniejsze lub równe 1 (na przykład kiedy y ma wartość 2).



Przykład ten oblicza liczby pierwsze — mniej więcej. Liczby mniejsze od 2 nie są w ścisłej definicji matematycznej uważane za pierwsze. A jeśli się naprawdę przyczeć, kod ten nie będzie działał dla liczb ujemnych, za to zadziała dla liczb zmienno-przecinkowych bez miejsc dziesiętnych. Warto również zauważyć, że kod ten musi w Pythonie 3.0 wykorzystywać operator `//` zamiast `/` z uwagi na przejście `/` na „prawdziwe dzielenie” opisane w rozdziale 5. (początkowe dzielenie musi odciąć resztę, a nie ją zachować!). Osoby, które chcą poeksperymentować z tym kodem, powinny spojrzeć do ćwiczenia na końcu czwartej części książki, w którym opakowuje się ten kod w funkcję.

Więcej o części pętli else

Ponieważ część else jest unikalna dla Pythona, często jest również dla osób początkujących bardzo myląca. Udostępnia ona jawną składnię służącą do zapisu często występującego scenariusza. To struktura kodu pozwalająca na przechwytywanie innej drogi wyjścia z pętli bez ustawiania i sprawdzania warunków czy opcji statusu.

Załóżmy na przykład, że piszemy pętlę przeszukującą listę pod kątem wartości i po zakończeniu pętli musimy wiedzieć, czy wartość została odnaleziona. Takie zadanie można zapisać w kodzie w poniższy sposób.

```

found = False
while x and not found:
    if match(x[0]):                                # Wartość na początku?
        print('Ni')
        found = True
    else:
        x = x[1:]                                    # Odcięcie początku i powtóżenie
if not found:
    print('Nie znaleziono')

```

W powyższym kodzie inicjalizujemy, ustawiamy i później sprawdzamy opcję statusu w celu określenia, czy wyszukiwanie zakończyło się powodzeniem. Jest to poprawny kod Pythona, który działa. Jest to również rodzaj struktury, w której idealnie przydaje się część `else`. Poniżej widać jej odpowiednik.

```

while x:                                         # Wyjście, gdy x jest puste
    if match(x[0]):
        print('Ni')
        break                                     # Wyjście, ominięcie else
    x = x[1:]
else:
    print('Nie znaleziono')                      # Tylko wtedy, gdy wyczerpano x

```

Ta wersja jest bardziej zwięzła. Opcja statusu znika, a test `if` na końcu pętli zastąpiono za pomocą `else` (wyrównanego w pionie ze słowem `while`). Ponieważ instrukcja `break` w środku głównej części `while` powoduje wyjście z pętli i obejście `else`, rozwiązanie to jest bardziej ustrukturyzowanym sposobem przechwytczenia przypadku niepowodzenia wyszukiwania.

Niektóre osoby mogły zauważyc, że w poprzednim przykładzie część `else` można było zastąpić sprawdzeniem pustego `x` po pętli (na przykład `if not x:`). Choć w tym przykładzie jest to prawda, część `else` udostępnia jawną składnię dla tego rodzaju wzorca kodu (ewidentnie mamy tutaj do czynienia z kodem przechwytyującym niepowodzenie wyszukiwania), a jawnie sprawdzenie pustego obiektu może w niektórych przypadkach nie mieć zastosowania. Część `else` staje się nawet bardziej przydatna w połączeniu z pętlą `for` — tematem kolejnego podrozdziału — ponieważ iteracja po sekwencji pozostaje poza naszą kontrolą.

Pętle `for`

Pętla `for` jest w Pythonie uniwersalnym iteratorem po sekwencjach. Może przechodzić elementy w dowolnym obiekcie będącym uporządkowaną sekwencją. Instrukcja `for` działa na łańcuchach znaków, listach, krotkach, innych wbudowanych obiektach, po których można iterować, i nowych obiektach, które — jak zobaczymy później — można tworzyć za pomocą klas. Spotkałmy ją przelotnie przy okazji przedstawiania typów obiektów sekwencji; czas teraz omówić ją bardziej dokładnie.

Ogólny format

Pętla `for` w Pythonie rozpoczyna się od wiersza nagłówka określającego cel (lub cele) przypisania wraz z obiektem, który chcemy przechodzić. Po nagłówku znajduje się blok (normalnie wciętych) instrukcji, które chcemy powtórzyć.

```

for <cel> in <obiekt>:
    <instrukcje>
else:
    <instrukcje>                                # Przypisanie elementów obiektu do celu
                                                # Powtarzane ciało pętli: użycie celu
                                                # Jeśli nie trafiliśmy na break

```

Znaczenie emulacji pętli while z języka C

W części poświęconej instrukcjom wyrażeń w rozdziale 11. pisaliśmy, że Python nie pozwala instrukcjom takim, jak przypisanie, na pojawianie się w miejscach, w których oczekiwane jest wyrażenie. Oznacza to, że w Pythonie nie będzie działał poniższy wzorzec kodu często spotykany w języku C.

```
while ((x = next()) != NULL) {...przetwarzanie x...}
```

Przypisania z języka C zwracają przypisaną wartość, natomiast przypisania w Pythonie są po prostu instrukcjami, a nie wyrażeniami. Eliminuje to pewną klasę notorycznie popełnianych błędów z języka C (nie można w Pythonie przypadkowo wpisać `=`, kiedy tak naprawdę mamy na myśli `==`). Jeśli jednak zależy nam na podobnym zachowaniu, istnieją co najmniej trzy sposoby otrzymania tego samego efektu w Pythonie za pomocą pętli, bez osadzania przypisania w testach pętli. Można przenieść przypisanie do ciała pętli za pomocą `break`:

```
while True:  
    x = next()  
    if not x: break  
    ...przetwarzanie x...
```

lub przesunąć przypisanie do pętli z testami:

```
x = True  
while x:  
    x = next()  
    if x:  
        ...przetwarzanie x...
```

albo przesunąć pierwsze przypisanie poza pętlę:

```
x = next()  
while x:  
    ...przetwarzanie x...  
    x = next()
```

Z tych trzech wzorców kodu pierwszy może być przez niektórych uznawany za najmniej ustrukturyzowany, jednak wydaje się jednocześnie najprostszy i w praktyce jest najczęściej używany. Niektóre pętle z języka C może również zastąpić prosta pętla `for` Pythona.

Kiedy Python wykonuje pętlę `for`, jeden po drugim przypisuje elementy z obiektu sekwencji do celu i wykonuje dla każdego z nich ciało pętli. Ciało pętli zazwyczaj wykorzystuje cel przypisania do odniesienia się do bieżącego elementu sekwencji, tak jakby był on kursorem przechodzącym sekwencję.

Nazwa użyta jako cel przypisania w wierszu nagłówka jest zazwyczaj (nową) zmienną w zakresie, w którym tworzona jest instrukcja `for`. Nie ma w niej nic specjalnego — może ona nawet zostać zmieniona w ciele pętli, jednak automatycznie zostanie ustawiona na kolejny element sekwencji, kiedy sterowanie powróci znowu na góre pętli. Po pętli zmienna najczęściej nadal odnosi się do ostatniego przechodzonego elementu (i tym samym ostatniego elementu sekwencji), o ile pętla nie zakończyła się na instrukcji `break`.

Instrukcja `for` obsługuje również opcjonalny blok `else`, który działa dokładnie tak samo jak w pętli `while` — jest wykonywany wtedy, gdy pętla kończy się bez trafienia na instrukcję `break` (to znaczy przetworzone zostały wszystkie elementy sekwencji). Omówione wcześniej instrukcje `break` i `continue` również w pętlach `for` działają w ten sam sposób co w `while`. Pełny format pętli `for` może zatem zostać przedstawiony w następujący sposób.

```

for <cel> in <obiekt>:
    <instrukcje>
    if <test>: break
    if <test>: continue
else:
    <instrukcje>

```

Przypisanie elementów obiektu do celu
Wyjście z pętli, pominięcie else
Przejście na górę pętli
Jeśli nie trafiliśmy na break

Przykłady

By zobaczyć, jak pętle `for` zachowują się w praktyce, wpiszmy teraz kilka przykładów do sesji interaktywnej.

Podstawowe zastosowanie

Jak wspomniano wcześniej, pętla `for` może przechodzić dowolny obiekt sekwencji. W naszym pierwszym przykładzie przypiszymy na przykład zmienną `x` do każdego z trzech elementów listy po kolej, od lewej do prawej strony. Dla każdego z nich zostanie wykonana instrukcja `print`. Wewnątrz instrukcji `print` (w ciele pętli) zmienna `x` odnosi się do bieżącego elementu listy.

```

>>> for x in ["mielonka", "jajka", "szynka"]:
...     print(x, end=' ')
...
mielonka jajka szynka

```

Kolejne dwa przykłady obliczają sumę i iloczyn wszystkich elementów listy. W dalszej części tego rozdziału i książki spotkamy się z narzędziami automatycznie stosującymi operacje, takie jak `+` oraz `*`, do elementów listy. Zazwyczaj jest to tak samo proste jak użycie pętli `for`.

```

>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24

```

Inne typy danych

W połączeniu z pętlą `for` można użyć dowolnego typu sekwencji, ponieważ narzędzie to jest uniwersalne. Pętle `for` działają na przykład na łańcuchach znaków i krotkach.

```

>>> S = "drwal"
>>> T = ("i", "jestem", "git")

>>> for x in S: print(x, end=' ')           # Iteracja po łańcuchu znaków
...
d r w a l

>>> for x in T: print(x, end=' ')           # Iteracja po krotce
...
i jestem git

```

Tak naprawdę, jak zobaczymy w kolejnym rozdziale przy okazji omawiania obiektów, na których można wykonywać iterację, pętle `for` działają nawet na pewnych obiektach, które nie są sekwencjami — w tym na plikach i słownikach!

Przypisanie krotek w pętli for

Jeśli wykonujemy iterację na sekwencji krotek, sam cel pętli może być *krotką* celów. To kolejny przykład działania mechanizmu przypisania rozpakowującego krotki, omówionego w rozdziale 11. Należy pamiętać, że pętla `for` przypisuje elementy obiektu sekwencji do celu, a przypisanie wszędzie działa tak samo.

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                                # Przypisanie krotek
...     print(a, b)
...
1 2
3 4
5 6
```

W powyższym kodzie pierwsze przejście pętli odpowiada zapisowi $(a, b) = (1, 2)$, drugie — $(a, b) = (3, 4)$ i tak dalej. W rezultacie z każdą iteracją automatycznie wypakowujemy bieżącą krotkę.

Ta forma wykorzystywana jest często w połączeniu z wywołaniem `zip`, z którym spotkamy się w dalszej części niniejszego rozdziału przy okazji implementacji przechodzenia równoległego. Regularnie pojawia się także w Pythonie w połączeniu z bazami danych SQL, w których tablice wyników zapytań zwarcane są w postaci sekwencji sekwencji, podobnych do wykorzystanej tutaj listy. Zewnętrzna lista to tabela bazy danych, zagnieżdżone krotki to wiersze wewnętrznej tej tabeli, a przypisanie krotek powoduje ekstrakcję kolumn.

Krotki w pętlach `for` przydają się także do iteracji po kluczach i wartościach słowników za pomocą metody `items` zastępującej pętlę po kluczach i indeksowanie w celu ręcznego pobrania wartości:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...     print(key, '=>', D[key])                  # Użycie iteratora po kluczach słownika i indeksowania
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...     print(key, '=>', value)                   # Iteracja po kluczach i wartościach
...
a => 1
c => 3
b => 2
```

Należy koniecznie zauważyc, że przypisanie krotek w pętlach `for` nie jest żadnym przypadkiem specjalnym. Każdy cel przypisania z punktu widzenia składni zadziała po słowie `for`. Zawsze możemy jednak wykonać przypisanie ręczne wewnątrz pętli w celu rozpakowania:

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both
...     print(a, b)                                # Odpowiednik z przypisaniem ręcznym
...
1 2
3 4
5 6
```

```
1 2  
3 4  
5 6
```

Użycie krotki w nagłówku pętli oszczędza nam dodatkowego kroku przy iteracji po sekwencjach sekwencji. Zgodnie z informacjami z rozdziału 11. w instrukcji `for` można w ten sposób automatycznie rozpakowywać nawet struktury *zagnieżdzone*:

```
>>> ((a, b), c) = ((1, 2), 3)          # Sekwencje zagnieżdzone też działają  
>>> a, b, c  
(1, 2, 3)  
  
>>> for ((a, b), c) in [(1, 2), 3], ((4, 5), 6]): print(a, b, c)  
...  
1 2 3  
4 5 6
```

Nie jest to jednak żaden przypadek specjalny — pętla `for` po prostu wykonuje przypisanie, które wykonaliśmy tuż przed nią, z każdą iteracją. W ten sposób można rozpakować dowolną strukturę sekwencji, właśnie dlatego, że *przypisanie sekwencji* jest tak uniwersalne.

```
>>> for ((a, b), c) in [[1, 2], 3], ['XY', 6]: print(a, b, c)  
...  
1 2 3  
X Y 6
```

Rozszerzone przypisanie sekwencji w pętlach for w Pythonie 3.0

Tak naprawdę, ponieważ zmienna pętli w pętli `for` może być dowolnym celem przypisania, możemy tutaj wykorzystać także składnię rozszerzonego przypisania rozpakowującego sekwencję z Pythona 3.0 w celu dokonania ekstrakcji elementów i części sekwencji wewnętrz sekwencji. W rzeczywistości to także nie jest żaden przypadek specjalny, a po prostu nowa forma przypisania z wersji 3.0 (omówiona w rozdziale 11.). Ponieważ działa ona w instrukcjach przypisania, automatycznie działa także w pętlach `for`.

Rozważmy raz jeszcze kod z przypisaniem krotki przedstawiony wyżej. Krotka wartości przypisywana jest do krotki nazw z każdą iteracją, dokładnie tak samo jak w prostej instrukcji przypisania:

```
>>> a, b, c = (1, 2, 3)          # Przypisanie krotki  
>>> a, b, c  
(1, 2, 3)  
  
>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:      # Wykorzystane w pętli for  
...     print(a, b, c)  
...  
1 2 3  
4 5 6
```

Ponieważ sekwencję można przypisać do bardziej uniwersalnego zbioru nazw za pomocą nazwy ze znakiem * służącej do zbierania większej liczby elementów, w Pythonie 3.0 możemy wykorzystać tę samą składnię do ekstrakcji części zagnieżdżonych sekwencji w pętli `for`:

```
>>> a, *b, c = (1, 2, 3, 4)          # Rozszerzone przypisanie sekwencji  
>>> a, b, c  
(1, [2, 3], 4)  
  
>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
...     print(a, b, c)  
...
```

```
1 [2, 3] 4
5 [6, 7] 8
```

W praktyce rozwiązywanie to można wykorzystać do wybrania kilku kolumn z wierszy danych reprezentowanych jako zagnieżdżone sekwencje. W Pythonie 2.X nazwy ze znakiem * nie są dozwolone, jednak podobny efekt można uzyskać za pomocą wycinków. Jedyna różnica polega na tym, że wycinek zwraca wynik specyficzny dla typu, natomiast do nazw ze znakiem * zawsze przypisywane są listy.

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]: # Ręczne wycinki z Pythona 2.6
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
...
1 (2, 3) 4
5 (6, 7) 8
```

Więcej informacji na temat tej formy przypisania można znaleźć w rozdziale 11.

Zagnieżdżone pętle for

Przyjrzyjmy się teraz pętli `for` nieco bardziej wyszukanej od zaprezentowanych dotychczas. Kolejny przykład ilustruje zastosowanie w pętli `for` części `else`, a także zagnieżdżanie instrukcji. Mając podaną listę obiektów (`items`) i listę kluczy (`tests`), kod ten wyszukuje w liście obiektów każdego klucza i zgłasza rezultat tego działania.

```
>>> items = ["aaa", 111, (4, 5), 2.01]      # Zbiór obiektów
>>> tests = [(4, 5), 3.14]                  # Szukane klucze
>>>
>>> for key in tests:                      # Dla wszystkich kluczy
...     for item in items:                   # Dla wszystkich elementów
...         if item == key:                  # Sprawdzenie dopasowania
...             print(key, "znaleziono")
...             break
...     else:
...         print(key, "nie znaleziono!")
...
(4, 5) znaleziono
3.14 nie znaleziono!
```

Ponieważ zagnieżdżona instrukcja `if` wykonuje instrukcję `break`, kiedy klucz zostaje dopasowany, część `else` zakłada, że jeśli się do niej dojdzie, to wyszukiwanie nie powiodło się. Warto zwrócić uwagę na zagnieżdżenie. Kiedy kod ten zostaje wykonany, w tym samym czasie wykonywane są dwie pętle. Zewnętrzna przegląda listę kluczy, natomiast wewnętrzna przeszukuje listę obiektów dla każdego z kluczy. Zagnieżdżenie części `else` jest kwestią kluczową — jest ona wcięta na tę samą odległość co wiersz nagłówka wewnętrznej pętli `for`, dzięki czemu jest powiązana właśnie z nią (a nie z `if` czy zewnętrzną pętlą `for`).

Warto zauważyć, że przykład ten jest łatwiejszy do zapisania, jeśli w teście przynależności wykorzystamy operator `in`. Ponieważ `in` w niejawnym sposobie przeszukuje obiekt, szukając dopasowania (przynajmniej logicznego), może zastąpić wewnętrzną pętlę.

```
>>> for key in tests:                      # Dla wszystkich kluczy
...     if key in items:                      # Niech Python sam sprawdza dopasowanie
...         print(key, "znaleziono")
...     else:
...         print(key, "nie znaleziono!")
...
(4, 5) znaleziono
3.14 nie znaleziono!
```

Na ogół dobrze jest pozwolić Pythonowi na wykonanie jak największej ilości samodzielnej pracy — jak w powyższym rozwiązaniu — ze względu na zwięzłość i wydajność kodu.

Kolejny przykład wykonuje typowe dla instrukcji `for` zadanie dotyczące struktur danych — zebranie powtarzających się elementów z dwóch sekwencji (łańcuchów znaków). Przypomina ono prostą procedurę części wspólnej zbiorów. Po wykonaniu pętli zmieniona res będzie się odnosiła do listy zawierającej wszystkie elementy znalezione w łańcuchach `seq1` i `seq2`.

```
>>> seq1 = "mielonka"
>>> seq2 = "biedronka"
>>>
>>> res = []                                # Na początek pusta lista
>>> for x in seq1:                          # Przeszukanie pierwszej sekwencji
...     if x in seq2:                      # Powtarzający się element?
...         res.append(x)                   # Dodanie na końcu listy wyników
...
>>> res
['i', 'e', 'o', 'n', 'k', 'a']
```

Niestety, kod ten będzie działał wyłącznie na dwóch określonych w nim zmiennych — `seq1` i `seq2`. Byłoby dobrze, gdyby tę pętlę można było w jakiś sposób uogólnić, tak by stała się narzędziem, z którego można skorzystać więcej niż tylko jeden raz. Jak zobaczymy, taki pomysł spowodował pojawienie się *funkcji* — koncepcji będącej tematem kolejnej części książki.

Znaczenie skanerów plików

Pętle przydają się wszędzie tam, gdzie chcemy powtórzyć jakąś operację lub przetworzyć coś więcej niż jeden raz. Ponieważ pliki zawierają wiele znaków i wierszy, są jednym z bardziej typowych przypadków użycia dla pętli. By załadować całą zawartość pliku do łańcucha znaków za jednym razem, wystarczy wywołać metodę `read` obiektu pliku.

```
file = open('test.txt', 'r')                  # Wczytanie zawartości do łańcucha znaków
print(file.read())
```

By jednak załadować plik w mniejszych częściach, często tworzy się albo pętlę `while` z instrukcją `break` na końcu pliku, albo pętlę `for`. By wczytać plik znak po znaku, przyda się jeden z poniższych kodów.

```
file = open('test.txt')
while True:
    char = file.read(1)                      # Wczytanie znak po znaku
    if not char: break
    print(char)

for char in open('test.txt').read():
    print(char)
```

Powyższa pętla `for` również przetwarza każdy znak po kolei, jednak ładuje cały plik do pamięci za jednym razem (i zakłada, że on się tam zmieści!). By zamiast tego wczytać plik wiersz po wierszu lub blok po bloku, można skorzystać z pętli `while` i jednego z poniższych rozwiązań.

```
file = open('test.txt')
while True:
    line = file.readline()                  # Wczytanie wiersz po wierszu
    if not line: break
    print(line, end='')                   # Wiersz zawiera już \n
```

```
file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)                      # Wczytanie bajtowych fragmentów — do 10 bajtów
    if not chunk: break
    print(chunk)
```

Dane binarne zazwyczaj wczytuje się w blokach. By wczytać tekst wiersz po wierszu, łatwiej jest jednak napisać kod z pętlą `for`, którego dodatkową zaletą jest to, że będzie się on również szybciej wykonywał.

```
for line in open('test.txt').readlines():
    print(line, end='')

for line in open('test.txt'):      # Użycie iteratorów — najlepszego trybu dla tekstowych danych
    # wejściowych
    print(line, end='')
```

Metoda obiektu pliku `readlines` ładuje cały plik naraz do listy łańcuchów znaków poszczególnych wierszy. Ostatni z powyższych przykładów wykorzystuje iterator plików do automatycznego wczytywania po jednym wierszu z każdą iteracją pętli (iteratory omówione są szczegółowo w rozdziale 14.). Więcej informacji na temat zastosowanych tutaj wywołań można znaleźć w dokumentacji biblioteki standardowej Pythona. Ostatni z przykładów jest ogólnie najlepszym rozwiązaniem w przypadku plików tekstowych — jest prosty, działa dla dowolnie dużych plików i nie ładuje całego pliku naraz do pamięci. Wersja z iteratorem może być najszybsza, jednak różnica w wydajności wejścia-wyjścia nie jest tak oczywista w Pythonie 3.0.

W kodzie napisanym w Pythonie 2.X można się spotkać z zastąpieniem nazwy `open` słowem `file`. Starsza metoda obiektu pliku `xreadlines` jest tam też wykorzystywana w celu uzyskania tego samego efektu, jaki daje automatyczny iterator wierszy pliku (metoda ta przypomina `readlines`, jednak nie ładuje całego pliku naraz do pamięci). Zarówno `file`, jak i `xreadlines` zostały usunięte w Pythonie 3.0, ponieważ są zbędne. Nie powinno się ich także używać w Pythonie 2.6, choć mogą się pojawić w jakimś starszym kodzie czy zasobach. Więcej informacji na temat wczytywania plików można znaleźć w rozdziale 36. Zobaczmy tam, że pliki tekstowe i binarne mają w wersji 3.0 nieco inną semantykę.

Techniki tworzenia pętli

Pętla `for` zalicza się do pętli w stylu liczników. Jest łatwiejsza do zapisania i szybsza do wykonania od `while`, dlatego jest pierwszym narzędziem, po które powinniśmy sięgać, kiedy musimy przejść jakąś sekwencję. Istnieją jednak również sytuacje, w których będzie nam potrzebna bardziej wyspecjalizowana iteracja. Co należy na przykład zrobić, kiedy musimy przejść co drugi czy co trzeci element listy lub po drodze listę tę zmodyfikować? Co z przejściem większej liczby sekwencji równolegle, w tej samej pętli `for`?

Tego typu unikalne iteracje można zawsze zapisać w kodzie z wykorzystaniem pętli `while` i ręcznego indeksowania, jednak Python udostępnia dwie funkcje wbudowane, które pozwalają na wykonanie wyspecjalizowanej pętli za pomocą `for`.

- Wbudowana funkcja `range` zwraca serię kolejnych, coraz większych liczb całkowitych, które mogą zostać wykorzystane jako indeksy dla pętli `for`.
- Wbudowana funkcja `zip` zwraca serię krotek równoległych elementów, która może zostać wykorzystana do przechodzenia wielu sekwencji w pętli `for`.

Ponieważ pętle `for` zazwyczaj działają szybciej od pętli z licznikami opartych na `while`, użycie `for` w połączeniu z odpowiednimi narzędziami, takimi jak powyższe, zawsze będzie działało na naszą korzyść. Przyjrzymy się kolejno tym wbudowanym funkcjom.

Pętle liczników — `while` i `range`

Funkcja `range` jest tak naprawdę narzędziem, które można zastosować w wielu różnych kontekstach. Choć najczęściej wykorzystywana jest do generowania indeksów dla pętli `for`, można jej użyć w każdym miejscu, w którym potrzebna jest nam lista liczb całkowitych. W Pythonie 3.0 `range` jest *iteratorem* generującym elementy na żądanie, dlatego musimy opanować tę funkcję w wywołanie `list` w celu wyświetlenia wszystkich wyników naraz (więcej informacji na temat iteratorów znajduje się w rozdziale 14.).

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

Z jednym argumentem funkcja `range` generuje listę liczb całkowitych od zera do wartości argumentu (ale bez niej samej). Jeśli przekażemy jej dwa argumenty, pierwszy uznawany jest za dolną granicę. Opcjonalny trzeci argument jest *krokiem*. Jeśli się go użyje, Python dodaje krok do każdej kolejnej liczby całkowitej wyniku (krok ma wartość domyślną 1). Zakresy mogą również być niedodatnie i nierosnące, o ile jest nam to do czegoś potrzebne.

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

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

Choć takie wyniki funkcji `range` mogą same w sobie być użyteczne, najbardziej przydają się wewnętrz pętli `for`. Po pierwsze, umożliwiają łatwe powtórzenie działania określonej liczby razy. By na przykład wyświetlić trzy wiersze, wystarczy wykorzystać `range` do wygenerowania odpowiedniej liczby liczb całkowitych. Pętle `for` automatycznie wymuszają wyniki z `range` w Pythonie 3.0, dlatego nie musimy tutaj używać wywołania `list`.

```
>>> for i in range(3):
...     print(i, 'Python')
...
0 Python
1 Python
2 Python
```

Funkcja `range` jest często wykorzystywana do pośredniej iteracji po sekwencji. Najłatwiejszym i najszybszym sposobem wyczerpującego przejścia sekwencji jest zawsze proste `for`, gdyż Python większość szczegółów zrobi za nas.

```
>>> X = 'mielonka'
>>> for item in X: print(item, end=' ')
# Prosta iteracja
...
m i e l o n k a
```

Wewnętrznie pętla `for` obsługuje szczegółы iteracji automatycznie, kiedy używa się jej w ten sposób. Jeśli naprawdę musimy wziąć logikę indeksowania w swoje ręce, możemy to zrobić za pomocą pętli `while`.

```
>>> i = 0
>>> while i < len(X):                                # Iteracja z pętlą while
...     print(X[i], end=' ')
```

```
...     i += 1
...
m i e l o n k a
```

Ręczne indeksowanie można jednak również wykonać w pętli `for`, jeśli użyjemy `range` do wygenerowania listy indeksów, po których będziemy iterować. To proces składający się z kilku kroków, ale wystarczy on do generowania wartości przesunięć, a nie elementów z tymi wartościami przesunięcia:

```
>>> X
'mielonka'
>>> len(X)
8
>>> list(range(len(X)))
[0, 1, 2, 3, 4, 5, 6, 7]
>>>
>>> for i in range(len(X)): print(X[i], end=' ') # Ręczne indeksowanie za pomocą pętli for
...
m i e l o n k a
```

Warto zwrócić uwagę na to, że powyższy przykład przechodzi listę *wartości przesunięcia* dla `X`, a nie prawdziwych *elementów* `X`. By pobrać każdy z elementów, musimy z powrotem zindeksować `X` wewnętrz pętli.

Przechodzenie niewyczerpujące — `range` i wycinki

Ostatni omówiony przykład działa, jednak nie jest to najszybciej działające rozwiązań. Wiąże się z nim również więcej pracy, niż jest to konieczne. O ile nie mamy specjalnych wymagań związanych z indeksowaniem, zwykle lepiej będzie użyć w Pythonie prostej postaci pętli `for`. Zawsze, kiedy to możliwe, należy użyć `for` zamiast `while` i wywołanie `range` w pętli `for` należy traktować tylko jako ostatnią deskę ratunku. Prostsze rozwiązanie zawsze będzie lepsze.

```
>>> for item in X: print(item)                      # Prosta iteracja
...
```

Wzorzec kodu wykorzystany w poprzednim przykładzie pozwala nam jednak na nieco bardziej wyspecjalizowane przechodzenie obiektów — na przykład pomijanie niektórych po drodze.

```
>>> S = 'abcdefghijklmnopqrstuvwxyz'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

W kodzie tym przetwarzamy *co drugi* element łańcucha `S` dzięki przejęciu listy wygenerowanej przez funkcję `range`. By przetworzyć co trzeci element, wystarczy zmienić trzeci argument funkcji `range` na 3. W rezultacie takie wykorzystanie funkcji `range` pozwala na przeskakiwanie elementów sekwencji w pętlach przy jednoczesnym zachowaniu prostoty związanej z konstrukcją pętli `for`.

Mimo to nie jest to obecnie najbardziej polecana technika w Pythonie. Jeśli naprawdę chcemy pomijać niektóre elementy sekwencji, rozszerzona forma *wyrażenia z wycinkiem* z trzema granicami (zaprezentowana w rozdziale 7.) umożliwia osiągnięcie tego samego w łatwiejszy sposób. By przetworzyć co drugi znak z łańcucha `S`, wystarczy wykonać wycinek z krokiem o wartości 2.

```
>>> S = 'abcdefghijklk'  
>>> for c in S[::-2]: print(c, end=' ')  
...  
a c e g i k
```

Rezultat będzie taki sam, jednak kod ten łatwiej jest zapisać, a także odczytać. Jedyną zaletą wykorzystania w tym kontekście range jest to, że rozwiązanie to w wersji 3.0 nie kopiuje łańcucha znaków i nie tworzy listy. W przypadku bardzo dużych łańcuchów znaków może to pozwolić zaoszczędzić nieco pamięci.

Modyfikacja list — range

Inną często spotykaną sytuacją, w której można użyć kombinacji range i for, jest pętla modyfikująca przechodzoną listę. Założymy na przykład, że z jakiegoś powodu musimy dodać 1 do każdego elementu listy. Można spróbować tego dokonać za pomocą prostej pętli for, jednak wynik niekoniecznie będzie taki, jaki byśmy chcieli.

```
>>> L = [1, 2, 3, 4, 5]  
  
>>> for x in L:  
...     x += 1  
...  
>>> L  
[1, 2, 3, 4, 5]  
>>> x  
6
```

Takie rozwiązanie nie działa — modyfikowana jest zmienna pętli x, a nie lista L. Przyczyny takiego stanu rzeczy są dość subtelne. Przy każdym przejściu pętli zmienna x odnosi się do kolejnej liczby całkowitej pobranej z listy. W pierwszej iteracji x jest na przykład liczbą całkowitą 1. W kolejnej iteracji ciało pętli ustawia x na inny obiekt — liczbę całkowitą 2 — jednak nie uaktualnia w miejscu listy, z której oryginalnie pochodziła liczba 1.

By naprawić zmodyfikowanie listy w czasie jej przechodzenia, musimy użyć indeksów, które pozwolą na przypisanie uaktualnionej wartości do każdej pozycji listy w miarę jej przechodzenia. Kombinacja funkcji range i len pozwala uzyskać pożądane indeksy.

```
>>> L = [1, 2, 3, 4, 5]  
  
>>> for i in range(len(L)):  
...     L[i] += 1  
...  
>>> L  
[2, 3, 4, 5, 6]
```

Taki kod pozwala na modyfikację listy w miarę przechodzenia pętli. Nie da się uzyskać tego samego za pomocą prostej pętli w stylu for x in L:, ponieważ taka pętla przechodzi same elementy listy, a nie ich pozycje. Co jednak z odpowiednikiem tego rozwiązania wykorzystującym pętlę while? Taka pętla będzie wymagać od nas nieco więcej pracy i naprawdę pozwala na modyfikację listy w miarę przechodzenia pętli.

```
>>> i = 0  
>>> while i < len(L):  
...     L[i] += 1  
...     i += 1  
...  
>>> L  
[3, 4, 5, 6, 7]
```

I tutaj rozwiązanie z funkcją `range` może nie być idealne. Wyrażenie listy składanej w postaci:

```
[x+1 for x in L]
```

wykonałoby prawie to samo, jednak bez modyfikacji oryginalnej listy w miejscu (moglibyśmy przypisać nowy obiekt listy z wyrażenia z powrotem do `L`, jednak nie uaktualniłoby to innych referencji do oryginalnej listy). Ponieważ listy składane są kluczową koncepcją związaną z pętlami, zostawimy pełne omówienie tego zagadnienia na kolejny rozdział.

Przechodzenie równolegle — `zip` oraz `map`

Jak widzieliśmy, funkcja wbudowana `range` pozwala na przechodzenie sekwencji za pomocą pętli `for` w sposób niewyczerpujący. W podobny sposób wbudowana funkcja `zip` pozwala na wykorzystanie pętli `for` do równoległego przejścia większej liczby sekwencji. W prostej operacji funkcja ta przyjmuje jako argument jedną lub większą liczbę sekwencji i zwraca serię krotek łączących w pary równolegle elementy z tych sekwencji. Założymy na przykład, że pracujemy z dwoma listami, jak w kodzie poniżej.

```
>>> L1 = [1, 2, 3, 4]
>>> L2 = [5, 6, 7, 8]
```

By połączyć elementy z tych dwóch list, możemy wykorzystać funkcję `zip` do utworzenia listy par krotek. Tak jak `range`, `zip` jest w Pythonie 3.0 obiektem, na którym można wykonywać iterację, dlatego musimy opakować tę funkcję w wywołanie `list` w celu wyświetlenia wszystkich jej wyników naraz — więcej informacji o iteratorach znajduje się w kolejnym rozdziale.

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))                                     # list() wymagane w Pythonie 3.0, w 2.6 nie
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Taki wynik może być przydatny również w innych kontekstach, natomiast w połączeniu z `for` obsługuje iteracje równolegle.

```
>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Powyżej przechodzimy wyniki wywołania funkcji `zip`, czyli pary elementów pobranych z dwóch list. Warto zauważyć, że ta pętla `for` ponownie wykorzystuje przypisanie krotek (w postaci, z jaką spotkaliśmy się już wcześniej) w celu rozpakowania każdej krotki w wyniku `zip`. Za pierwszym razem wygląda to tak, jakbyśmy wykonali instrukcję przypisania `(x, y) = (1, 5)`.

Rezultat będzie taki, że w pętli przechodzimy zarówno listę `L1`, jak i listę `L2`. Podobny efekt moglibyśmy osiągnąć za pomocą pętli `while`, która obsługuje ręczne indeksowanie, jednak wymagałoby to więcej kodu i prawdopodobnie działałoby wolniej od rozwiązania z `for` oraz `zip`.

Funkcja `zip` jest tak naprawdę bardziej uniwersalna, niż mogłoby to wynikać z powyższego przykładu. Przyjmuje ona dowolny typ sekwencji (a tak naprawdę obiekt, na którym można wykonać iterację — w tym pliki) i może przyjąć więcej niż dwa argumenty. W przypadku trzech

argumentów, jak w poniższym przykładzie, buduje listę krotek trzyelementowych z elementami z każdej z sekwencji, odzwierciedlając w ten sposób kolumny (otrzymujemy krotkę N -argumentową dla N argumentów).

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Co więcej, kiedy długość argumentów różni się, funkcja `zip` odcina wynikowe krotki do długości najkrótszej sekwencji. W poniższym przykładzie łączymy ze sobą dwa łańcuchy w celu równoległego wybrania znaków, jednak wynik składa się jedynie z tylu krotek, jaką długość ma najkrótsza sekwencja:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Równoznaczność funkcji `map` w Pythonie 2.6

W Pythonie 2.X powiązana z `zip` funkcja wbudowana `map` w podobny sposób łączy w pary elementy sekwencji, jednak dopełnia krótsze sekwencje za pomocą `None`, jeśli długość argumentów jest różna, zamiast odcinać wynik do długości krótszej sekwencji:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'

>>> map(None, S1, S2)                                     # Tylko w Pythonie 2.X
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

Powyższy przykład używa zdegenerowanej formy funkcji `map`, która w Pythonie 3.0 nie jest już obsługiwana. Normalnie `map` przyjmuje w postaci argumentów funkcję oraz jedną lub większą liczbę sekwencji i zbiera wynik wywołania funkcji z równoległymi elementami pobranymi z sekwencji. Funkcję `map` będziemy omawiać bardziej szczegółowo w rozdziałach 19. oraz 20., natomiast poniższy przykład odwzorowuje wbudowaną funkcję `ord` na każdy element łańcucha znaków i zbiera wyniki. Tak jak `zip`, funkcja `map` jest w Pythonie 3.0 generatorem wartości, dla tego konieczne jest przekazanie jej do `list` w celu zebrania wszystkich wyników naraz.

```
>>> list(map(ord, 'mielonka'))
[109, 105, 101, 108, 111, 110, 107, 97]
```

Kod ten działa tak samo jak poniższa instrukcja pętli, jednak często jest szybszy.

```
>>> res = []
>>> for c in 'mielonka': res.append(ord(c))
>>> res
[109, 105, 101, 108, 111, 110, 107, 97]
```



Uwaga na temat wersji: Zdegenerowana postać `map` wykorzystująca argument funkcji `None` nie jest już obsługiwana w Pythonie 3.0, ponieważ w dużej mierze pokrywa się z `zip` (a do tego, szczerze mówiąc, nieco odbiegała od celu zastosowania funkcji `map`). W wersji 3.0 należy albo skorzystać z funkcji `zip`, albo samodzielnie napisać kod pętli dopełniający wyniki. Zobaczymy, jak można to zrobić, w rozdziale 20., po tym, jak będziemy mieli okazję zapoznać się z pewnymi dodatkowymi zagadnieniami związonymi z iteracją.

Tworzenie słowników za pomocą funkcji zip

W rozdziale 8. zasugerowałem, że zaprezentowane wyżej wywołanie `zip` może się również przydać do generowania słowników, kiedy zbiory kluczy i wartości muszą być obliczane w czasie wykonania. Skoro już zapoznaliśmy się z funkcją `zip`, czas wyjaśnić, co ma ona wspólnego z tworzeniem słowników. Jak powiedzieliśmy wcześniej, słownik zawsze można utworzyć za pomocą literatu lub przypisania wartości do kluczy.

```
>>> D1 = {'mielonka':1, 'jajka':3, 'tost':5}
>>> D1
{'tost': 5, 'jajka': 3, 'mielonka': 1}

>>> D1 = {}
>>> D1['mielonka'] = 1
>>> D1['jajka'] = 3
>>> D1['tost'] = 5
```

Co jednak zrobić, kiedy program uzyskuje klucze i wartości słownika w postaci *list* w czasie wykonywania, już po utworzeniu skryptu? Powiedzmy na przykład, że mamy poniższe listy kluczy i wartości.

```
>>> keys = ['mielonka', 'jajka', 'tost']
>>> vals = [1, 3, 5]
```

Jedną z możliwości przekształcenia ich w słownik będzie zastosowanie na listach funkcji `zip` i równolegle przejdzie ich za pomocą pętli `for`.

```
>>> list(zip(keys, vals))
[('mielonka', 1), ('jajka', 3), ('tost', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'tost': 5, 'jajka': 3, 'mielonka': 1}
```

Okazuje się jednak, że od Pythona 2.2 można całkowicie pominąć pętlę `for` i po prostu przekazać połączone w pary za pomocą `zip` listy kluczy i wartości do wywołania wbudowanego konstruktora słownika `dict`.

```
>>> keys = ['mielonka', 'jajka', 'tost']
>>> vals = [1, 3, 5]

>>> D3 = dict(zip(keys, vals))
>>> D3
{'tost': 5, 'jajka': 3, 'mielonka': 1}
```

Wbudowana nazwa `dict` jest tak naprawdę nazwą typu w Pythonie (o nazwach typów i ich podklasach dowiemy się więcej w rozdziale 31.). Jej wywołanie powoduje coś w stylu konwersji z listy na słownik, choć tak naprawdę jest to żądanie konstrukcji obiektu. W kolejnym rozdziale zapoznamy się z podobną, jednak bogatszą koncepcją *list składanych*, które budują listy w jednym wyrażeniu. Powrócimy także do *słowników składanych* z Pythona 3.0, stanowiących alternatywę dla wywołania konstruktora `dict` dla połączonych za pomocą `zip` par kluczy i wartości.

Generowanie wartości przesunięcia i elementów — enumerate

Wcześniej omawialiśmy wykorzystanie funkcji `range` do wygenerowania wartości przesunięcia elementów łańcucha znaków, a nie samych elementów znajdujących się na tych pozycjach. W niektórych programach potrzebne nam będą jednak obie wartości — używany element i przy okazji jego wartość przesunięcia. Zazwyczaj takie coś rozwiązywano za pomocą prostej pętli `for`, która przechowywała również licznik bieżącej wartości przesunięcia elementu.

```
>>> S = 'mielonka'
>>> offset = 0
>>> for item in S:
...     print(item, 'występuje na pozycji przesunięcia', offset)
...     offset += 1
...
m występuje na pozycji przesunięcia 0
i występuje na pozycji przesunięcia 1
e występuje na pozycji przesunięcia 2
l występuje na pozycji przesunięcia 3
o występuje na pozycji przesunięcia 4
n występuje na pozycji przesunięcia 5
k występuje na pozycji przesunięcia 6
a występuje na pozycji przesunięcia 7
```

To rozwiązanie działa, jednak w nowszych wersjach Pythona to samo robi za nas nowa funkcja wbudowana `enumerate`.

```
>>> S = 'mielonka'
>>> for (offset, item) in enumerate(S):
...     print(item, 'występuje na pozycji przesunięcia', offset)
...
m występuje na pozycji przesunięcia 0
i występuje na pozycji przesunięcia 1
e występuje na pozycji przesunięcia 2
l występuje na pozycji przesunięcia 3
o występuje na pozycji przesunięcia 4
n występuje na pozycji przesunięcia 5
k występuje na pozycji przesunięcia 6
a występuje na pozycji przesunięcia 7
```

Funkcja `enumerate` zwraca *obiekt generatora* — rodzaj obiektu obsługujący protokół iteracji, z którym zapoznamy się w kolejnym rozdziale i który omówimy bardziej szczegółowo w kolejnej części książki. W skrócie, obsługuje on metodę `__next__` wywoływaną za pomocą funkcji wbudowanej `next` i zwracającą za każdym przejściem pętli krotkę (`indeks, wartość`). Krotkę tę możemy rozpakować za pomocą przypisania krotek w pętli `for` (przypomina to użycie `zip`).

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x02765AA8>
>>> next(E)
(0, 'm')
>>> next(E)
(1, 'i')
>>> next(E)
(2, 'e')
```

Jak zawsze, normalnie nie widzimy tego wszystkiego, ponieważ konteksty iteracyjne — w tym listy składane, temat rozdziału 14. — wykonują protokół iteracji automatycznie.

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'i', 'ee', 'lll', 'oooo', 'nnnnn', 'kkkkkk', 'aaaaaaaa']
```

By w pełni zrozumieć zagadnienia związane z iteracją, takie jak `enumerate`, `zip` czy listy składane, musimy przejść do kolejnego rozdziału zawierającego ich omówienie z formalnego punktu widzenia.

Podsumowanie rozdziału

W niniejszym rozdziale zapoznaliśmy się z instrukcjami pętli Pythona, a także z pewnymi koncepcjami związanymi z pętlami w tym języku. Przyjrzaliśmy się instrukcjom `while` i `for`, a także zobaczyliśmy, jak działają powiązane z nimi części `else`. Omówiliśmy również instrukcje `break` i `continue`, które mają znaczenie jedynie wewnątrz pętli, a także poznaliśmy kilka wbudowanych narzędzi wykorzystywanych często w pętlach `for` — w tym `range`, `zip`, `map` oraz `enumerate` (choć ich rola jako iteratorów w Pythonie 3.0 zostanie w pełni wyjaśniona dopiero w kolejnym rozdziale).

W kolejnym rozdziale będziemy kontynuowali omawianie iteracji, przedstawiając listy składane oraz protokół iteracji Pythona — zagadnienia ściśle powiązane z pętlami `for`. Wyjaśnimy tam także pewne subtelności narzędzi iteracyjnych, które zostały wprowadzone w niniejszym rozdziale, takich jak `range` oraz `zip`. Jak zawsze jednak najpierw należy za pomocą quizu przećwiczyć kwestie przedstawione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Jakie są najważniejsze różnice funkcjonalne pomiędzy `while` i `for`?
2. Jaka jest różnica pomiędzy `break` i `continue`?
3. Kiedy w pętli wykonywana jest część `else`?
4. W jaki sposób można w Pythonie zapisać pętlę opartą na liczniku?
5. Do czego w pętli `for` można wykorzystać funkcję `range`?

Sprawdź swoją wiedzę — odpowiedzi

1. Pętla `while` jest ogólną instrukcją pętli, jednak to `for` zaprojektowane jest z myślą o iteracji wykonywanej na elementach sekwencji (czy, mówiąc bardziej ogólnie, obiektach, po których można iterować). Choć `while` może imitować `for` w przypadku pętli opartych na licznikach, wymaga większej ilości kodu i może działać wolniej.
2. Instrukcja `break` powoduje natychmiastowe wyjście z pętli (znajdziemy się poniżej całej instrukcji pętli `while` lub `for`), natomiast `continue` przeskakuje z powrotem na góre pętli (znajdziemy się tuż przed testem w `while` lub kolejnym elementem pobieranym w `for`).
3. Część `else` w pętlach `while` i `for` zostanie wykonana raz, kiedy pętla się kończy — o ile kończy się normalnie, bez trafienia na instrukcję `break`. Instrukcja `break` powoduje natychmiastowe wyjście z pętli i pominięcie części `else` (o ile jest ona w ogóle obecna).
4. Pętle liczników można zapisywać za pomocą instrukcji `while`, która ręcznie przechowuje indeks, lub za pomocą pętli `for`, która do generowania kolejnych wartości przesunięcia

- wykorzysta wbudowaną funkcję `range`. Żadna z nich nie jest preferowanym sposobem, jeśli po prostu potrzebujemy przejść wszystkie elementy sekwencji. W takiej sytuacji należy zamiast tego, kiedy tylko jest to możliwe, użyć prostej instrukcji `for`, bez `range` i liczników. Jest ona łatwiejsza do zapisania w kodzie, a także zazwyczaj szybsza do wykonania.
5. Funkcję wbudowaną `range` można wykorzystać w pętli `for` w celu zaimplementowania ustalonej liczby powtórzeń, przejścia po wartościach przesunięć zamiast po elementach znajdujących się na tych pozycjach, pominięcia kolejnych elementów w miarę przechodzenia, a także modyfikacji listy w trakcie przechodzenia jej. Żadna z tych ról nie wymaga `range` i w większości przypadków istnieją alternatywy — przejście samych elementów, wycinki z trzema wartościami granicznymi, a także listy składane są obecnie nieraz lepszymi rozwiązaniami (pomimo naturalnych upodobań byłych programistów języka C, którzy chcą wszystko zliczać).

Iteracje i składanie list — część 1.

W poprzednim rozdziale poznaliśmy dwie instrukcje Pythona odpowiedzialne za obsługę pętli: `while` i `for`. Pętle te są w stanie obsłużyć większość powtarzalnych zadań, ale do pracy wymagają danych sekwencyjnych, które są zjawiskiem tak powszechnym, że Python dorobił się narzędzi upraszczających i przyspieszających tego typu operacje. Niniejszy rozdział rozpoczyna naszą eksplorację tych narzędzi. A dokładniej: zajmiemy się koncepcją *protokołu iteracyjnego* zaimplementowanego w Pythonie (model oparty na wywołaniu metod obiektów iteracyjnych), jak również poznamy podstawy konstrukcji *list składanych* — mechanizmu blisko spokrewnionego z pętlą `for`, pozwalającego na wykonywanie wyrażeń na sekwencji elementów obiektu iterowanego.

Obydwa wspomniane narzędzia są ściśle związane zarówno z pętlami, jak i z funkcjami, więc zajmiemy się nimi w dwóch podejściach. Niniejszy rozdział omówi je w kontekście pętli i przetwarzania sekwencyjnego, niejako w ramach kontynuacji poprzedniego rozdziału. W dalszej części książki (rozdział 20.) iteracje i listy składane omówimy natomiast w kontekście narzędzi funkcyjnych. Dodatkowo w niniejszym rozdziale omówię zagadnienia związane z iteratorami, które pojawiły się (lub zmieniły) w Pythonie 3.0.

Warto uprzedzić z góry: część zagadnień poruszanych w tym rozdziale może na pierwszy rzut oka wydać się dość skomplikowana. Jednak w praktyce okazują się bardzo użyteczne i dają mnóstwo możliwości. Ich znajomość nie jest ściśle wymagana, ale ponieważ stały się one niezwykle powszechnie w programach w Pythonie, zrozumienie tej tematyki znacząco pomoże w analizie kodu napisanego przez innych programistów.

Pierwsze spojrzenie na iteratory

W poprzednim rozdziale wspomniałem, że pętla `for` może działać na dowolnym typie sekwencji Pythona, w tym na listach, krotkach i łańcuchach znaków — jak w poniższym przykładzie.

```
>>> for x in [1, 2, 3, 4]: (print x ** 2, end=' ')
...
1 4 9 16
>>> for x in (1, 2, 3, 4): (print x ** 3, end=' ')
...
1 8 27 64
>>> for x in 'mielonka': (print x * 2, end=' ')
...
mm ii ee ll oo nn kk aa
```

Tak naprawdę pętla `for` okazuje się o wiele bardziej uniwersalna, niż wynikałoby to z powyższych przykładów. Działa ona bowiem na dowolnym obiekcie, po którym można iterować. Tak samo jest w przypadku wszystkich innych narzędzi iteracyjnych w Pythonie, przechodzących obiekty od lewej do prawej strony — w tym pętli `for`, list składanych, testów przynależności `in`, a także wbudowanej funkcji `map`.

Koncepcja obiektów iterowanych (ang. *iterable object*) jest w Pythonie czymś relatywnie nowym, ale stała się cechą dominującą tego języka. Jest to w zasadzie uogólnienie pojęcia sekwencji. Obiekt uznawany jest za taki, jeśli jest albo fizycznie przechowywaną sekwencją, albo obiektem zwracającym jeden wynik naraz w kontekście narzędzia iteracyjnego, takiego jak pętla `for`. W pewnym sensie obiekty tego typu obejmują zarówno fizyczne sekwencje, jak i *sekwencje wirtualne* tworzone na żądanie.¹

Protokół iteracyjny, iteratory plików

Jednym z łatwiejszych sposobów zrozumienia, co to może oznaczać, jest przyjrzenie się, jak to działa w przypadku typu wbudowanego, takiego jak plik. Jak pamiętamy z rozdziału 9., otwarte obiekty plików obsługują metodę o nazwie `readline()`, wczytującą po jednym wierszu tekstu z pliku naraz. Przy każdym wywołaniu metody `readline()` przesuwamy się do kolejnego wiersza. Na końcu pliku zwracany jest pusty łańcuch znaków, który możemy wykryć w celu wyjścia z pętli.

```
>>> f = open('script1.py')      # Odczytanie 4-wierszowego skryptu z bieżącego katalogu
>>> f.readline()              # funkcja readline wczytuje po jednym wierszu
'import sys\n'
>>> f.readline()
'print sys.path\n'
>>> f.readline()
'x = 2\n'
>>> f.readline()
'print 2 ** 33\n'
>>> f.readline()              # Zwraca pusty ciąg znaków na końcu pliku
''
```

Pliki obsługują również metodę o nazwie `__next__()`, która ma dokładnie ten sam efekt — przy każdym wywołaniu zwraca kolejny wiersz z pliku. Jedyną zauważalną różnicą jest to, że `__next__()` na końcu pliku w miejscu pustego łańcucha znaków zwraca wyjątek `StopIteration`.

```
>>> f = open('script1.py')      # __next__ również wczytuje po jednym wierszu
>>> f.__next__()              # ale wywołuje wyjątek na końcu pliku
'import sys\n'
>>> f.__next__()
'print sys.path\n'
>>> f.__next__()
'x = 2\n'
>>> f.__next__()
'print 2 ** 33\n'
>>> f.__next__()
```

¹ Terminologia dotycząca tej tematyki jest zdefiniowana dość luźno. W tekście posługuję się terminami „obiekt iterowany” oraz „iterator” w odniesieniu do ogólnie rozumianych obiektów obsługujących mechanizm iteracji. Najczęściej określenie „obiekt iterowany” odnosi się do obiektu klasy obsługiwanej przez funkcję `iter()`, natomiast „iterator” odnosi się do obiektu zwracanego przez funkcję `iter()`, który to obiekt obsługuje metodę `next(I)`. Należy jednak pamiętać, że ta konwencja nie jest stosowana w sposób spójny, zarówno w innych publikacjach dotyczących Pythona, jak i w tej książce.

```
Traceback (most recent call last):
...pominęta treść wyjątku...
StopIteration
```

Ten interfejs jest dokładnie tym, co w Pythonie nazywamy *protokiem iteracji* — obiektem obsługującym metodę `__next__()` przechodzącą do kolejnego wyniku, który zgłasza wyjątek `StopIteration` na końcu serii wyników. W Pythonie każdy obiekt tego rodzaju nazywamy obiektem iterowanym (ang. *iterable*). Takie obiekty można przechodzić za pomocą pętli `for` czy innego narzędzia iteracyjnego, ponieważ wszystkie narzędzia iteracyjne wewnętrznie działają dzięki wywoływaniu metody `__next__()` w każdej iteracji i przechwytywaniu wyjątku `StopIteration` w celu ustalenia, kiedy skończyć.

Rezultat tej magii jest taki, że — jak wspomnieliśmy w rozdziale 9. — najlepszym sposobem na wczytanie wiersza pliku tekstowego w tej chwili *wcale nie jest wczytanie go*. Zamiast tego lepiej jest pozwolić pętli `for` na automatyczne wywołanie metody `__next__()` w celu przejścia do kolejnego wiersza z każdą iteracją. Poniższy kod wczytuje plik wiersz po wierszu (wyświetlając przy okazji wersję zapisaną wielkimi literami) bez jawnego odczytania czegokolwiek z pliku.

```
>>> for line in open('script1.py'):
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)
```

Użycie iteratorów plików
Wywołuje __next__ przechwytyuje StopIteration

Warto zwrócić uwagę na parametr `end=' '` funkcji `print()`, który spowoduje, że funkcja ta nie dopisze znaku nowego wiersza `\n` (jak to czyni standardowo) — w przeciwnym razie w wyniku między wierszami otrzymalibyśmy dodatkowy pusty wiersz. Metoda ta uznawana jest za najlepszy sposób wczytywania plików tekstowych z trzech powodów — jest najprostsza do zapisania w kodzie, najszybsza do wykonania i najlepsza, jeśli chodzi o użycie pamięci. Starszy, oryginalny sposób uzyskania tego samego efektu za pomocą pętli `for` polegał na wywołaniu metody pliku `readlines()` w celu załadowania zawartości pliku do pamięci w postaci listy łańcuchów znaków.

```
>>> for line in open('script1.py').readlines():
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 33)
```

Technika wykorzystująca `readlines()` nadal działa, jednak nie jest uznawana za najlepszą praktykę i spisuje się słabo, jeśli chodzi o wykorzystanie pamięci. Tak naprawdę, ponieważ ta wersja rzeczywiście ładuje cały plik naraz do pamięci, dla plików większych od rozmiaru pamięci dostępnej na komputerze nie będzie wcale działała. Z kolei wersja oparta na iteratorze nie jest podatna na problemy związane z pamięcią, ponieważ wczytuje jeden wiersz naraz.

Co więcej, wersja z iteratorem została w Pythonie mocno zoptymalizowana, dlatego powinna również działać szybciej, choć właściwość ta jest w dużym stopniu zależna od wersji (Python 3.0 nieco zmienił się w tym zakresie na niekorzyść, a to z powodu ponownej implementacji bibliotek wejścia-wyjścia w taki sposób, aby ujednolicić obsługę standardu Unicode i zmniejszyć zależność kodu od systemu operacyjnego).

Jak wspomniano w rozdziale 13. w ramce „Znaczenie skanerów plików”, wiersz pliku można również wczytać za pomocą pętli `while`.

```
>>> f = open('script1.py')
>>> while True:
...     line = f.readline()
...     if not line: break
...     print line.upper(),
...
...te same dane wyjściowe...
```

Kod ten będzie jednak najprawdopodobniej działał wolniej od pętli `for` opartej na iteratorze, ponieważ iteratory działają wewnątrz Pythona z szybkością języka C, natomiast wersja z pętlą `while` wykonuje kod bajtowy Pythona za pośrednictwem maszyny wirtualnej Pythona. Za każdym razem gdy wymieniamy kod Pythona na kod języka C, szybkość wzrasta. Ponownie jednak: własność ta nie jest uniwersalna, szczególnie w Pythonie 3.0. Zagadnieniami wydajności zajmiemy się w dalszej części książki, gdy będziemy omawiać porównanie efektywności tego typu konstrukcji.

Kontrola iteracji — `iter` i `next`

W Pythonie 3.0 wprowadzono mechanizmy kontroli iteracji, dzięki którym programiści nie muszą wprowadzać dużej ilości kodu. Służy do tego funkcja wbudowana `next()`, która automatycznie wywołuje metodę `__next__()` obiektu. Jeśli `X` jest obiektem iterowanym, wywołanie `next(X)` da ten sam efekt co wywołanie `X.__next__()`, jednak kod jest o wiele bardziej estetyczny. W przypadku plików możemy stosować dowolną z form:

```
>>> f = open('script1.py')
>>> f.__next__()                                # Bezpośrednie wywołanie metody iteratora
'import sys\n'
>>> f.__next__()
'print(sys.path)\n'

>>> f = open('script1.py')
>>> next(f)                                     # Niejawne wywołanie metody __next__
'import sys\n'
>>> next(f)
'print(sys.path)\n'
```

Istnieje jeszcze jeden element protokołu iteratorów. Przy wywołaniu pętli `for` z obiektu iterowanego generowany jest iterator — służy do tego funkcja wbudowana `iter()`. Dopiero obiekt zwrócony z funkcji `iter()` obsługuje metodę `next()` wykorzystywaną przez protokół iteratorów. Aby zrozumieć szczegóły, warto przyjrzeć się sposobowi obsługi w pętli `for` wbudowanych typów sekwencyjnych, jak na przykład listy:

```
>>> L = [1, 2, 3]
>>> I = iter(L)                                 # Pozyskanie obiektu iteratora
>>> I.next()                                    # Odczyt następnego elementu iteratora
1
>>> I.next()
2
>>> I.next()
3
>>> I.next()
Traceback (most recent call last):
...pominęta część wyniku...
StopIteration
```

Tego typu inicjalizacja nie jest konieczna w przypadku plików, ponieważ obiekt plikowy jest sam w sobie iteratorem. Innymi słowy, obiekty plikowe posiadają metodę `__next__()`, zatem do obsługi plików nie ma potrzeby tworzenia osobnego obiektu:

```
>>> f = open('script1.py')
>>> iter(f) is f
True
>>> f.__next__()
'import sys\n'
```

Listy oraz inne wbudowane typy sekwencyjne nie są własnymi iteratorami, ponieważ w przypadku tych sekwencji istnieje możliwość wielokrotnej równoległej iteracji. W przypadku tych typów należy utworzyć iterator za pomocą funkcji `iter()`.

```
>>> L = [1, 2, 3]
>>> iter(L) is L
False
>>> L.__next__()
AttributeError: 'list' object has no attribute '__next__'

>>> I = iter(L)
>>> I.__next__()
1
>>> next(I)                                # Równoważne I.__next__()
2
```

Narzędzia iteracyjne Pythona wywołują te funkcje w sposób automatyczny, jednak możemy ich użyć w sposób *jawny*. Poniższy przykład demonstruje równoważność jawnego i niejawnego użycia mechanizmów iteracji:²

```
>>> L = [1, 2, 3]
>>>
>>> for X in L:                            # Iteracja automatyczna
...     print(X ** 2, end=' ')
# Wywołuje funkcję iter(), metodę __next__(), przechwytyując wyjątki
...
1 4 9

>>> I = iter(L)                           # Ręczna iteracja: pętla for robi to samo niejawnie
>>> while True:
...     try:
...         X = next(I)                    # Instrukcja try pozwala obsługiwać wyjątki
...     except StopIteration:            # Alternatywnie: I.__next__()
...         break
...     print(X ** 2, end=' ')
...
1 4 9
```

² W rzeczywistości pętla `for` wewnętrznie wykorzystuje metodę `I.__next__()`, a nie zademonstrowaną tu funkcję `next(I)`. Z reguły jednak nie ma żadnych różnic, ale należy zdawać sobie sprawę, że w Pythonie 3.0 istnieją obiekty wbudowane (jednym z nich jest wynik wywołania funkcji `os.popen()`), które obsługują pierwszą formę iteratora, a nie obsługują drugiej, choć nie ma problemu z użyciem ich w pętli `for`. Można jednak bezpiecznie założyć, że formy te są równoważne. Dla pełnego obrazu należy też wspomnieć, że w Pythonie 3.0 obsługa funkcji `os.popen()` została przepisana z użyciem modułu `subprocess`. W ten sposób została zaimplementowana klasa dopakowująca obsługującą metodę `next()`, przez co w sposób niejawnym nie jest już wywoływana metoda `__next__()` (ale bezpośrednio można ją nadal wywołać). Jest to specyficzna cecha Pythona 3.0, która, jak się okazuje, pozostawiła swoje piętno również na innych elementach biblioteki standarowej! Tematowi temu przyjrzymy się szerzej w rozdziałach 37. i 38. Warto również wspomnieć, że w Pythonie 3.0 nie są dostępne funkcje `popen2()`, `popen3()`, `popen4()` modułu `os` — w ich miejscu należy stosować funkcję `subprocess.Popen()` z odpowiednim argumentem (przykłady wywołań można znaleźć w podręczniku biblioteki standarowej Pythona 3.0).

Aby zrozumieć ten kod, należy mieć świadomość, że instrukcje `try` służą do wykonywania kodu i przechwytywania wyjątków wywoływanych w tym kodzie (tematykę tę omówimy szczegółowo w części VII). Warto jeszcze wspomnieć, że pętle `for` i inne formy iteracji w przypadku klas definiowanych przez użytkownika mogą działać inaczej od opisanych tu zasad, na przykład odwołując się do kolejnych indeksów obiektu na liście zamiast stosowania opisanego tu protokołu iteratorów. Do tego zagadnienia wróćmy przy okazji tematu przeciążania operatorów klas (rozdział 29.).



Uwaga na temat wersji: W Pythonie 2.6 metoda iterująca nosi nazwę `X.next()`, a nie `X.__Next__()`. W celu uproszczenia przenoszenia kodu do Pythona 3.0 w Pythonie 2.6 dostępna jest również funkcja wbudowana `next(X)` (ale nie jest ona dostępna we wcześniejszych wersjach Pythona). Funkcja ta w Pythonie 2.6 wywołuje metodę `X.next()`, natomiast w Pythonie metodę `X.__next__()`. Pozostałe mechanizmy iteracyjne działają jednak tak samo. Podsumowując: w Pythonie 3.0 możemy stosować metodę `X.__next__()` lub funkcję `next(X)`, w Pythonie 2.6 stosujemy metodę `X.next()` lub funkcję `next(X)`, natomiast we wcześniejszych wersjach Pythona używamy wyłącznie metody `X.next()`.

Inne iteratory typów wbudowanych

Poza fizycznymi sekwencjami, takimi jak listy, pozostałe typy również mają przydatne iteratory. Klasycznym sposobem przechodzenia kluczy słownika jest na przykład jawne zażądanie listy kluczy za pomocą metody `keys()`.

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> for key in D.keys():
...     print(key, D[key])
...
a 1
c 3
b 2
```

W nowszych wersjach Pythona nie potrzebujemy już wywoływać metody `keys()` — słowniki obsługują iterator automatycznie zwracający w kontekście iteracyjnym po jednym klucz na raz:

```
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'c'
>>> next(I)
'b'
>>> next(I)
Traceback (most recent call last):
...pominięta część wyniku...
StopIteration
```

W efekcie nie ma już potrzeby używania metody `keys()` do przeglądania kluczy słownika: pętla `for` wykorzysta protokół iteratorów do uzyskania kluczy po jednym w każdym swoim przebiegu.

```
>>> for key in D:
...     print(key, D[key])
...
a 1
c 3
b 2
```

Bez wnikania w szczegóły warto wspomnieć, że istnieją również inne obiekty Pythona obsługujące protokół iteracyjny, które dzięki temu można wykorzystywać w pętli `for`. Na przykład obiekty typu `shelve` (system plików do zapisu i odczytu obiektów Pythona) oraz wyniki wywołania funkcji `os.popen()` (narzędzie do wywoływania poleceń powłoki z możliwością odczytu zwracanych przez nie wyników) również są obiektami iterowanymi:

```
>>> import os
>>> P = os.popen('dir')
>>> P.__next__()
' Volume in drive C is S0004828V03\n'
>>> P.__next__()
' Volume Serial Number is 08BE-3CD4\n'
>>> next(P)
TypeError: _wrap_close object is not an iterator
```

Obiekty zwracane z funkcji `popen()` w Pythonie 2.6 obsługują metodę `P.next()`. W Pythonie 3.0 obsługują metodę `P.__next__()`, ale nie obsługują funkcji wbudowanej `next(P)`. Jednak ta funkcja (według dokumentacji) powinna wywoływać metodę `P.__next__()`, więc nie wiadomo, czy ta własność zostanie zachowana w kolejnych wersjach języka (jak wspomniano we wcześniejszym przypisie, wydaje się, że to po prostu błąd w implementacji Pythona 3.0). Problem istnieje wyłącznie w przypadku ręcznej obsługi iteracji; jeśli wynik funkcji `popen()` poddamy automatycznej iteracji w pętli `for` lub innym kontekście iteracyjnym (opisanym w kolejnych punktach rozdziału), kolejne wiersze wyniku zostaną prawidłowo przetworzone niezależnie od użytej wersji Pythona.

Protokół iteracyjny jest również powodem opakowania niektórych wyników w funkcję `list()` — chodziło o wypisanie wszystkich elementów iteratatora. Obiekty iterowane są bowiem obsługiwane po jednym elemencie i nie działają jak lista.

```
>>> R = range(5)
>>> R
range(0, 5)                                     # W 3.0 zakresy są obiektami iterowanymi
>>> I = iter(R)                                 # Użycie protokołu iteracyjnego
>>> next(I)
0
>>> next(I)
1
>>> list(range(5))                            # Użycie listy do jednoczesnej prezentacji wszystkich wyników
[0, 1, 2, 3, 4]
```

Teraz, gdy wiemy nieco więcej na temat protokołu iteracyjnego, możemy przeanalizować zachowanie funkcji `enumerate()`, którą poznaliśmy w poprzednim rozdziale.

```
>>> E = enumerate('spam')                      # Wynik funkcji enumerate() również jest obiektem iterowanym
>>> E
<enumerate object at 0x0253F508>
>>> I = iter(E)                                # Wygenerowanie wyników z użyciem protokołu iteracyjnego
(0, 's')
>>> next(I)                                    # Przekształcenie na listę generuje wszystkie elementy jednocześnie
(1, 'p')
>>> list(enumerate('spam'))
[(0, 's'), (1, 'p'), (2, 'a'), (3, 'm')]
```

Z reguły nie mamy okazji obserwować tych szczegółów w działaniu, ale są one wykorzystywane niejawnie przez mechanizm pętli `for`. W rzeczywistości wszelkie przypadki kolejnego przeglądania elementów w Pythonie wykorzystują protokół iteracyjny. Kolejne przykłady poznamy w następnym punkcie.

Listy składane — wprowadzenie

Poznaliśmy mechanizm działania protokołu iteracyjnego, nadszedł więc czas na poznanie najbardziej powszechnych jego zastosowań. Obok pętli `for` najczęstszą formą użycia protokołu iteracyjnego są listy składane (ang. *list comprehension*).

W poprzednim podrozdziale nauczyliśmy się, jak można za pomocą funkcji `range()` zmodyfikować listę w miarę jej przechodzenia.

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)):
...     L[i] += 10
...
>>> L
[11, 12, 13, 14, 15]
```

Takie rozwiązanie działa, jednak, jak wspomnieliśmy wcześniej, może nie być to najbardziej optymalne i najlepsze podejście w Pythonie. Wyrażenie list składanych sprawia, że tego typu przypadki użycia stały się mało efektywne. W poniższym przykładzie pętlę zastąpiliśmy pojedynczym wyrażeniem generującym pożądaną listę wyników.

```
>>> L = [x + 10 for x in L]
>>> L
[21, 22, 23, 24, 25]
```

Rezultat jest taki sam, jednak wymaga mniejszej ilości kodu z naszej strony i prawdopodobnie działa znacznie szybciej. Lista składana nie jest dokładnie tym samym co wersja z instrukcją `for`, ponieważ tworzy ona *nowy* obiekt listy (co może mieć znaczenie, jeśli istnieje wiele referencji do oryginalnej listy), dla większości zastosowań jest jednak wystarczająco bliska. Jest to również często stosowane i na tyle wygodne rozwiązań, że zasługuje na osobne omówienie.

Podstawy list składanych

Z listami składanymi po raz pierwszy spotkaliśmy się w rozdziale 4. Z punktu widzenia składni listy składane pochodzą od konstrukcji w zapisie teorii zbiorów, stosującej operacje do każdego elementu zbioru, jednak do ich używania nie jest potrzebna znajomość całej teorii. W Pythonie dla większości osób listy składane wyglądają po prostu jak pisane od tyłu pętle `for`.

Przyjrzyjmy się dokładniej zaprezentowanemu wyżej przykładowi:

```
>>> L = [x + 10 for x in L]
```

Listy składane zapisywane są w nawiasach kwadratowych, ponieważ są one metodą tworzenia nowej listy. Zaczynają się od dowolnego, zbudowanego przez nas wyrażenia, które wykorzystuje tworzoną przez nas zmienną pętli (`x + 10`). Potem następuje część, którą powinniśmy już rozpoznać jako nagłówek pętli `for`. W części tej wymieniona zostaje zmienna pętli oraz obiekt, po którym będziemy iterować (`for x in L`).

By wykonać wyrażenie, Python wykonuje iterację po `L` wewnętrz interpretera, przypisując `x` do każdego elementu z kolei, i generuje wyniki wykonania lewej strony wyrażenia na przechodzonych elementach. Otrzymany wynik jest nową listą zawierającą `x + 10` dla każdego `x` z listy `L`.

Z technicznego punktu widzenia listy składane są konstrukcją całkowicie opcjonalną, ponieważ zawsze możemy zbudować listę wyników wyrażenia ręcznie za pomocą pętli `for` dodającej po kolej wyniki do listy w miarę przechodzenia elementów.

```
>>> res = []
>>> for x in L:
...     res.append(x + 10)
...
>>> res
[21, 22, 23, 24, 25]
```

Tak naprawdę lista składana robi dokładnie to samo w sposób niejawny.

Listy składane mają jednak bardziej zwięzły zapis, a ponieważ ten wzorzec kodu budowania list wyników jest w Pythonie tak często spotykany, w wielu kontekstach okazują się one po prostu bardzo wygodne. Co więcej, listy składane mogą działać o wiele szybciej od ręcznie zapisanych instrukcji pętli `for` (tak naprawdę mniej więcej dwa razy szybciej), ponieważ ich iteracje wykonywane są wewnątrz interpretera z szybkością języka C, a nie z szybkością kodu Pythona. Z tego powodu, szczególnie w przypadku większych zbiorów danych, używanie ich może być bardzo korzystne z punktu widzenia wydajności.

Wykorzystywanie list składanych w plikach

Przyjrzyjmy się teraz kolejnemu często spotykanemu zastosowaniu list składanych, co pozwoli nam lepiej zrozumieć tę technikę. Jak pamiętamy, obiekt pliku ma metodę `readlines()`, która za jednym razem ładuje plik do listy łańcuchów znaków.

```
>>> f = open('script1.py')
>>> lines = f.readlines()
>>> lines
['import sys\n', 'print sys.path\n', 'x = 2\n', 'print 2 ** 33\n']
```

Takie rozwiązanie działa, jednak wszystkie wiersze wyniku mają na końcu znak nowego wiersza (`\n`). W wielu programach znak ten przeszkadza — musimy na przykład uważać, by przy wyświetlaniu uniknąć podwójnych odstępów. Byłoby miło, gdyby udało nam się w jednym ruchu pozbyć wszystkich znaków nowych linii, prawda?

Z każdym razem, gdy rozważamy wykonanie operacji na każdym elemencie sekwencji, znajdziemy się w królestwie list składanych. Zakładając na przykład, że zmienna `lines` z poprzedniego przykładu pozostaje bez zmian, poniższy kod pozwala nam wykonać to zadanie, wywołując na każdym wierszu z listy metodę łańcuchów znaków `rstrip()`, która usuwa białe znaki po prawej stronie. Podobnie zadziałałby również wycinek `line[:-1]`, jednak tylko wtedy, gdybyśmy byli pewni, że wszystkie wiersze zostały poprawnie zakończone.

```
>>> lines = [line.rstrip() for line in lines]
>>> lines
['import sys', 'print sys.path', 'x = 2', 'print 2 ** 33']
```

Takie rozwiązanie działa, jednak ponieważ listy składane są kolejnym kontekstem iteracyjnym, podobnie jak proste pętle `for`, nie musimy nawet wstępnie otwierać pliku. Jeśli otworzymy go w wyrażeniu, lista składana automatycznie wykorzysta omówiony wcześniej protokół iteracji. Wczyta zatem po jednym wierszu z pliku na raz, wywołując metodę `next()` obiektu pliku, wykona na tym wierszu metodę `rstrip()` oraz doda do listy wyników. I znów otrzymujemy to, co chcieliśmy — wynik zastosowania metody `rstrip()` na każdym wierszu pliku.

```
>>> lines = [line.rstrip() for line in open('script1.py')]
>>> lines
['import sys', 'print sys.path', 'x = 2', 'print 2 ** 33']
```

Powyższe wyrażenie wiele operacji wykonuje w sposób niejawny, jednak dzięki temu sporo otrzymujemy za darmo — Python przegląda plik i buduje listę wyników operacji w sposób automatyczny. Jest to również wydajny sposób zapisania tej operacji w kodzie — ponieważ większość pracy wykonywana jest wewnątrz interpretera Pythona, rozwiązywanie to będzie najprawdopodobniej dużo szybsze od odpowiadającej mu pętli `for`. I znów, podobnie jak w przypadku większych plików, zysk w zakresie szybkości może być znaczący.

Oprócz wydajności listy składane mają jeszcze jedną zaletę: wykonują sporo pracy w niewielkiej ilości kodu. W naszym przykładzie na każdym wierszu przetwarzanego pliku możemy wykonać dowolną znakową. Poniższy listing to odpowiednik zastosowanego wyżej przykładu na przekształcenie wszystkich liter pliku na wielkie, tym razem jednak w ujęciu rozwinięcia list. Następne przykłady to wariacje na ten sam temat (warto zwrócić uwagę na połączone w łańcuch wywołania metod z przykładu drugiego, które są możliwe dzięki temu, że metoda ciągu znaków zwraca nowy ciąg, na którym jest wywoływana kolejna metoda łańcucha).

```
>>> [line.upper() for line in open('script1.py')]
['IMPORT SYS\n', 'PRINT(SYS.PATH)\n', 'X = 2\n', 'PRINT(2 ** 33)\n']
>>> [line.rstrip().upper() for line in open('script1.py')]
['IMPORT SYS', 'PRINT(SYS.PATH)', 'X = 2', 'PRINT(2 ** 33)']
>>> [line.split() for line in open('script1.py')]
[['import', 'sys'], ['print(sys.path)'], ['x', '=', '2'], ['print(2', '**', '33')']]
>>> [line.replace(' ', '!') for line in open('script1.py')]
['import!sys\n', 'print(sys.path)\n', 'x!=2\n', 'print(2!**!33)\n']
>>> [('sys' in line, line[0]) for line in open('script1.py')]
[(True, 'i'), (True, 'p'), (False, 'x'), (False, 'p')]
```

Rozszerzona składnia list składanych

Tak naprawdę listy składane mogą w praktyce być jeszcze bardziej zaawansowane. Przydatnym rozszerzeniem jest dołączenie do zagnieżdżonej pętli `for` instrukcji `if`, która odfiltruje elementy niespełniające podanego warunku.

Załóżmy na przykład, że chcemy powtórzyć poprzedni przykład, jednak musimy pobrać jedynie wiersze rozpoczynające się od litery `p` (być może pierwszy znak wiersza jest jakiegoś rodzaju kodem działania). Jest to możliwe dzięki dodaniu do wyrażenia części `if` z filtrem.

```
>>> lines = [line.rstrip() for line in open('script1.py') if line[0] == 'p']
>>> lines
['print sys.path', 'print 2 ** 33']
```

W powyższym kodzie instrukcja `if` sprawdza każdy wiersz wczytany z pliku, weryfikując, czy jego pierwszym znakiem jest `p`. Jeśli tak nie jest, wiersz ten jest pomijany w liście wyników. Wyrażenie to jest dość rozbudowane, ale łatwe do zrozumienia, jeśli przełożymy je na prostą instrukcję pętli `for` (listę składaną można zawsze zamienić na instrukcję `for`, dodając do wyniku elementy iteratora).

```
>>> res = []
>>> for line in open('script1.py'):
...     if line[0] == 'p':
...         res.append(line.rstrip())
...
>>> res
['print sys.path', 'print 2 ** 33']
```

Takie rozwiązanie z pętlą `for` działa, jednak zamiast jednego wiersza kodu wymaga czterech i prawdopodobnie działa zauważalnie wolniej.

Listy składane mogą się stać nawet bardziej skomplikowane, jeśli będzie taka konieczność. Mogą na przykład zawierać zagnieżdżone pętle zapisane w kodzie jako seria części `for`. Ich pełna składnia pozwala na umieszczenie dowolnej liczby `for`, a każda z nich może mieć opcjonalną, powiązaną część `if` (więcej informacji na temat ich składni znajdziesz się w rozdziale 20.).

Poniższy kod tworzy na przykład listę konkatenacji `x + y` dla każdego `x` z jednego łańcucha znaków i każdego `y` z drugiego. W rezultacie zbiera permutację znaków z dwóch łańcuchów.

```
>>> [x + y for x in 'abc' for y in 'lmn']
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Wyrażenie to można łatwiej zrozumieć, przekształcając je na pętlę `for`. Poniżej znajduje się jego — najprawdopodobniej wolniejszy — odpowiednik, który daje ten sam rezultat.

```
>>> res = []
>>> for x in 'abc':
...     for y in 'lmn':
...         res.append(x + y)
...
>>> res
['al', 'am', 'an', 'bl', 'bm', 'bn', 'cl', 'cm', 'cn']
```

Pomijając tego typu poziom komplikacji, listy składane często jednak bywają nawet zbyt zwięzłe. Są one przeznaczone dla prostych typów iteracji. W przypadku bardziej skomplikowanych zadań struktura pętli `for` będzie zwykle łatwiejsza do zrozumienia i zmodyfikowania w przyszłości. Jak zawsze w świecie programowania, jeśli coś staje się dla nas zbyt trudne do zrozumienia, najprawdopodobniej użycie tego nie jest najlepszym pomysłem.

Dla iteratorów i list składanych powrócimy w rozdziale 20. w kontekście narzędzi programowania funkcyjnego. Jak zobaczymy, okażą się one powiązane z funkcjami w równym stopniu jak z instrukcjami pętli.

Inne konteksty iteracyjne

W dalszej części książki przekonamy się, że klasy definiowane przez użytkownika również mogą implementować protokół iteracyjny. Tym bardziej warto znać narzędzia potrafiące korzystać z tego mechanizmu, ponieważ każde narzędzie, które potrafi obsłużyć wbudowane typy iterowane, będzie potrafiło pracować z dowolnymi iteratorami definiowanymi przez użytkownika.

Dotychczas prezentowałem iteratory w kontekście instrukcji pętli `for`, będącej jednym z podstawowych tematów niniejszego rozdziału. Należy jednak pamiętać, że każdy protokół przechodzący obiekt od lewej do prawej strony korzysta z protokołu iteracji. Obejmuje to widziane już pętle `for`.

```
>>> for line in open('script1.py'):
...     print(line.upper(), end='')
...
IMPORT SYS
PRINT SYS.PATH
X = 2
PRINT(2 ** 33)                                # Użycie iteratorów plików
```

Z protokołu iteracji korzystają jednak również listy składane, testy przynależności `in`, wbudowana funkcja `map` i inne elementy wbudowane, takie jak wywołania `sorted` czy `zip`. Po zastosowaniu na otwartym pliku wymienione funkcje i instrukcje automatycznie wykorzystują protokół iteracyjny do przetwarzania pliku wiersz po wierszu:

```
>>> uppers = [line.upper() for line in open('script1.py')]
>>> uppers
['IMPORT SYS\n', 'PRINT SYS.PATH\n', 'X = 2\n', 'PRINT 2 ** 33\n']
>>> map(str.upper, open('script1.py'))           # map() jest w 3.0 obiektem iterowanym
<map object at 0x02660710>

>>> list(map(str.upper, open('script1.py')))
['IMPORT SYS\n', 'PRINT SYS.PATH\n', 'X = 2\n', 'PRINT 2 ** 33\n']

>>> 'y = 2\n' in open('script1.py')
False
>>> 'x = 2\n' in open('script1.py')
True
```

Z wywołaniem `map` spotkaliśmy się w poprzednim rozdziale. Ta funkcja wbudowana wykonująca wywołanie funkcji na każdym elemencie obiektu iterowanego przekazanego w argumencie. Funkcja `map` jest w swoim działaniu podobna do listy składanej, jednak ma bardziej ograniczony zakres zastosowań, ponieważ wymaga użycia funkcji, a nie dowolnego wyrażenia. W Pythonie 3.0 funkcja `map` zwraca obiekt iterowany, zatem w celu wyświetlenia wszystkich wyników na raz musimy przekształcić ten obiekt na listę. Więcej informacji na temat tej zmiany w Pythonie 3.0 można znaleźć w dalszej części książki. Ponieważ listy składane są powiązane z pętlami `for`, powrócimy do nich w rozdziałach 19. i 20.

W Pythonie dostępnych jest wiele narzędzi operujących na obiektach iterowanych: `sorted` sortuje elementy, `zip` łączy elementy kilku obiektów iterowanych, `enumerate` wylicza elementy wraz z ich indeksem, `filter` zwraca elementy, dla których spełniony jest podany warunek, `reduce` wykonuje wskazaną funkcję na elementach. Wszystkie te funkcje pracują na obiektach iterowanych, a `zip`, `enumerate` i `filter` w Pythonie 3.0 dodatkowo zwracają obiekt iterowany (podobnie jak `map`). Oto przykłady wywołania wymienionych funkcji na otwartym pliku.

```
>>> sorted(open('script1.py'))
['import sys\n', 'print(2 ** 33)\n', 'print(sys.path)\n', 'x = 2\n']
>>> list(zip(open('script1.py'), open('script1.py')))
[('import sys\n', 'import sys\n'), ('print(sys.path)\n', 'print(sys.path)\n'),
 ('x = 2\n', 'x = 2\n'), ('print(2 ** 33)\n', 'print(2 ** 33)\n')]
>>> list(enumerate(open('script1.py')))
[(0, 'import sys\n'), (1, 'print(sys.path)\n'), (2, 'x = 2\n'),
 (3, 'print(2 ** 33)\n')]
>>> list(filter(bool, open('script1.py')))
['import sys\n', 'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n']
>>> import functools, operator
>>> functools.reduce(operator.add, open('script1.py'))
'import sys\nprint(sys.path)\nx = 2\nprint(2 ** 33)\n'
```

Wszystkie te funkcje są narzędziami iteracyjnymi, ale każda ma własne, unikalne zadanie. Funkcje `zip()` i `enumerate()` mieliśmy okazję poznać w poprzednim rozdziale, natomiast `filter()` i `reduce()` wykorzystują koncepcje funkcyjne, które będą omawiane w rozdziale 19.

Funkcję `sorted()`, użytą w powyższym przykładzie, poznaliśmy w rozdziale 4., a w rozdziale 8. wykorzystaliśmy ją w kontekście słowników. Funkcja wbudowana `sorted()` jest odpowiednikiem metody `sort()` obsługiwanej przez sekwencje, ale zwraca nową listę zawierającą

posortowane elementy i potrafi przetwarzać dowolny obiekt iterowany. Należy pamiętać, że w Pythonie 3.0 funkcja `sorted()` zwraca listę, czym różni się od funkcji `map()` i pozostałych funkcji o podobnym działaniu.

Istnieją również inne funkcje wbudowane obsługujące obiekty iterowane, ale trudno jest zadeemonstrować ich działanie na przykładzie pliku. Na przykład funkcja `sum()` oblicza sumę elementów obiektu iterowanego. Funkcja `any()` zwraca wartość `True`, jeśli dowolny element ma wartość `True`, natomiast funkcja `all()` zwraca `True`, jeśli wszystkie elementy mają wartość `True`. Funkcje z poniższego przykładu działają na podobnej zasadzie co `reduce()`: wykonują operację na wszystkich elementach obiektu iterowanego, ale zwracają pojedynczy wynik.

```
>>> sum([3, 2, 4, 1, 5, 0])                                # sum oczekuje samych liczb
15
>>> any(['spam', '', 'ni'])
True
>>> all(['spam', '', 'ni'])
False
>>> max([3, 2, 5, 1, 4])
5
>>> min([3, 2, 5, 1, 4])
1
```

W rzeczywistości funkcji `max()` i `min()` można użyć w kontekście pliku — zwrócią, odpowiednio, ciągi znaków o największej i najmniejszej wartości (po posortowaniu użyteczność tego typu operacji pozostawiam wyobraźni Czytelników).

```
>>> max(open('script1.py'))                                # Wiersz z ciągiem znaków o najwyższej wartości
'x = 2\n'
>>> min(open('script1.py'))
'import sys\n'
```

Co ciekawe, protokół iteracji jest w Pythonie jeszcze bardziej wszechobecny, niż wynikałoby to z dotychczasowych prezentacji. Każdy element z wbudowanego zestawu narzędzi Pythona przechodzący obiekty od lewej do prawej strony zdefiniowany został tak, by używać protokołu iteracji na tych obiektach. Dotyczy to nawet tych bardziej ezoterycznych elementów, jak wbudowane funkcje `list()` oraz `tuple()` (tworzące z obiektu iterowanego odpowiednio nową listę lub krotkę), czy metoda łańcuchów znaków `join()` (umieszczająca podłączuch między łańcuchami znaków obiektu iterowanego), a nawet przypisania sekwencji. Z tego powodu wszystkie z poniższych metod będą działały na otwartym pliku i automatycznie wczytają po jednym wierszu na raz.

```
>>> list(open('script1.py'))
['import sys\n', 'print sys.path\n', 'x = 2\n', 'print 2 ** 33\n']

>>> tuple(open('script1.py'))
('import sys\n', 'print sys.path\n', 'x = 2\n', 'print 2 ** 33\n')

>>> '&&'.join(open('script1.py'))
'import sys\n&&print sys.path\n&&x = 2\n&&print 2 ** 33\n'

>>> a, b, c, d = open('script1.py')
>>> a, d
('import sys\n', 'print 2 ** 33\n')
>>> a, *b = open('script1.py')                               # Rozszerzona składnia Pythona 3.0
>>> a, b
('import sys\n', ['print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n'])
```

We wcześniejszym przykładzie widzieliśmy, że wbudowana funkcja `zip()` również akceptuje wynik funkcji `zip()` (będący obiektem iterowanym). Podobnie działa funkcja `set()` oraz *wyrażenia rozwijanych zbiorów i słowników*, dostępne od Pythona 3.0, które mieliśmy okazję poznać w rozdziałach 4., 5. i 8.

```
>>> set(open('script1.py'))
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}
>>> {line for line in open('script1.py')}
{'print(sys.path)\n', 'x = 2\n', 'print(2 ** 33)\n', 'import sys\n'}
>>> {ix: line for ix, line in enumerate(open('script1.py'))}
{0: 'import sys\n', 1: 'print(sys.path)\n', 2: 'x = 2\n', 3: 'print(2 ** 33)\n'}
```

Zbiory i słowniki rozwijane potrafią również wykorzystać rozszerzoną składnię list rozwijanych, które omówiliśmy wcześniej w tym rozdziale, w tym również warunek `if`:

```
>>> {line for line in open('script1.py') if line[0] == 'p'}
{'print(sys.path)\n', 'print(2 ** 33)\n'}
>>> {ix: line for (ix, line) in enumerate(open('script1.py')) if line[0] == 'p'}
{1: 'print(sys.path)\n', 3: 'print(2 ** 33)\n'}
```

Podobnie jak w przykładzie z listą rozwijaną, w tym przypadku również przeglądamy plik wiersz po wierszu, wybieramy z niego wiersze rozpoczynające się literą „`p`”. W wyniku powstaje zbiór i słownik, ale sporo pracy mamy wykonane za darmo dzięki połączeniu iteracji w pliku oraz składni rozwinięcia.

Jest jeszcze jeden kontekst iteracyjny, o którym z pewnością warto wspomnieć, choć na razie w formie ciekawostki. Więcej informacji na temat specjalnej formy `*arg` znajdziemy w rozdziale 18. Chodzi o metodę zapisu i rozpakowywania listy argumentów funkcji. Jak można się domyślić, składnia ta obsługuje również dowolny obiekt iterowany, w tym także pliki (więcej szczegółów na temat składni można znaleźć w rozdziale 18.).

```
>>> def f(a, b, c, d): print(a, b, c, d, sep='&')
...
>>> f(1, 2, 3, 4)
1&2&3&4
>>> f(*[1, 2, 3, 4])                                     # Rozpakowanie listy argumentów
1&2&3&4

>>> f(*open('script1.py'))                                # Iteracja po wierszach
import sys
&print(sys.path)
&x = 2
&print(2 ** 33)
```

Wrzeczywiście, w związku z tym, że składnia rozpakowywania argumentów akceptuje obiekty iterowane, możemy również wykorzystać funkcję wbudowaną `zip()` do rozpakowywania zzipowanych krotek lub w przypadku zagnieżdżenia wywołań funkcji `zip()` (uwaga, czytanie poniższego kodu jest niewskazane dla osób o szczególnie słabych nerwach!).

```
>>> X = (1, 2)
>>> Y = (3, 4)
>>>
>>> list(zip(X, Y))                                      # zip() krotek: zwraca obiekt iterowany
[(1, 3), (2, 4)]
>>>
>>> A, B = zip(*zip(X, Y))                             # Rozpakowanie zipa!
>>> A
(1, 2)
>>> B
(3, 4)
```

Istnieją również inne narzędzia Pythona zwracające obiekty iterowane zamiast gotowych list wyników. Są to między innymi funkcja `range()` oraz obiekty widoku słownika. W kolejnym podrozdziale omówimy szczegółowo nowości w protokole iteracyjnym wprowadzone w Pythonie 3.0.

Nowe obiekty iterowane w Pythonie 3.0

Jedną z podstawowych cech Pythona 3.0 jest to, że w porównaniu z wersjami 2.x kładzie silniejszy nacisk na iteratory. Obok iteratorów związanych z typami wbudowanymi, jak pliki czy słowniki, metody słowników `keys()`, `values()` i `items()` również zwracają obiekty iterowane. Podobna zmiana spotkała funkcje wbudowane `range()`, `map()`, `zip()` i `filter()`. Jak mieliśmy okazję przekonać się w poprzednim podrozdziale, ostatnie trzy z nich przetwarzają obiekty iterowane, jak również je zwracają. Konstrukcje te w Pythonie 3.0 generują wyniki na żądanie, zamiast zwracać gotowe listy, jak ma to miejsce w Pythonie 2.6.

Takie podejście oszczędza pamięć, ale w niektórych kontekstach może mieć wpływ na styl programowania. W kilku miejscach mieliśmy już okazję zademonstrować konieczność przekształcenia obiektu iterowanego na listę w celu wyświetlenia wszystkich wyników naraz.

```
>>> zip('abc', 'xyz')                                # Obiekt iterowany w 3.0, lista w 2.6
<zipped object at 0x02E66710>
>>> list(zip('abc', 'xyz'))                         # Wymuszenie listy wyników w 3.0
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

W Pythonie 2.6 nie ma takiej potrzeby, ponieważ takie funkcje jak `zip()` zwracają gotową listę wyników. W 3.0 zwracają obiekt iterowany, który generuje kolejne wyniki na żądanie. Oznacza to, że w przypadku konieczności pobrania wszystkich wyników naraz, na przykład w celu wyświetlenia ich w konsoli, musimy dokonać dodatkowego przekształcenia na listę. Przyjrzymy się niektórym nowościom Pythona 3.0 związanym z iteratorami.

Iterator `range()`

Podstawowe zastosowanie funkcji `range()` omówiliśmy w poprzednim rozdziale. W Pythonie 3.0 funkcja `range()` zwraca iterator, który na żądanie generuje liczby z zadanego zakresu bez użycia pamięci na całą listę wyników. Funkcja ta jest odpowiednikiem i następcą funkcji `xrange()` z Pythona 2.6 (patrz uwaga na temat zgodności). Aby uzyskać listę kolejnych wartości z zadanego zakresu (na przykład w celu ich wypisania w konsoli), należy zastosować `list(range(...))`:

```
C:\misc> c:\python30\python
>>> R = range(10)                                     # range() zwraca iterator, nie listę
>>> R
range(0, 10)

>>> I = iter(R)                                      # Przekształcenie zakresu na iterator
>>> next(I)                                         # Przejście do następnej wartości
0                                                    # Pętle for, rozwinięcia itp. wykonują to w sposób niejawny
>>> next(I)
1
>>> next(I)
2
>>> list(range(10))                                 # Przekształcenie na listę
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

W przeciwnieństwie do list zwracanych w Pythonie 2.x obiekt zakresu w Pythonie 3.0 obsługuje iterację, odczyt po indeksie i funkcję `len()`. Nie obsługuje pozostałych operacji typowych dla sekwencji (aby je wykorzystać, należy przekształcić zakres na listę z użyciem funkcji `list()`).

```
>>> len(R)                                # range obsługuje len() i odczyt po indeksie
10
>>> R[0]
0
>>> R[-1]
9

>>> next(I)                               # Kontynuacja iteratora od ostatniego miejsca
3
>>> I.__next__()                         # .next() działa tak samo jak __next__(), ale zaleca się stosować next()
4
```



Uwaga na temat zgodności: Python 2.x obsługuje funkcję wbudowaną `xrange()`, która działa analogicznie do `range()`, lecz zamiast listy liczb z zakresu zwraca obiekt generujący kolejne wartości na żądanie. To dokładnie to samo co w Pythonie 3.0 robi funkcja `range()`, zatem funkcja `xrange()` nie jest już dostępna w 3.0. W kodzie 2.x nadal można znaleźć jej wywołania, szczególnie w sytuacjach, gdy jej odpowiednik `range()` jest mało efektywny pod względem użycia pamięci. Jak wspomniano w poprzednim rozdziale, z podobnych powodów w 3.0 została porzucona metoda `file.readline()`, która w Pythonie 3.0 doczekała się następcy w postaci użycia iteratorów w standardowych operacjach plikowych.

Iteratory `map()`, `zip()` i `filter()`

Funkcje wbudowane `map()`, `zip()` i `filter()` w Pythonie 3.0 stały się iteratorami, podobnie jak funkcja `range()`. Dzięki temu swoje wyniki generują na bieżąco, bez umieszczania w pamięci całej listy wyników naraz. Funkcje te już w 2.x mogły być używane do przetwarzania obiektów iterowanych, a od 3.0 zwracają wynik będący obiektem iterowanym. Jednak w przeciwieństwie do funkcji `range()` zwracającej obiekt, na którym można wykonać wiele równoległych niezależnych iteracji, funkcje te zwracają samodzielne iteratory, które „zużywają” swoje wyniki w trakcie przetwarzania. Innymi słowy, na takich wynikach nie można utworzyć kilku niezależnych iteratorów przetwarzających je równolegle w różnych miejscach iteracji.

Poniższy listing przedstawia przykład użycia funkcji wbudowanej `map()`, którą przedstawiliśmy w poprzednim rozdziale. Podobnie jak w przypadku innych obiektów iterowanych, możemy w miarę potrzeby przekształcić na listę wyniki funkcji `map()`, ale domyślna mechanika iteratorów pozwala w przypadku dużych porcji danych zaoszczędzić sporą ilość pamięci.

```
>>> M = map(abs, (-1, 0, 1))                # map() zwraca iterator, nie listę
>>> M
<map object at 0x0276B890>
>>> next(M)                                 # Ręczne użycie iteratora, „zużywa” wyniki
1                                         # Iterator nie obsługuje funkcji len() ani indeksowania
>>> next(M)
0
>>> next(M)
1
>>> next(M)
StopIteration
>>> for x in M: print(x)                    # Iterator zwrocony z map() nie zawiera już żadnych elementów
...
>>> M = map(abs, (-1, 0, 1))                # Ponowne przetwarzanie wymaga stworzenia nowego iteratora
```

```

>>> for x in M: print(x)                                # Konteksty iteracyjne automatycznie używają funkcji next()
...
1
0
1
>>> list(map(abs, (-1, 0, 1)))                      # Iterator można przekształcić na listę
[1, 0, 1]

```

Funkcja wbudowana `zip()` przedstawiona w poprzednim rozdziale również zwraca iterator działający na tych samych zasadach:

```

>>> Z = zip((1, 2, 3), (10, 20, 30))      # zip() działa tak samo: zwraca jednorazowy iterator
>>> Z
<zip object at 0x02770EE0>

>>> list(Z)
[(1, 10), (2, 20), (3, 30)]

>>> for pair in Z: print(pair)                # Po pełnym przebiegu elementy są wyczerpane...
...
>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> for pair in Z: print(pair)                # Iterator może być użyty ręcznie lub automatycznie
...
(1, 10)
(2, 20)
(3, 30)

>>> Z = zip((1, 2, 3), (10, 20, 30))
>>> next(Z)
(1, 10)
>>> next(Z)
(2, 20)

```

Zbliżone właściwości ma wynik działania funkcji wbudowanej `filter()`, której bardziej szczegółowo przyjrzymy się w dalszej części książki. Funkcja ta filtryuje podany obiekt iterowany na podstawie zadanej funkcji zwracającej wartość logiczną lub jej odpowiednik (jak już wspominaliśmy, w Pythonie każda niepusta wartość jest odpowiednikiem wartości `True`).

```

>>> filter(bool, ['spam', '', 'ni'])
<filter object at 0x0269C6D0>
>>> list(filter(bool, ['spam', '', 'ni']))
['spam', 'ni']

```

Jak większość funkcji omawianych w tym punkcie, funkcja `filter()` w Pythonie 3.0 przetwarza obiekt iterowany i w wyniku zwraca inny obiekt iterowany.

Kilka iteratorów na tym samym obiekcie

Warto zwrócić uwagę na różnice między wynikiem funkcji wbudowanej `range()` a wynikami innych funkcji wbudowanych omówionych w niniejszym rozdziale. Wynik ten obsługuje sprawdzanie długości z użyciem funkcji `len()` i indeksowanie i nie jest iteratorem (to znaczy w kontekście iteracyjnym iterator jest tworzony przez wywołanie funkcji `iter()`). Co najciekawsze, obiekt ten obsługuje wielokrotną równoległą iterację elementów.

```

>>> R = range(3)                                     # range() pozwala na wielokrotną iterację
>>> next(R)
TypeError: range object is not an iterator

>>> I1 = iter(R)                                    # Dwa iteratory jednego obiektu

```

```

>>> next(I1)
0
>>> next(I1)                                # I1 znajduje się na innej pozycji niż I2
1
>>> I2 = iter(R)
>>> next(I2)
0
>>> next(I1)
2

Dla odróżnienia: wyniki funkcji zip(), map() i filter() nie obsługują wielokrotnej niezależnej iteracji.

>>> Z = zip((1, 2, 3), (10, 11, 12))
>>> I1 = iter(Z)
>>> I2 = iter(Z)                            # Dwa iteratory jednego wyniku funkcji zip()
>>> next(I1)
(1, 10)
>>> next(I1)
(2, 11)
>>> next(I2)                                # I2 znajduje się na tej samej pozycji co I1!
(3, 12)

>>> M = map(abs, (-1, 0, 1))                # Tak samo działa wynik map() oraz filter()
>>> I1 = iter(M); I2 = iter(M)
>>> print(next(I1), next(I1), next(I1))
1 0 1
>>> next(I2)
StopIteration

>>> R = range(3)                            # Natomiast wynik range() pozwala na wielokrotną iterację
>>> I1, I2 = iter(R), iter(R)
>>> [next(I1), next(I1), next(I1)]
[0 1 2]
>>> next(I2)
0

```

W rozdziale 29. zajmiemy się tworzeniem własnych klas iterowanych i zobaczymy, że obiekty obsługujące wielokrotną iterację w wyniku działania funkcji `iter()` zwracają z reguły nowy, specjalizowany obiekt iteratora, natomiast obiekty nieobsługujące wielokrotnej iteracji same są iteratorami, przez co w wyniku funkcji `iter()` zwracają same siebie. W rozdziale 20. dowiemy się ponadto, że *funkcje i wyrażenia generatorów* zachowują się w sposób zbliżony do wyniku funkcji `zip()`, obsługując pojedynczą iterację. W tym samym rozdziale będziemy mieli okazję przeanalizować subtelne konsekwencje prób wykorzystania obiektów nieobsługujących wielokrotnej iteracji w pętlach, w których usiłuje się przeglądać je wiele razy.

Iteratory widoku słownika

W rozdziale 8. wspomniałem o tym, że w Pythonie 3.0 metody słownikowe `keys()`, `values()` i `items()` zwracają iterowane obiekty *widoku* słownika, a nie gotowe listy wartości, jak to miało miejsce w poprzednich wersjach języka. Obiekty widoku słownika zachowują kolejność elementów słownika i odzwierciedlają wprowadzane zmiany. Wzbogaceni o wiedzę dotyczącej iteratorów możemy poznać pozostałe szczegóły iteratorów widoku słownika.

```

>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()                           # W 3.0 obiekt widoku, nie lista

```

```

>>> K
<dict_keys object at 0x026D83C0>

>>> next(K)                                     # Widoki nie są iteratorami
TypeError: dict_keys object is not an iterator

>>> I = iter(K)
>>> next(I)                                     # Widoki dostarczają iteratory,
' a'                                              # które można obsługiwać ręcznie,
>>> next(I)                                     # ale nie obsługują len() i indeksowania
' c'

>>> for k in D.keys(): print(k, end=' ')      # W kontekstach iteracyjnych odbywa się to automatycznie
...
a c b

```

Widok słownika, podobnie jak wszystkie iteratory, można przekształcić w listę, wykorzystując funkcję wbudowaną `list()`. Z reguły nie jest to konieczne, chyba że w celu wyświetlenia na ekranie zawartości widoku w trybie interaktywnym lub w celu wykonania operacji typowych dla list, jak indeksowanie:

```

>>> K = D.keys()
>>> list(K)                                     # Przekształcenie na listę
['a', 'c', 'b']

>>> V = D.values()                            # Podobnie dla wyniku metod values() i items()
>>> V
<dict_values object at 0x026D8260>
>>> list(V)
[1, 3, 2]

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (k, v) in D.items(): print(k, v, end=' ')
...
a 1 c 3 b 2

```

Dodatkowo w Pythonie 3.0 słowniki oferują własny iterator zwracający kolejne klucze. Dzięki temu w takim kontekście nie ma konieczności bezpośredniego używania metody `keys()`:

```

>>> D
{'a': 1, 'c': 3, 'b': 2}                      # Słowniki oferują własny iterator
                                                # w każdej iteracji zwracający kolejny klucz
>>> I = iter(D)
>>> next(I)
'a'
>>> next(I)
'c'

>>> for key in D: print(key, end=' ')          # Metoda keys() nie jest niezbędna do iteracji po kluczu,
...                                            # ale w 3.0 keys() również zwraca obiekt iterowany!
a c b

```

Należy pamiętać, że metoda `keys()` nie zwraca listy kluczy, więc tradycyjnie stosowany w starszych wersjach sposób przeglądania słownika po posortowanej liście kluczy nie zadziała w 3.0. W takim przypadku należy w pierwszej kolejności przekształcić wynik metody `keys()` na listę i dopiero na niej wykonać funkcję `sorted()`, można również wykonać funkcję `sorted()` na samym słowniku:

```

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D.keys()): print(k, D[k], end=' ')

```

```
...
a 1 b 2 c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D): print(k, D[k], end=' ')    # Zalecana technika sortowania kluczy
...
a 1 b 2 c 3
```

Inne zagadnienia związane z iteratorami

W rozdziale 20. omówimy więcej zagadnień związanych z listami rozwijanymi i iteratorami, tym razem w powiązaniu z funkcjami, natomiast w rozdziale 29. wróćmy do tych zagadnień w kontekście klas. Między innymi omówimy następujące własności:

- funkcje zdefiniowane przez użytkownika mogą być przekształcone w *generatory obiektów iterowanych* — służy do tego instrukcja `yield`,
- listy rozwijane ujęte w nawiasy okrągłe (zamiast kwadratowych) *zwracają obiekty iterowane*,
- klasy zdefiniowane przez użytkownika mogą obsługiwać protokół iteracyjny dzięki *przeciążeniu* metod `__iter__()` lub `__getitem__()`.

Iteratory zdefiniowane przez użytkownika pozwalają na użycie dowolnego obiektu lub funkcji w dowolnym z poznanych przez nas kontekstów iteracyjnych.

Podsumowanie rozdziału

W niniejszym rozdziale zapoznaliśmy się z instrukcjami pętli Pythona. Po raz pierwszy przyrzeliśmy się także listom składanym oraz *protokolowi iteracji* Pythona — sposobowi, dzięki któremu obiekty niebędące sekwencjami mogą być przetwarzane sekwencyjnie w pętlach. Jak widzieliśmy, listy składane stosujące wyrażenia do wszystkich elementów obiektu, po którym można iterować, przypominają pętle `for`. Przyrzeliśmy się różnym funkcjom wbudowanym obsługującym protokół iteracji, jak również przeanalizowaliśmy zmiany w tym zakresie wprowadzone w Pythonie 3.0.

To kończy nasze omówienie poszczególnych instrukcji proceduralnych. Następny rozdział zamknie tę część książki, omawiając opcje dokumentacji kodu Pythona. Dokumentacja jest również częścią ogólnego modelu składni, a także istotnym elementem dobrze napisanych programów. W kolejnym rozdziale, poprzedzającym zagadnienia związane z większymi strukturami, takimi jak funkcje, zajmiemy się również zbiorem ćwiczeń dotyczących tej części książki. Jak zawsze jednak najpierw należy za pomocą quizu przećwiczyć kwestie przedstawione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób powiązane są pętle `for` i iteratory?
2. W jaki sposób powiązane są pętle `for` i listy składane?

3. Należy podać cztery konteksty iteracyjne w języku Python.
4. Jaki jest obecnie w Pythonie najlepszy sposób wczytania wiersza po wierszu z pliku tekstuowego?
5. Jakiego rodzaju broni oczekiwali byśmy po Hiszpańskiej Inkwizycji?

Sprawdź swoją wiedzę — odpowiedzi

1. Pętla `for` wykorzystuje *protokół iteracji* do przejścia elementów obiektu, po którym iteruje. Z każdą iteracją wywołuje metodę `next` obiektu i przechwytuje wyjątek `StopIteration` w celu ustalenia, kiedy należy zakończyć pętlę.
2. Obie są narzędziami iteracyjnymi. Listy składane są zwięzłym i wydajnym sposobem wykonania często wykorzystawanego zastosowania pętli `for` — zebrania wyników zastosowania wyrażenia do wszystkich elementów przechodzonego obiektu. Listę składaną zawsze można przełożyć na pętlę `for`, a część wyrażenia listy składanej wygląda nawet jak nagłówek pętli `for`.
3. Konteksty iteracyjne w Pythonie obejmują pętlę `for`, listy składane, wbudowaną funkcję `map()`, wyrażenie sprawdzające przynależność `in`, a także wbudowane funkcje `sorted()`, `sum()`, `any()` oraz `all()`. Do kategorii tej zaliczamy również wbudowane funkcje `list()` i `tuple()`, metodę łańcuchów znaków `join` i przypisania sekwencji — wszystkie wykorzystują protokół iteracji (metodę `next`) do przejścia obiektów po jednym elemencie na raz.
4. Najlepszą metodą wczytywania wierszy pliku tekstuowego wcale nie jest obecnie jawne ich wczytywanie. Zamiast tego należy otworzyć plik wewnątrz kontekstu iteracyjnego, takiego jak pętla `for` lub lista składana, i pozwolić narzędziu iteracyjnemu na automatyczne przejście pliku wiersz po wierszu dzięki wykonaniu metody `next()` pliku z każdą iteracją. Takie podejście jest najlepsze z punktu widzenia prostoty kodu, szybkości wykonania i wymagań związanych z miejscem w pamięci.
5. Zaakceptuję każdą z następujących poprawnych odpowiedzi: strach, zastraszenie, piękne czerwone mundury, wygodna kanapa i miękkie poduszki.

Wprowadzenie do dokumentacji

Ta część książki kończy się przedstawieniem technik i narzędzi wykorzystywanych w dokumentowaniu kodu Pythona. Choć kod Pythona został zaprojektowany tak, by sam z siebie był czytelny, kilka dobrze rozmieszczonego, zrozumiałych dla ludzi komentarzy może pomóc innym osobom pojąć sposób działania naszego programu. Python zawiera składnię oraz narzędzia ułatwiające dokumentację kodu.

Choć jest to koncepcja związana raczej z narzędziami, prezentuję ją w tym miejscu książki częściowo dlatego, że dotyczy ona modelu składni Pythona, a po części z myślą o osobach, które próbują zrozumieć zbiór narzędzi Pythona. Z tego drugiego powodu rozszerzę tutaj nieco wskazówki dotyczące dokumentacji, podane jeszcze w rozdziale 4. Jak zwykle rozdział ten kończy się ostrzeżeniami dotyczącymi często spotykanych pułapek, quizem końcowym, a także zbiorem ćwiczeń podsumowujących tę część książki.

Źródła dokumentacji Pythona

W tym miejscu książki większość osób powinna już zaczynać rozumieć, że Python zawiera niesamowitą ilość gotowych możliwości i opcji — wbudowanych funkcji i wyjątków, zdefiniowanych atrybutów i metod obiektów, modułów biblioteki standardowej. Co więcej, tak naprawdę jedynie powierzchownie przyjrzelismy się każdej z tych kategorii.

Jednym z pytań zadawanych przez oszołomione osoby poczatkujące jest często: „Gdzie mogę znaleźć informacje na temat tych wszystkich wbudowanych narzędzi?”. Niniejszy podrozdział zawiera wskazówki dotyczące różnych źródeł dokumentacji dostępnych w Pythonie. Prezentuje również łańcuchy znaków dokumentacji (tak zwane *docstrings*) i system *PyDoc*, który z nich korzysta. Te zagadnienia pozostają może na peryferiach samego języka, jednak stają się niezbędną wiedzą, kiedy nasz kod osiąga poziom przykładów i ćwiczeń z tej części książki.

W tabeli 15.1 przedstawiono różne miejsca, w których można szukać informacji dotyczących Pythona, zgodnie ze wzrastającym poziomem objętości. Ponieważ dokumentacja jest tak istotnym narzędziem praktycznego programowania, każdą z tych kategorii omówimy za chwilę z osobna.

Tabela 15.1. Źródła dokumentacji Pythona

Forma	Rola
Komentarze ze znakiem #	Dokumentacja w pliku
Funkcja <code>dir</code>	Lista atrybutów dostępnych w obiektach
Łańcuchy znaków dokumentacji — <code>__doc__</code>	Dokumentacja w pliku dołączana do obiektów
PyDoc — funkcja <code>help</code>	Interaktywna pomoc dla obiektów
PyDoc — raporty HTML	Dokumentacja modułów w przeglądarce
Zbiór standardowej dokumentacji	Oficjalne opisy języka i biblioteki
Zasoby internetowe	Samouczki, przykłady, artykuły dostępne w Internecie
Publikowane książki	Komercyjne teksty i podręczniki

Komentarze ze znakami

Komentarze ze znakami # są najbardziej podstawowym sposobem udokumentowania kodu. Python po prostu ignoruje tekst znajdujący się za znakiem # (o ile nie znajduje się on wewnątrz literala łańcucha znaków), dlatego można po nim umieścić słowa i opisy znaczące dla programistów. Takie komentarze dostępne są jednak jedynie w plikach źródłowych. By zamieścić w kodzie komentarze dostępne nieco szerzej, trzeba będzie skorzystać z łańcuchów znaków dokumentacji.

Zalecane obecnie praktyki mówią, że łańcuchy znaków dokumentacji najlepiej nadają się do udokumentowania większej funkcjonalności (typu „mój plik robi...”), a komentarze ze znakami # lepiej jest ograniczać do udokumentowania mniejszych części kodu (na przykład „to dziwne wyrażenie wykonuje...”). Więcej informacji na temat łańcuchów znaków dokumentacji za moment.

Funkcja `dir`

Wbudowana funkcja `dir` to łatwa metoda pobrania listy wszystkich atrybutów dostępnych wewnątrz obiektu (to jest jego metod i prostszych elementów danych). Może być wywołana na dowolnym obiekcie posiadającym atrybuty. By na przykład dowiedzieć się, co udostępniane jest w module `sys` biblioteki standardowej, wystarczy ten moduł zaimportować i przekazać do funkcji `dir` (poniższe wyniki pochodzą z Pythona 3.0; w wersji 2.6 mogą być nieco inne).

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
 '__stderr__', '__stdin__', '__stdout__', '__clear_type_cache__', '__current_frames__',
 '__getframe__', 'api_version', 'argv', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook', 'dllhandle',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix', 'executable',
 'exit', 'flags', 'float_info', 'getcheckinterval', 'getdefaultencoding',
 ...pozostałe pominięto...]
```

Na listingu pokazano jedynie część zwracanych nazw. By zobaczyć pełną ich listę, należy wykonać te instrukcje na własnym komputerze.

By dowiedzieć się, jakie atrybuty udostępniane są we wbudowanych typach obiektów, należy wykonać funkcję `dir` na dowolnym literale (lub istniejącej instancji) pożdanego typu. Żeby na przykład zobaczyć atrybuty list i łańcuchów znaków, można przekazać do funkcji `dir` puste obiekty tego typu.

```
>>> dir([])
['__add__', '__class__', '__contains__', ...więcej... 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']

>>> dir('')
['__add__', '__class__', '__contains__', ...więcej... 'capitalize', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex', 'rjust', ...pozostałe pominięto...]
```

Wynik wywołania funkcji `dir` dla dowolnego typu wbudowanego zawiera listę atrybutów powiązanych z implementacją tego typu (z technicznego punktu widzenia — metod przeciążających operatorów). Wszystkie one zaczynają i kończą się podwójnymi znakami `_` by odróżnić się od pozostałych. W tym punkcie książki możemy je spokojnie zignorować.

Jak się okazuje, ten sam efekt można osiągnąć, przekazując do `dir` nazwę typu zamiast jego literala.

```
>>> dir(str) == dir('')
True
# Ten sam wynik co w poprzednim przykładzie
>>> dir(list) == dir([])
True
```

To rozwiązanie działa, ponieważ nazwy takie, jak `str` i `list`, które kiedyś były funkcjami-konwerterami typów, obecnie są w Pythonie nazwami typów. Wywołanie jednej z nich powoduje wywołanie konstruktora, który generuje instancję określonego typu. Więcej informacji na temat konstruktorów oraz metod przeciążania operatorów zamieścimy przy okazji omawiania klas w szóstej części książki.

Funkcja `dir` udostępnia nam swego rodzaju zagadkę. Podaje listę nazw atrybutów, jednak nie mówi nam nic na temat tego, co te nazwy oznaczają. Po takie dodatkowe informacje należy sięgnąć do kolejnego źródła dokumentacji.

Łańcuchy znaków dokumentacji — `__doc__`

Oprócz komentarzy ze znakami `#` Python obsługuje również dokumentację automatycznie dołączaną do obiektów i przechowuje ją do przeglądania w czasie wykonywania. Z punktu widzenia składni komentarze takie są łańcuchami znaków znajdującymi się na górze plików modułów i instrukcji klas, przed całym kodem wykonywalnym (komentarze ze znakami `#` można umieszczać przed nimi). Python automatycznie wstawia łańcuchy znaków (znane jako *łańcuchy znaków dokumentacji*, czyli *docstrings*) do atrybutów `__doc__` odpowiednich obiektów.

Łańcuchy znaków dokumentacji zdefiniowane przez użytkownika

Rozważmy na przykład poniższy plik, `docstrings.py`. Jego łańcuchy znaków dokumentacji pojawiają się na początku pliku i na początku funkcji oraz klasy w niej się znajdującej. W kodzie w formie komentarzy wielowierszowych wykorzystałem blokowy łańcuch znaków z potrójnymi cudzysłowami, jednak może go zastąpić dowolny inny rodzaj łańcucha. Nie omawialiśmy jeszcze instrukcji `def` i `class`, dlatego należy w nich zignorować wszystko z wyjątkiem łańcuchów znaków na górze.

```

"""
Dokumentacja modułu
Tutaj jego opis
"""

spam = 40

def square(x):
    """
    Dokumentacja funkcji
    Możemy wziąć Państkę wątrobę?
    """
    return x **2                      # Kwadrat

class Employee:
    "dokumentacja klasy"
    pass

print(square(4))
print(square.__doc__)

```

Celem protokołu dokumentacji jest to, żeby komentarze zostały zachowane do przejrzenia w atrybutach `__doc__` po zimportowaniu pliku. By zatem wyświetlić łańcuchy znaków dokumentacji powiązane z modelem i jego obiektami, należy zimportować plik i wyświetlić dla tych elementów atrybuty `__doc__`, w których Python zapisał tekst.

```

>>> import docstrings
16

Dokumentacja funkcji
Możemy wziąć Państkę wątrobę?

>>> print(docstrings.__doc__)

Dokumentacja modułu
Tutaj jego opis

>>> print(docstrings.square.__doc__)

Dokumentacja funkcji
Możemy wziąć Państkę wątrobę?

>>> print(docstrings.Employee.__doc__)
dokumentacja klasy

```

Warto zauważyć, że w przypadku chęci wyświetlania łańcuchów znaków dokumentacji zazwyczaj będziemy chcieli użyć instrukcji `print`. Inaczej otrzymamy jeden łańcuch znaków z osadzonymi znakami nowych wierszy.

Można również dołączyć łańcuchy znaków dokumentacji do *metod* klas (omówionych w szóstej części książki), jednak ponieważ są one po prostu instrukcjami `def` zagnieżdzonymi w instrukcjach `class`, nie są żadnym przypadkiem specjalnym. By pobrać łańcuch znaków dokumentacji dla funkcji metody znajdującej się w klasie modułu, wystarczy rozszerzyć ścieżkę i dojść do metody przez klasę — `moduł.klasa.metoda.__doc__` (przykład łańcucha znaków dokumentacji dla metody można znaleźć w rozdziale 28.).

Standardy dotyczące łańcuchów znaków dokumentacji

Nie istnieje żaden uniwersalny standard określający, co powinno się znaleźć w tekście łańcucha znaków dokumentacji (choć niektóre firmy wprowadzają tutaj własne, wewnętrzne reguły). Pojawiło się kilka propozycji dotyczących języków znaczników i szablonów (na przykład HTML

lub XML), jednak wydaje się, że w świecie Pythona się one nie przyjęły. I szczerze mówiąc, przekonanie programistów Pythona, by dokumentowali swój kod za pomocą ręcznie pisanej kodu HTML, nie nastąpi raczej za naszego życia...

Dokumentacja w ogóle ma wśród programistów niski priorytet. Zazwyczaj jeśli w ogóle w pliku znajdują się jakiekolwiek komentarze, możemy zaliczać się do szczęśliwców. Ja sam zachęcam jednak każdego do obfitego dokumentowania kodu — dokumentacja naprawdę jest bardzo istotną częścią dobrze napisanego programu. Problem polega na tym, że nie istnieje obecnie żaden standard dotyczący struktury łańcuchów znaków dokumentacji. Jeśli chcemy z nich korzystać, należy to po prostu robić.

Wbudowane łańcuchy znaków dokumentacji

Jak się okazuje, wbudowane moduły i obiekty Pythona wykorzystują podobne techniki do dołączania dokumentacji wykraczającej poza listę atrybutów zwracaną przez funkcję `dir`. By na przykład zobaczyć czytelny dla człowieka opis wbudowanego modułu, należy go zaimportować i wyświetlić jego łańcuch znaków `__doc__`.

```
>>> import sys
>>> print(sys.__doc__)
This module provides access to some objects used or maintained by the interpreter and
↪to functions that interact strongly with the interpreter.

Dynamic objects:

argv -- command line arguments; argv[0] is the script pathname if known
path -- module search path; path[0] is the script directory, else ''
modules -- dictionary of loaded modules
...resztę tekstu pominięto...
```

Opisy funkcji, klas oraz metod wbudowanych modułów również dołączane są za pomocą atrybutów `__doc__`.

```
>>> print(sys.getrefcount.__doc__)
getrefcount(object) -> integer

Return the reference count of object. The count returned is generally one higher than
↪you might expect, because it includes the (temporary) ...resztę tekstu pominięto...
```

Łańcuchy znaków udostępniają również więcej informacji na temat wbudowanych funkcji.

```
>>> print(int.__doc__)
int(x[, base]) -> integer

Convert a string or number to an integer, if possible. A floating point argument will
↪be truncated towards zero (this does not include a ...resztę tekstu pominięto...

>>> print(map.__doc__)
map(func, *iterables) --> map object

Make an iterator that computes the function using arguments from each of the
↪iterables. Stops when the shortest iterable is exhausted.
```

Przeglądając łańcuchy znaków dokumentacji, możemy otrzymać wiele informacji na temat wbudowanych narzędzi. Nie musimy tego jednak robić — funkcja `help`, do której zaraz przejdziemy, robi to za nas automatycznie.

PyDoc — funkcja help

Łańcuchy znaków dokumentacji okazały się na tyle przydatne, że w Pythonie dostępne jest teraz narzędzie ułatwiające ich wyświetlanie. Standardowe narzędzie *PyDoc* to kod Pythona potrafiący łączyć ze sobą łańcuchy znaków dokumentacji i powiązane z nimi informacje strukturalne oraz formatować je w ładnie zaaranżowane raporty różnego typu. Dodatkowe narzędzia służące do ekstrakcji i formatowania łańcuchów znaków dokumentacji dostępne są na licencji open source (w tym narzędzia obsługujące tekst ustrukturyzowany — więcej na ten temat można znaleźć w Internecie), natomiast PyDoc jest częścią biblioteki standardowej Pythona.

PyDoc można uruchomić na wiele sposobów, w tym za pomocą opcji skryptów wiersza poleceń (więcej informacji na ten temat można znaleźć w dokumentacji biblioteki Pythona). Dwa najbardziej znane interfejsy PyDoc to wbudowana funkcja `help` oraz interfejs GUI/HTML dla PyDoc. Funkcja `help` wymusza na PyDoc wygenerowanie prostego tekstowego raportu (wyglądającego jak tak zwana *manpage* z systemów opartych na Uniksie).

```
>>> import sys
>>> help(sys.getrefcount)
Help on built-in function getrefcount in module sys:

getrefcount(...)
    getrefcount(object) -> integer

    Return the reference count of object. The count returned is generally one higher
    ↪than you might expect, because it includes the (temporary) ...resztę tekstu
    ↪pominieto...
```

Warto zauważyć, że aby wywołać funkcję `help`, wcale nie trzeba importować modułu `sys`. By jednak uzyskać pomoc dotyczącą modułu `sys`, trzeba go zaimportować. Funkcja `help` oczekuje przekazania referencji do obiektu. W przypadku większych obiektów, jak moduły i klasy, wyświetlanie `help` podzielone jest na kilka części, z których niektóre pokazane zostały poniżej. By zobaczyć pełny raport, należy wykonać ten kod w sesji interaktywnej.

```
>>> help(sys)
Help on built-in module sys:

NAME
    sys

FILE
    (built-in)

MODULE DOCS
    http://docs.python.org/library/sys

DESCRIPTION
    This module provides access to some objects used or maintained by the interpreter
    ↪and to functions that interact strongly with the interpreter. ...resztę tekstu
    ↪pominieto...

FUNCTIONS
    __displayhook__ = displayhook(...)
        displayhook(object) -> None

        Print an object to sys.stdout and also save it in builtins. ...resztę tekstu
        ↪pominieto...

DATA
    __stderr__ = <io.TextIOWrapper object at 0x0236E950>
```

```
__stdin__ = <io.TextIOWrapper object at 0x02366550>
__stdout__ = <io.TextIOWrapper object at 0x02366E30>
...resztę tekstu pominięto...
```

Część informacji z tego raportu to łańcuchy znaków dokumentacji, a część (na przykład wzorce wywoływania funkcji) to informacje strukturalne zbierane przez PyDoc automatycznie przy okazji inspekcji wewnętrznych mechanizmów obiektów, o ile są one dostępne. Funkcję `help` można również wywołać na wbudowanych funkcjach, metodach oraz typach. By otrzymać pomoc dotyczącą wbudowanego typu, należy użyć nazwy typu (na przykład `dict` dla słownika, `str` dla łańcucha znaków, `list` dla listy). W zamian otrzymamy długi tekst opisujący wszystkie metody dostępne dla określonego typu.

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

class dict(object)
| dict() -> new empty dictionary.
| dict(mapping) -> new dictionary initialized from a mapping object's ...resztę
| tekstu pominięto...

>>> help(str.replace)
Help on method_descriptor:

replace(...)
    S.replace (old, new[, count]) -> str

    Return a copy of S with all occurrences of substring ...resztę tekstu pominięto...
```

```
>>> help(ord)
Help on built-in function ord in module builtins:

ord(...)
    ord(c) -> integer

    Return the integer ordinal of a one-character string.
```

Funkcja `help` na naszych modułach działa tak samo dobrze jak na tych wbudowanych. Poniżej widać jej raport dotyczący utworzonego wcześniej pliku `docstrings.py`. Ponownie część tekstu to łańcuchy znaków dokumentacji, a część to informacje pobierane automatycznie dzięki inspekcji struktury obiektu.

```
>>> import docstrings
>>> help(docstrings.square)
Help on function square in module docstrings:

square(x)
    Dokumentacja funkcji
    Możemy wziąć Pańską wątrobę?

>>> help(docstrings.Employee)
Help on class Employee in module docstrings:

class Employee(builtins.object)
| dokumentacja klasy
|
| Data descriptors defined here:
...resztę tekstu pominięto...

>>> help(docstrings)
Help on module docstrings:
```

```
NAME
    docstrings

FILE
    c:\misc\docstrings.py

DESCRIPTION
    Dokumentacja modułu
    Tutaj jego opis

CLASSES
    builtins.object
    Employee

    class Employee(builtins.object)
        | dokumentacja klasy
        |
        | Data descriptors defined here:
        ...resztę tekstu pominięto...

FUNCTIONS
    square(x)
        Dokumentacja funkcji
        Możemy wziąć Pańską wątrobę?

DATA
    spam = 40
```

PyDoc — raporty HTML

Funkcja `help` przydaje się do pobrania dokumentacji, kiedy pracujemy w trybie interaktywnym. By jednak wyświetlić informacje w bardziej atrakcyjny sposób, PyDoc udostępnia również graficzny interfejs użytkownika (prosty, lecz jednocześnie przenośny skrypt oparty na Pythonie i `tkinter`) i może wygenerować raport w formie strony HTML, którą da się wyświetlić w dowolnej przeglądarce. W tym trybie PyDoc może działać lokalnie lub jako zdalny serwer w trybie klient-serwer. Raporty zawierają automatycznie tworzone odnośniki, które pozwalają na przechodzenie przez dokumentację powiązanych komponentów aplikacji za pomocą prostego klikania.

By uruchomić PyDoc w tym trybie, najpierw trzeba uruchomić graficzny interfejs użytkownika silnika wyszukiwarki, widoczny na rysunku 15.1. Można to zrobić albo wybierając opcję *Module Docs* z menu *Python* dostępnego pod przyciskiem *Start* w systemie Windows, albo uruchamiając skrypt `pydoc.py` znajdujący się w katalogu biblioteki standardowej Pythona `Lib` w systemie Windows (należy uruchomić `pydoc.py` z argumentem `-g` w wierszu poleceń). Należy podać nazwę interesującego nas modułu i nacisnąć przycisk *Enter*. PyDoc przejdzie ścieżkę wyszukiwania importowanych modułów (`sys.path`) w celu odnalezienia referencji do żądaneego modułu.

Po znalezieniu odpowiedniego wpisu należy go wybrać i kliknąć przycisk *go to selected*. PyDoc uruchomi na naszym komputerze przeglądarkę internetową, w której wyświetli raport wygenerowany w formacie HTML. Na rysunku 15.2 widać informacje wyświetlane przez PyDoc dla wbudowanego modułu `glob`.

Warto zwrócić uwagę na odnośniki (łącza) w części *Modules* tej strony. Można je kliknąć w celu przejścia na strony PyDoc poświęcone powiązanym, zaimportowanym modułom. W przypadku większych stron PyDoc generuje również odnośniki do ich części.

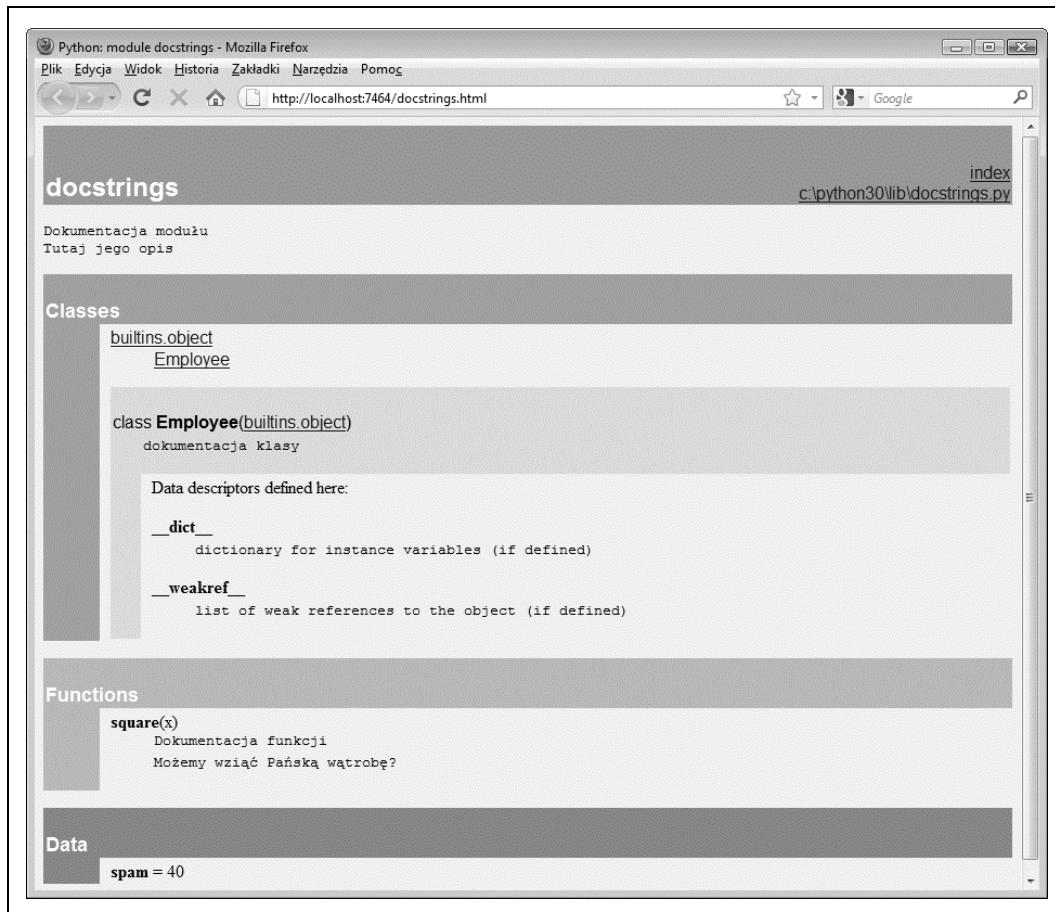


Rysunek 15.1. Graficzny interfejs użytkownika silnika wyszukiwarki najwyższego poziomu w Pythonie. Należy wpisać nazwę szukanego modułu, nacisnąć Enter, wybrać moduł, a później kliknąć „go to selected” (lub pominąć nazwę modułu i kliknąć „open browser” w celu zobaczenia wszystkich dostępnych modułów)



Rysunek 15.2. Po odnalezieniu modułu w graficznym interfejsie użytkownika z rysunku 15.1 i naciśnięciu „go to selected” dokumentacja modułu generowana jest w formacie HTML i wyświetlana w oknie przeglądarki internetowej w sposób podobny do widocznego na rysunku wbudowanego modułu biblioteki standardowej

Tak jak interfejs funkcji help, interfejs graficzny obok modułów wbudowanych działa również na modułach zdefiniowanych przez użytkownika. Na rysunku 15.3 widać stronę wygenerowaną dla naszego pliku modułu *docstrings.py*.



Rysunek 15.3. PyDoc tworzy strony dokumentacji dla modułów wbudowanych oraz tworzonych przez użytkownika. Na rysunku widać stronę modułu zdefiniowanego przez użytkownika, na której znajdują się wszystkie łańcuchy znaków dokumentacji pobrane z pliku źródłowego

PyDoc można dostosować do własnych wymagań i uruchamiać na różne sposoby, których nie będziemy tutaj omawiać.Więcej informacji na ten temat można znaleźć w odpowiednim fragmencie dokumentacji biblioteki standardowej Pythona. Najważniejszą kwestią, jaką należy zapamiętać, jest to, że PyDoc udostępnia nam raporty implementacyjne gratis — jeśli zdecydujemy się w plikach używać łańcuchów znaków dokumentacji, PyDoc wykona całą pracę związaną z zebraniem ich i sformatowaniem w sposób odpowiedni do wyświetlania. PyDoc udostępnia jedynie pomoc dla obiektów takich, jak funkcje i moduły, jednak ułatwia dostęp do dokumentacji średniego poziomu dla takich narzędzi. Raporty z PyDoc są bardziej użyteczne od surowych list atrybutów i mniej wyczerpujące od standardowych podręczników czy dokumentacji.

PyDoc można również uruchomić w celu zapisania dokumentacji HTML dla modułu w pliku, tak by dało się ją przejrzeć później lub wydrukować. Wskazówki dotyczące takiego rozwiązania można znaleźć w dokumentacji PyDoc. Warto również zauważyć, że PyDoc może nie



Niezła sztuczka na dziś związana z PyDoc: Jeśli w górnym polu okna z rysunku 15.1 nie podamy nazwy modułu i naciśniemy przycisk *open browser*, PyDoc wyświetli stronę internetową zawierającą odnośniki do każdego modułu, jaki jest w stanie zainportować na komputerze. Obejmuje to moduły biblioteki standardowej Pythona, moduły zainstalowanych rozszerzeń, moduły zdefiniowane przez użytkownika znajdujące się w ścieżce wyszukiwania, a nawet statycznie lub dynamicznie dołączone moduły napisane w języku C. Takie informacje trudno jest znaleźć samodzielnie bez napisania kodu sprawdzającego zbiór źródeł modułów.

działać dobrze, kiedy będzie uruchomiony na skryptach odczytujących coś ze standardowego wejścia — PyDoc importuje moduł docelowy w celu przejrzenia jego zawartości i może nie mieć połączenia z tekstem standardowego wejścia, kiedy używany jest w trybie GUI. Moduły, które można zainportować bez wymagań w zakresie natychmiastowego wprowadzania danych, zawsze będą jednak działały z PyDoc.

Zbiór standardowej dokumentacji

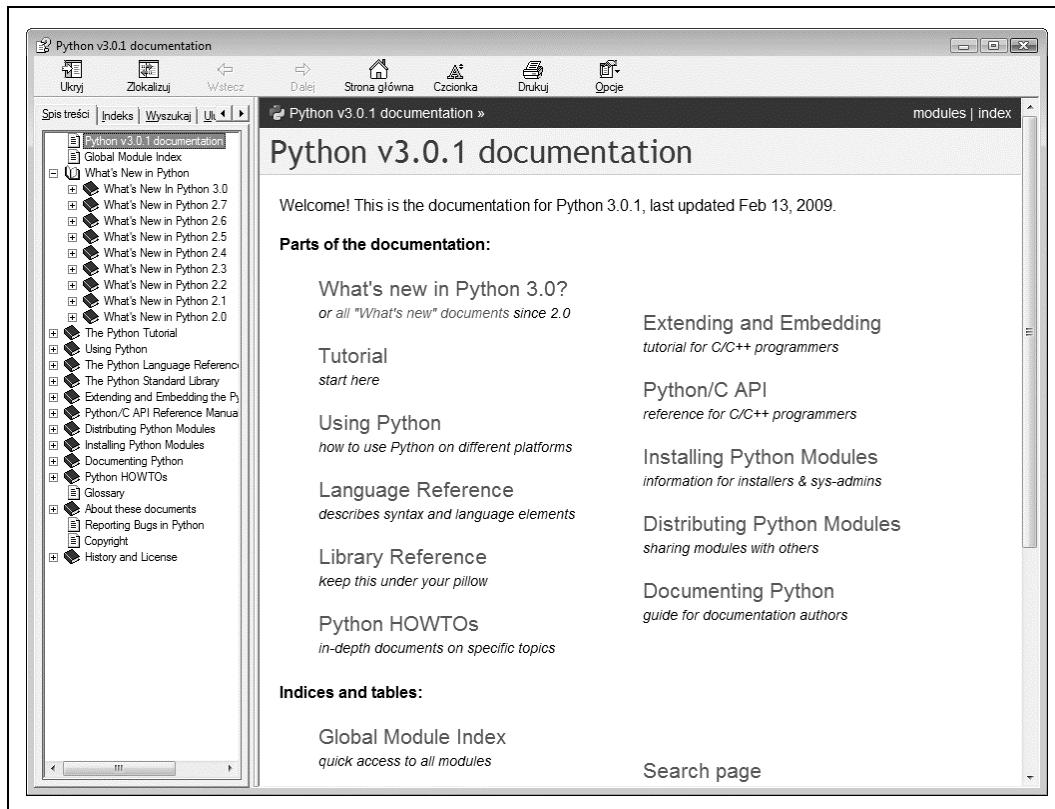
Pełny, najbardziej aktualny opis języka oraz jego zbioru narzędzi można zawsze znaleźć w standardowej dokumentacji Pythona. Jest ona dostępna w formacie HTML i innych, a także instalowana w systemie Windows wraz z Pythonem — dostępna w menu *Python* przycisku *Start*, a także otwierana z menu *Help* wewnętrz IDLE. Dokumentację można również pobrać osobno ze strony internetowej <http://www.python.org> w różnych formatach lub przeglądać ją w Internecie pod tym samym adresem (należy szukać odnośnika *Documentation*). W systemie Windows dokumentacja jest skompilowanym plikiem pomocy, który obsługuje wyszukiwanie, natomiast wersje dostępne w Internecie na stronie *Python.org* zawierają wyszukiwarkę.

Po otwarciu dokumentacji w formacie dla systemu Windows wyświetlana jest strona główna podobna do rysunku 15.4. Dwa najważniejsze wpisy to najprawdopodobniej *Library Reference* (dokumentujący wbudowane typy, funkcje, wyjątki oraz moduły biblioteki standardowej), a także *Language Reference* (udostępniający formalny opis szczegółów na poziomie języka). Pod opcją *Tutorial* można znaleźć krótkie wprowadzenie przeznaczone dla osób początkujących.

Zasoby internetowe

Na oficjalnej stronie internetowej Pythona (<http://www.python.org>) można znaleźć odnośniki do różnych zasobów dotyczących Pythona; niektóre z nich dotyczą pewnych specjalnych tematów czy zastosowań. Po kliknięciu odnośnika *Documentation* można uzyskać dostęp do samouczka, a także przewodnika dla osób początkujących (*Beginner's Guide*). Na stronie tej wymienione są również zasoby nieanglojęzyczne.

W Internecie można dzisiaj znaleźć niezliczoną ilość wiki, blogów, stron internetowych i najróżniejszych innych źródeł poświęconych Pythonowi. By znaleźć społeczność programistów Pythona, wystarczy wyszukać w Google „programowanie w Pythonie” czy „Python programming”.



Rysunek 15.4. Zbiór standardowej dokumentacji Pythona dostępny na stronie www.python.org, z menu Help aplikacji IDLE oraz z odpowiedniego menu przycisku Start w systemie Windows. W systemie Windows to plik pomocy, który można przeszukiwać. Dla wersji publikowanych w Internecie dostępna jest wyszukiwarka. Najczęściej korzysta się z opcji Library Reference

Publikowane książki

Można również siegnąć po jedną z wielu książek dotyczących Pythona. Należy jednak pamiętać, że książki zawsze będą nieco do tyłu w stosunku do najświeższych zmian w tym języku ze względu na pracę włożoną w samo pisanie oraz naturalne opóźnienia wynikające z cyklu wydawniczego. Zazwyczaj kiedy książka zostaje już opublikowana, jest co najmniej trzy do czterech miesięcy w tyle za aktualnym stanem Pythona. W przeciwieństwie do standardowej dokumentacji języka książki nie są również darmowe.

Dla wielu osób wygoda i jakość profesjonalnie publikowanego tekstu są warte tej ceny. Co więcej, Python zmienia się na tyle wolno, że książki poświęcone temu językowi zachowują ważność wiele lat po publikacji, w szczególności jeśli ich autorzy publikują uaktualnienia na swoich stronach internetowych. Wskazówki dotyczące innych książek poświęconych Pythonowi można znaleźć w „Przedmowie”.

Często spotykane problemy programistyczne

Przed przejściem do ćwiczeń do tej części książki przejrzyjmy jeszcze błędy popełniane często przy tworzeniu instrukcji i programów przez osoby poczatkujące. Wiele z nich odnosi się do ostrzeżeń, o których wspomniałem już wcześniej w tej części książki — zostały one tutaj powtórzone w celu zebrania ich w jednym miejscu. W miarę nabierania doświadczenia w tworzeniu kodu w Pythonie każdy nauczy się unikać tych pułapek, jednak kilka dodatkowych uwag może już teraz pomóc w uniknięciu większych wpadek.

- **Nie należy zapominać o dwukropkach.** Zawsze należy pamiętać o umieszczeniu znaku : na końcu nagłówka instrukcji złożonej (pierwszym wierszu `if`, `while` czy `for`). Każdy o tym na początku zapomina (tak samo robiłem ja sam i większość z ponad trzech tysięcy moich studentów), jednak zawsze można się pocieszyć tym, że niedługo wejdzie nam to w nawyk.
- **Należy rozpoczynać kod w pierwszej kolumnie.** Trzeba pamiętać o zaczynaniu kodu najwyższego poziomu (niezagnieżdzonego) w kolumnie numer jeden. Dotyczy to niezagnieżdzonego kodu wpisywanego do plików modułów, ale także niezagnieżdzonego kodu wpisywanego w sesji interaktywnej.
- **Puste wiersze mają w sesji interaktywnej znaczenie.** Puste wiersze w instrukcjach złożonych są zawsze ignorowane w plikach modułów, ale kiedy kod wpisuje się w sesji interaktywnej, taki wiersz kończy instrukcję. Innymi słowy, puste wiersze mówią interaktywnemu wierszowi polecień, że skończyliśmy instrukcję złożoną. Jeśli chcemy ją kontynuować, nie możemy nacisnąć przycisku `Enter` przy znaku zachęty ... (lub w IDLE), dopóki naprawdę nie skończymy.
- **Należy stosować spójną indentację.** O ile nie wiemy, co edytor zrobi z tabulatorami, trzeba unikać mieszania tabulatorów i spacji w indentacji bloku kodu. W przeciwnym razie może się okazać, że to, co widzimy w edytorze, może nie być tym samym, co zobaczy Python, kiedy tabulatory zamieni na pewną liczbę spacji. Tak samo jest w każdym innym języku ustrukturyzowanym blokowo, nie tylko w Pythonie. Jeśli kolejny programista w innym sposobie ustawi tabulatory, nie zrozumie struktury naszego kodu. Bezpieczniej jest używać w każdym bloku tylko tabulatorów lub tylko spacji.
- **Nie należy stosować w Pythonie stylu kodowania z języka C.** Przypomnienie dla programistów języków C i C++: wokół testów nagłówków `if` i `while` (na przykład `if (X == 1)`) nie trzeba umieszczać nawiasów. Można tak zrobić (każde wyrażenie można umieścić w nawiasach), jednak w tym kontekście są one całkowicie zbędne. Nie należy również kończyć wszystkich instrukcji średnikiem. Z technicznego punktu widzenia jest to w Pythonie dopuszczalne, jednak całkowicie bezużyteczne, o ile nie umieszczamy w jednym wierszu więcej niż jednej instrukcji (normalnie instrukcję kończy koniec wiersza). Należy też pamiętać, by nie osadzać instrukcji przypisania w testach pętli `while` i nie używać nawiasów klamrowych (`{}`) wokół bloków kodu (zamiast tego należy w spójny sposób je wcinać).
- **Zamiast while czy range należy używać prostych pętli for.** Kolejne przypomnienie: prosta pętla `for` (na przykład `for x in seq:`) jest prawie zawsze łatwiejsza do zapisania w kodzie i szybsza do wykonania od pętli licznika opartego na `while` czy `range`. Ponieważ Python wewnętrznie obsługuje indeksowanie dla prostego `for`, czasami pętla taka może być nawet dwa razy szybsza od odpowiadającej jej pętli `while`. Nie należy dać się skusić na ręczne odliczanie elementów w Pythonie!

• **Należy uważać na obiekty zmienne w przypisaniach.** Wspomniałem o tym w rozdziale 11.: przypisując obiekty zmienne po kilka za jednym razem (na przykład `a = b = []`), a także korzystając z rozszerzonego przypisania (`a += [1, 2]`), należy bardzo uważać. W obu przypadkach modyfikacje w miejscu mogą wpływać na inne zmienne.Więcej informacji na ten temat można znaleźć w rozdziale 11.

• **Nie należy oczekiwac wyników od funkcji, które modyfikują obiekty w miejscu.** Z tym również spotkaliśmy się już wcześniej — operacje takie, jak metody `list.append` czy `list.sort`, wprowadzone w rozdziale 8., nie zwracają wartości (innych od `None`), dlatego powinno się je wywoływać bez przypisywania wyniku. Osoby poczynające nierzadko tworzą kod w stylu `mylist = mylist.append(X)`, by otrzymać wynik zastosowania metody `append`. Tak naprawdę jednak kod ten przypisuje listę `mylist` do `None`, a nie do zmodyfikowanej listy (i w rezultacie całkowicie tracimy referencję do listy).

Jeszcze bardziej podchwytnym przykładem takiego zachowania jest w Pythonie 2.X próba przejścia posortowanych kluczów słownika. Często spotyka się kod w stylu `for k in D.keys().sort():`. Takie rozwiązanie prawie działa — metoda `keys` buduje listę kluczów, a metoda `sort` ją sortuje — jednak ponieważ metoda `sort` zwraca `None`, pętla nie powiedzie się, gdyż w rezultacie będzie pętlą po `None` (obiekcie niebędącym sekwencją). Taki kod nie działa tym bardziej w Pythonie 3.0, ponieważ klucze słowników są widokami, a nie listami! By w poprawny sposób zapisać w kodzie to wyrażenie, należy albo skorzystać z nowszej funkcji `sorted` (zwracającej posortowaną listę), albo podzielić wywołania metody na instrukcje: `Ks = list(D.keys()), potem Ks.sort(), a na końcu for k in Ks:`. Jest to zresztą jeden z przypadków, w którym w pętli w sposób jawny będziemy chcieli wywołać metodę `keys`, zamiast polegać na iteratorach słowników — iteratory nie potrafią sortować.

• **W wywołaniach funkcji zawsze należy stosować nawiasy.** By wywołać funkcję, należy po jej nazwie umieścić nawiasy — bez względu na to, czy przyjmuje ona argumenty (trzeba zatem użyć kodu `funkcja()`, a nie samej nazwy `funkcja`). W czwartej części książki zobaczymy, że funkcje są po prostu obiektami, które mają specjalną operację — wywołanie wyzwalane za pomocą nawiasów.

Na moich kursach problem ten zdaje się najczęściej występować z plikami. Wiele osób poczynających w celu zamknięcia pliku wpisuje `file.close`, a nie `file.close()`. Ponieważ odniesienie się do funkcji bez jej wywołania jest w Pythonie dopuszczalne, pierwsza, pozbawiona nawiasów wersja kończy się sukcesem, jednak nie zamyka pliku.

• **Nie należy używać rozszerzeń i ścieżek w operacjach importu i przeładowania.** W instrukcjach `import` należy pominąć ścieżki do katalogów oraz rozszerzenia plików (pisze się zatem `import mod`, a nie `import mod.py`). Podstawy modułów omówione zostały w rozdziale 3. i wróćmy do omawiania tej tematyki w piątej części książki. Ponieważ moduły mogą mieć rozszerzenia inne od `.py` (na przykład `.pyc`), zakodowanie danego rozszerzenia na stałe w kodzie jest nie tylko niepoprawne z punktu widzenia składni, ale dodatkowo nie ma większego sensu. Cała składnia ścieżek specyficzna dla danej platformy pochodzi z ustawień ścieżki wyszukiwania modułów, a nie z instrukcji `import`.

Podsumowanie rozdziału

Niniejszy rozdział wprowadził nas w zagadnienia związane z dokumentacją kodu — zarównoisaną przez nas samych dla tworzonych programów, jak i tą dostępną dla wbudowanych narzędzi. Omówiliśmy łańcuchy znaków dokumentacji, a także zasoby dotyczące Pythona dostępne w Internecie i na naszych komputerach. Zobaczyliśmy, jak funkcja `help` oraz interfejs strony internetowej mogą udostępnić dodatkowe źródła dokumentacji. Ponieważ jest to ostatni rozdział tej części książki, przyjrzeliśmy się również często popełnianym błędom z nadzieją uniknięcia ich w przyszłości.

W kolejnej części książki zaczniemy stosować znane nam już koncepcje do większej konstrukcji programu — funkcji. Zanim do tego dojdzie, należy wykonać ćwiczenia dotyczące tej części książki umieszczone na końcu niniejszego rozdziału. A jeszcze wcześniej czas na wykonanie tradycyjnego quizu podsumowującego rozdział.

Sprawdź swoją wiedzę — quiz

1. Kiedy powinniśmy używać łańcuchów znaków dokumentacji zamiast komentarzy ze znakami #?
2. Należy podać trzy sposoby przeglądania łańcuchów znaków dokumentacji.
3. W jaki sposób można uzyskać listę dostępnych atrybutów obiektu?
4. W jaki sposób można otrzymać listę wszystkich modułów dostępnych na komputerze?
5. Jaką książkę o Pythonie należy kupić po tej?

Sprawdź swoją wiedzę — odpowiedzi

1. Łańcuchy znaków dokumentacji (znane jako *docstrings*) uznawane są za lepsze w przypadku większej, funkcjonalnej dokumentacji opisującej użycie modułów, funkcji, klas oraz metod w kodzie. Komentarze ze znakami # są dzisiaj ograniczone do mikrodokumentacji bardziej zawiłych wyrażeń czy instrukcji. Taki podział wynika również z tego, że łańcuchy znaków dokumentacji łatwiej jest znaleźć w pliku źródłowym, a dodatkowo po ekstrakcji można je wyświetlać za pośrednictwem systemu PyDoc.
2. Łańcuchy znaków dokumentacji można zobaczyć, wyświetlając atrybut `__doc__` obiektu, przekazując obiekt do funkcji `help` PyDoc lub wybierając odpowiednie moduły z wyszukiwarki graficznego interfejsu użytkownika PyDoc w trybie klient-serwer. Dodatkowo PyDoc można wykorzystać do zapisania dokumentacji modułu w pliku HTML do późniejszego przeglądania czy drukowania.
3. Wbudowana funkcja `dir(X)` zwraca listę wszystkich atrybutów dołączonych do dowolnego obiektu.
4. Należy uruchomić graficzny interfejs użytkownika PyDoc, nie wypełniać pola z nazwą modułu i wybrać opcję *open browser*. Powoduje to otwarcie okna przeglądarki zawierającej odnośnik do każdego modułu dostępnego w naszych programach.

5. Moją, oczywiście. A tak naprawdę — w „Przedmowie” znajduje się lista kilku książek będących dobrym uzupełnieniem niniejszej, zarówno przewodników po samym języku, jak i tekstuów dotyczących jego praktycznych zastosowań.

Ćwiczenia do części trzeciej

Skoro już wiemy, jak zapisywać w kodzie podstawową logikę programów, poniższe ćwiczenia będą od nas wymagały zaimplementowania określonych zadań za pomocą instrukcji. Najwięcej pracy będzie w ćwiczeniu 4., które pozwala nam sprawdzić alternatywne wersje kodu. Istnieje wiele sposobów układania instrukcji i część nauki Pythona polega na opanowaniu tego, które układy działają lepiej od innych.

Rozwiązania ćwiczeń znajdują się w podrozdziale „Część III” w dodatku B.

1. Zapisywanie w kodzie podstawowych pętli.

- Należy napisać pętlę `for` wyświetlającą kod ASCII dla każdego znaku z łańcucha o nazwie `S`. Do konwersji każdego znaku na kod ASCII należy wykorzystać wbudowaną funkcję `ord(znak)`. W celu sprawdzenia, jak ona działa, należy ją przetestować w sesji interaktywnej.
 - Następnie należy zmodyfikować pętlę w taki sposób, by obliczała ona sumę kodów ASCII dla wszystkich znaków łańcucha.
 - Na koniec należy wprowadzić ostatnią modyfikację, tak by pętla zwracała listę zawierającą kody ASCII każdego znaku z łańcucha. Czy wyrażenie `map(ord, S)` ma podobny efekt? (Wskazówka: patrz rozdział 14. książki).
- Znaki ukośników lewych. Co stanie się na naszym komputerze, kiedy w sesji interaktywnej wpiszemy poniższy kod?

```
for i in range(50):
    print('Witaj %d\n'a' % i)
```

Należy pamiętać, że poza interfejsem IDLE ten przykład może spowodować piski komputera, zatem wykonywanie go w załączonym pomieszczeniu niekoniecznie jest najlepszym pomysłem. IDLE, zamiast piszczeć, wyświetla dziwne znaki (zobacz znaki ucieczki z tabeli 7.2).

- Sortowanie słowników. W rozdziale 8. widzieliśmy, że słowniki są nieuporządkowanymi kolekcjami. Należy napisać pętlę `for` wyświetlającą elementy słownika w posortowanym (rosnącym) porządku. Wskazówka: należy użyć metod słowników `keys` i `list sort` lub nowszej funkcji wbudowanej `sorted`.
- Alternatywne logiki programu. Należy rozważyć poniższy kod, wykorzystujący pętlę `while` oraz opcję statusu (flagę) `found` do wyszukiwania w liście potęg liczby 2 wartości 2 podniesionej do piętej potęgi (32). Kod ten zapisany jest w pliku modułu o nazwie `power.py`.

```
L = [1, 2, 4, 8, 16, 32, 64]
X = 5

found = False
i = 0
while not found and i < len(L):
    if 2 ** X == L[i]:
        found = True
    else:
```

```
i = i+1  
if found:  
    print('pod indeksem', i)  
else:  
    print(X, 'nie odnaleziono')  
  
C:\book\tests> python power.py  
pod indeksem 5
```

W takiej postaci przykład ten nie wykorzystuje normalnych technik zapisu kodu w Pythonie. Należy postępować zgodnie z poniższymi krokami w celu poprawienia kodu. W przypadku wszystkich transformacji można albo wpisać kod interaktywnie, albo zachować go w pliku wykonywanym z systemowego wiersza poleceń — skorzystanie z pliku bardzo ułatwia zadanie.

- Najpierw należy przepisać ten kod z użyciem części else pętli while, która pozwoli wyeliminować opcję statusu found oraz ostatnią instrukcję if.
- Później należy przepisać ten przykład w taki sposób, by wykorzystywał on pętlę for z częścią else, co pozwoli wyeliminować jawną logikę indeksującą listę. Wskazówka: by otrzymać indeks elementu, należy skorzystać z metody listy index (`L.index(X)` zwraca wartość przesunięcia pierwszego X z listy L).
- Następnie należy całkowicie usunąć pętlę for, przepisując przykład za pomocą prostego wyrażenia z operatorem przynależności in. Więcej informacji na ten temat można znaleźć w rozdziale 8.; można również przetestować to rozwiązanie, wpisując do sesji interaktywnej `2 in [1, 2, 3]`.
- Na koniec należy użyć pętli for oraz metody listy append w celu wygenerowania listy potęg liczby 2 o nazwie L w miejsce użycia zapisanego na stałe w kodzie literała listy.

Kilka pogłębionych zagadnień:

- Czy użycie wyrażenia `2 ** X` poza pętlami poprawiłoby wydajność kodu? W jaki sposób można by zapisać takie rozwiązanie w kodzie?
- Jak widzieliśmy ćwiczeniu 1., Python zawiera narzędzie `map(funkcja, lista)`, które również może wygenerować listę potęg liczby 2 w następujący sposób: `map(lambda x: 2 ** x, range(7))`. Można spróbować wpisać ten kod w sesji interaktywnej. Z instrukcją lambda spotkamy się w rozdziale 19.

CZĘŚĆ IV

Funkcje

Podstawy funkcji

W trzeciej części książki zapoznaliśmy się z instrukcjami proceduralnymi w Pythonie. Teraz przejdziemy do omówienia zbioru instrukcji dodatkowych, których można użyć w celu utworzenia własnych funkcji.

Mówiąc w uproszczeniu, *funkcja* jest narzędziem grupującym zbiór instrukcji w taki sposób, by mogły one być wykonane w programie więcej niż jeden raz. Funkcje obliczają również wartość wyniku i pozwalają nam określić parametry służące za dane wejściowe — mogą się one zatem zmieniać z każdym wykonaniem kodu. Zapisanie operacji w postaci funkcji sprawia, że staje się ona przydatnym narzędziem, z którego można korzystać w wielu różnych kontekstach.

Co jednak ważniejsze, funkcje są alternatywą dla programowania polegającego na kopiowaniu i wklejaniu. Zamiast tworzyć kilka zbędnych kopii kodu operacji, możemy wykorzystać jedną funkcję. W taki sposób zmniejszamy również nakład pracy, jaki trzeba by było poświęcić w przyszłości, gdyby operacja musiała zostać zmodyfikowana. Dzięki funkcji kod należy uaktualnić tylko w jednym miejscu, a nie w kilku.

Funkcje są najważniejszą strukturą programu w Pythonie, służącą do maksymalizowania możliwości ponownego wykorzystania kodu i minimalizowania powtarzalności kodu. Jak zobaczymy niżej, funkcje są również narzędziem projektowym pozwalającym na rozbicie złożonych systemów na części, którymi łatwiej jest zarządzać. W tabeli 16.1 przedstawiono najważniejsze narzędzia związane z funkcjami, które omówimy w tej części książki.

Tabela 16.1. Instrukcje oraz wyrażenia powiązane z funkcjami

Instrukcja	Przykłady
Wywołania	<code>myfunc('mielonka', 'jajka', meat=ham)</code>
<code>def, return</code>	<code>def adder(a, b=1, *c): return a + b + c[0]</code>
<code>global</code>	<code>def changer(): global x; x = 'nowy'</code>
<code>nonlocal</code>	<code>def changer(): nonlocal x; x = 'nowy'</code>
<code>yield</code>	<code>def squares(x): for i in range(x): yield i ** 2</code>
<code>lambda</code>	<code>funcs = [lambda x: x**2, lambda x: x*3]</code>

Po co używa się funkcji?

Zanim przejdziemy do szczegółów, warto zarysować pełny obraz tego, czym tak naprawdę są funkcje. Funkcje są prawie uniwersalnym narzędziem nadającym strukturę programowi. Można je spotkać w innych językach programowania, w których czasami nazywane są *procedurami* czy *podprocedurami*. Słowem wstępnie, funkcje pełnią dwie główne role.

Maksymalizacja ponownego wykorzystania kodu i minimalizacja jego powtarzalności

Tak jak w większości języków programowania, funkcje Pythona są najprostszym sposobem spakowania razem logiki, z której możemy chcieć korzystać w więcej niż jednym miejscu i częściej niż raz. Dotychczas cały wpisywany kod wykonywany był natychmiast. Funkcje pozwalają na grupowanie i uogólnienie kodu, tak by mógł on później być wykorzystany w dowolny sposób i wiele razy. Ponieważ pozwalają na zapisanie operacji w jednym miejscu i wykorzystanie jej w wielu miejscach, funkcje Pythona są podstawowym narzędziem *faktoryzacji* dostępnym w tym języku. Pozwalają na zredukowanie powtarzalności kodu w programach i tym samym zmniejszają wysiłek związany z utrzymaniem kodu w przyszłości.

Proceduralne podzielenie na części

Funkcje umożliwiają również dzielenie systemów na fragmenty z jasno zdefiniowanymi rolami. By na przykład zrobić pizzę od podstaw, zaczynamy od zagniecenia ciasta, nałożenia go na blaszę, ułożenia dodatków, pieczenia i tak dalej. Gdybyśmy programowali robota robiącego pizzę, funkcje pomogłyby nam podzielić całe zadanie „zrób pizzę” na kilka części — po jednej funkcji dla każdego podzadania tego procesu. Łatwiej jest implementować mniejsze zadania w izolacji niż cały proces naraz. Funkcje wiążą się z *procedurami* — sposobami zrobienia czegoś, a nie celem tego działania. W szóstej części książki, gdy zaczniemy tworzyć nowe obiekty za pomocą klas, zobaczymy, dlaczego to rozróżnienie ma takie znaczenie.

W tej części książki omówimy narzędzia wykorzystywane do tworzenia funkcji w Pythonie — podstawy funkcji, reguły dotyczące zakresu, przekazywanie argumentów, a także kilka powiązanych z nimi zagadnień, takich jak generatory oraz narzędzia funkcyjne. Ponieważ na tym poziomie wiedzy bardzo duże znaczenie ma polimorfizm, powrócimy do tej wprowadzonej na początku książki koncepcji. Jak zobaczymy niewątem, funkcje nie wnoszą wiele nowej składni, jednak prowadzą nas do większych koncepcji programistycznych.

Tworzenie funkcji

Choć raczej w sposób nieoficjalny, używaliśmy już funkcji we wcześniejszych rozdziałach. By na przykład utworzyć obiekt pliku, wywoływałyśmy wbudowaną funkcję `open`. W podobny sposób użyliśmy funkcji wbudowanej `len` do obliczenia liczby elementów obiektu kolekcji.

W niniejszym rozdziale dowiemy się, jak w Pythonie pisze się *nowe* funkcje. Funkcje pisane przez nas zachowują się w ten sam sposób jak prezentowane już funkcje wbudowane — są wywoływane w wyrażeniach, przekazuje się do nich wartości i zwraca się z nich wyniki. Pisanie nowych funkcji wymaga jednak zastosowania kilku nowych koncepcji, o których jeszcze nie wspominaliśmy. Co więcej, w Pythonie funkcje zachowują się bardzo różnie od funkcji z języków kompilowanych, takich jak C. Poniżej znajduje się krótkie wprowadzenie do najważniejszych koncepcji dotyczących funkcji w Pythonie — wszystkie te kwestie omówimy w tej części książki.

- **def to kod wykonywalny.** Funkcje w Pythonie zapisywane są za pomocą nowej instrukcji `def`. W przeciwnieństwie do funkcji z języków kompilowanych (takich, jak C) `def` jest instrukcją wykonywalną — funkcja w Pythonie nie istnieje, dopóki Python nie dotrze do jej kodu i nie wykona instrukcji `def`. Tak naprawdę poprawne (i nawet czasami użyteczne) jest zagnieźdzanie instrukcji `def` wewnętrz instrukcji `if`, pętli `while`, a nawet innych instrukcji `def`. W typowej operacji instrukcje `def` zapisywane są w plikach modułów i wykonywane w celu wygenerowania funkcji, kiedy plik modułu zostaje zimportowany po raz pierwszy.
- **Instrukcja def tworzy obiekt i przypisuje go do nazwy.** Kiedy Python dotrze do instrukcji `def` i ją wykona, generowany jest nowy obiekt funkcji, który zostaje przypisany do nazwy funkcji. Tak jak w przypadku wszystkich operacji przypisania, nazwa funkcji staje się referencją do obiektu funkcji. W nazwie funkcji nie ma nic magicznego — jak zobaczymy, obiekt funkcji można przypisać do innych nazw czy przechować w liście. Do obiektów funkcji można także dodać dowolne zdefiniowane przez użytkownika *atrybuty* w celu przechowania danych.
- **Wyrażenie lambda tworzy obiekt i zwraca go jako wynik.** Funkcje można również tworzyć za pomocą wyrażenia `lambda` — opcji pozwalającej na wykorzystywanie definicji funkcji wewnętrz wiersza w miejscach, w których składnia instrukcji `def` nie będzie działała (jest to bardziej skomplikowana koncepcja omówiona w rozdziale 19.).
- **Instrukcja return przesyła wynikowy obiekt z powrotem do wywołującego.** Kiedy wywołuje się funkcję, kod wywołujący zatrzymuje się do czasu, aż funkcja zakończy pracę i zwróci do niego sterowanie. Funkcje obliczające wartość przesyłają ją z powrotem do wywołującego za pomocą instrukcji `return`, a zwierana wartość staje się wynikiem wywołania funkcji.
- **Instrukcja yield odsyła wynikowy obiekt z powrotem do wywołującego, jednak zapamiętuje, gdzie zakończyła działanie.** Funkcje znane jako *generatory* mogą również wykorzystywać instrukcję `yield` do odsyłania wartości i zawieszenia stanu w taki sposób, by można je było kontynuować później w celu otrzymania serii wyników rozłożonych w czasie. To kolejne zaawansowane zagadnienie, które zostanie omówione później w tej części książki.
- **Instrukcja global deklaruje zmienne, które mają być przypisane, na poziomie modułu.** Domyślnie wszystkie nazwy przypisane w funkcji są lokalne dla tej funkcji i istnieją tylko na czas jej wykonywania. By przypisać jakąś nazwę w module otaczającym funkcję, nazwa ta musi zostać wymieniona w instrukcji `global`. Nazw zawsze szuka się w *zakresach* — miejscach, w których przechowywane są zmienne; przypisanie wiąże nazwę z zakresem.
- **Instrukcja nonlocal deklaruje zmienne z funkcji zawierającej, które mają być przypisane.** Podobnie jak instrukcja `global`, dodana w Pythonie 3.0 instrukcja `nonlocal` pozwala funkcji na przypisanie zmiennej istniejącej w zakresie instrukcji `def` zawierającej ją z punktu widzenia składni. Pozwala to na wykorzystywanie funkcji zawierających do przechowywania *stanu* — informacji pamiętanych, kiedy funkcja jest wywoływana — bez konieczności korzystania ze współdzielonych zmiennych globalnych.
- **Argumenty przekazywane są przez przypisanie (referencję obiektu).** W Pythonie argumenty przekazywane są do funkcji za pomocą przypisania (co, jak już wiemy, oznacza referencję do obiektu). Jak zobaczymy niebawem, w modelu Pythona kod wywołujący i funkcja współdzielą obiekty przez referencje, jednak nie ma tutaj aliasów nazw. Zmiana nazwy argumentu wewnętrz funkcji nie zmienia odpowiadającej jej nazwy w kodzie

wywołującym. Modyfikacja przekazanych obiektów zmiennych może jednak zmienić obiekty współdzielone przez kod wywołujący.

- **Argumenty, zwracane wartości i zmienne nie są deklarowane.** Tak jak w każdym innym przypadku w Pythonie, w funkcjach nie ma ograniczenia w zakresie typów. Niczego dotyczącą funkcji nie trzeba tak naprawdę wcześniej deklarować — można zatem przekazać argumenty dowolnego typu czy zwrócić dowolny rodzaj obiektu. W rezultacie jedną funkcję można zastosować do różnych typów obiektów — każdy obiekt zawierający zgodny *interfejs* (metody i wyrażenia) będzie dobry, bez względu na typ.

Jeśli cokolwiek z powyższego opisu nie jest zrozumiałe — nie ma się co martwić. Wszystkie te koncepcje zostaną zilustrowane w tej części książki rzeczywistym kodem. Zacznijmy od rozszerzenia niektórych z powyższych zagadnień i przyjrzenia się kilku przykładom.

Instrukcje def

Instrukcja `def` tworzy obiekt funkcji i przypisuje go do nazwy. Jej ogólny format jest następujący:

```
def <nazwa>(arg1, arg2, ... argN):  
    <instrukcje>
```

Tak jak w przypadku wszystkich instrukcji złożonych Pythona, `def` składa się z wiersza nagłówka, po którym następuje zazwyczaj wcięty blok instrukcji (lub prosta instrukcja umieszczona po dwukropku). Blok instrukcji staje się *ciałem* funkcji, czyli kodem wykonywanym przez Pythona, za każdym razem, gdy funkcja jest wywoływana.

Wiersz nagłówka z `def` określa *nazwę* funkcji przypisywaną do obiektu funkcji, a także zero lub większą liczbę *argumentów* (czasami nazywanych *parametrami*) znajdujących się w nawiasach. Nazwy argumentów w nagłówku przypisywane są do obiektów przekazanych w nawiasach w momencie wywołania funkcji.

Ciało funkcji często zawiera instrukcję `return`.

```
def <nazwa>(arg1, arg2, ... argN):  
    ...  
    return <wartość>
```

Instrukcja `return` Pythona może pojawić się w dowolnym miejscu ciała funkcji. Kończy ona wywołanie funkcji i odsyła wyniki z powrotem do wywołującego. Instrukcja `return` składa się z wyrażenia obiektu podającego wynik funkcji. Jest ona opcjonalna — jeśli nie występuje, funkcja kończy się, kiedy sterowanie wychodzi poza jej ciało. Z technicznego punktu widzenia funkcja bez instrukcji `return` automatycznie zwraca obiekt `None`, jednak ta zwracana wartość jest zazwyczaj ignorowana.

Funkcje mogą również zawierać instrukcję `yield`, zaprojektowane w taki sposób, by z czasem tworzyły serię wartości, jednak ich omówienie odłożymy do czasu przedstawienia w rozdziale 20. zagadnień związanych z generatorami.

Instrukcja def uruchamiana jest w czasie wykonania

Instrukcja `def` Pythona jest prawdziwą instrukcją wykonywalną — kiedy jest wykonywana, tworzy nowy obiekt funkcji i przypisuje go do nazwy (należy pamiętać, że w Pythonie istnieje tylko *czas wykonywania* — nie ma osobnego czasu kompilacji). Ponieważ jest instrukcją, może

pojawiać się w każdym miejscu, w jakim występują instrukcje — nawet zagnieżdżona w innych instrukcjach. Choć instrukcje `def` są zazwyczaj wykonywane, kiedy importowany jest zawierający je moduł, można je również umieścić w instrukcji `if` pozwalającej na wybór alternatywnej definicji funkcji.

```
if test:  
    def func():  
        ...  
else:  
    def func():  
        ...  
...  
func()  
# Zdefiniowanie funkcji w taki sposób  
# Albo w taki sposób  
# Wywołanie wybranej i zbudowanej wersji
```

Jedną z metod zrozumienia tego kodu jest uświadomienie sobie, że `def` przypomina instrukcję `=`, ponieważ po prostu przypisuje nazwę w czasie wykonywania. W przeciwieństwie do języków kompilowanych, takich jak C, funkcje Pythona nie muszą być w pełni zdefiniowane przed wykonaniem programu. Instrukcje `def` nie są analizowane, dopóki nie zostaną wykonyane, a kod *wewnątrz* `def` nie zostanie obliczony aż do momentu, w którym funkcja ta zostanie później wywołana.

Ponieważ definicja funkcji odbywa się w czasie wykonywania, w nazwie funkcji nie ma nic specjalnego. Ważny jest obiekt, do którego nazwa ta się odnosi.

```
othername = func  
othername()  
# Przypisanie obiektu funkcji  
# Ponowne wywołanie funkcji func
```

W powyższym kodzie funkcja `func` została przypisana do innej nazwy i wywołana już za jej pomocą. Tak jak wszystko inne w Pythonie, funkcje są po prostu *obiekty*. Są zapisywane w jawnym sposób w pamięci w momencie wykonywania programu. Tak naprawdę poza wywoływaniem funkcje pozwalają na dołączanie dowolnych *atrybutów* w celu zapisania informacji i późniejszego ich wykorzystania:

```
def func(): ...  
func()  
func.attr = value  
# Utworzenie obiektu funkcji  
# Wywołanie obiektu  
# Dołączenie atrybutów
```

Pierwszy przykład — definicje i wywoływanie

Poza powyższymi kwestiami związanymi z czasem wykonywania (które dla programistów pracujących z tradycyjnymi językami kompilowanymi wydają się dość wyjątkowe) funkcje Pythona są stosunkowo proste w użyciu. Przejdzmy zatem do pierwszego prawdziwego przykładu, który pozwoli nam zademonstrować podstawy. Jak wynika z powyższego opisu, na funkcję składają się dwa elementy — *definicja* (instrukcja `def` tworząca funkcję) oraz *wywołanie* (wyrażenie nakazujące Pythonowi wykonanie ciała funkcji).

Definicja

Poniżej znajduje się definicja wpisana w sesji interaktywnej, definiująca funkcję o nazwie `times`, która zwraca iloczyn dwóch argumentów.

```
>>> def times(x, y):  
...     return x * y  
...  
# Utworzenie i przypisanie funkcji  
# Ciało funkcji wykonywane po wywołaniu
```

Kiedy Python trafi w kodzie na `def` i wykona tę instrukcję, utworzy nowy obiekt funkcji zawierający jej kod i przypisze ten obiekt do nazwy `times`. Zazwyczaj taka instrukcja znajduje się w pliku modułu i wykonywana jest, kiedy importowany jest zawierający ją plik. W przypadku tak niewielkiej funkcji sesja interaktywna będzie jednak w zupełności wystarczająca.

Wywołanie

Po wykonaniu instrukcji `def` możemy wywołać (wykonać) funkcję w programie, dodając nawiasy po jej nazwie. Nawiasy mogą opcjonalnie zawierać jeden lub większą liczbę argumentów, które mają być przekazane (przypisane) do zmiennych z nagłówka funkcji.

```
>>> times(2, 4)                                # Argumenty w nawiasach
8
```

Powyższe wyrażenie przekazuje do funkcji `times` dwa argumenty. Jak wspomniano wcześniej, argumenty przekazywane są przez przypisanie, dlatego w tym przypadku zmienna `x` z nagłówka funkcji przypisywana jest do wartości 2, natomiast zmienna `y` przypisywana jest do wartości 4. Później wykonywane jest ciało funkcji. W tej funkcji ciało składa się z jednej instrukcji `return` odsyłającej wynik w postaci wartości wyrażenia wywołującego. Zwracany obiekt został tutaj wyświetlony w sesji interaktywnej (tak, jak w większości języków, iloczyn `2 * 4` wynosi w Pythonie 8), jednak gdybyśmy potrzebowali użyć go później, moglibyśmy go przypisać do zmiennej — jak w poniższym kodzie.

```
>>> x = times(3.14, 4)                          # Zapisanie obiektu wyniku
>>> x
12.56
```

Sprawdźmy teraz, co się stanie, kiedy funkcja zostanie wywołana trzeci raz, z zupełnie innymi obiektami przekazanymi w charakterze argumentów.

```
>>> times('Ni', 4)                             # Dla funkcji typ nie ma znaczenia
'NiNiNiNi'
```

Tym razem nasza funkcja oznacza coś zupełnie innego (i zawiera kolejne odniesienie do Monty Pythona). W trzecim wywołaniu do zmiennych `x` oraz `y` w miejsce dwóch liczb przekazywane są łańcuch znaków i liczba całkowita. Jak wiemy, operator `*` działa zarówno na liczbach, jak i sekwencjach. Ponieważ w Pythonie nigdy nie deklarujemy typów zmiennych, argumentów ani zwracanych wartości, możemy wykorzystać funkcję `times` zarówno do *mnożenia* liczb, jak i *powtarzania* sekwencji.

Innymi słowy, to, co oznacza i robi nasza funkcja, zależy od tego, co do niej przekażemy. Jest to w Pythonie kwestia kluczowa (i tym samym chyba klucz do wykorzystywania całego języka), którą omówimy w kolejnym podrozdziale.

Polimorfizm w Pythonie

Jak widzieliśmy przed momentem, znaczenie wyrażenia `x * y` w naszej prostej funkcji `times` jest w całości uzależnione od typów obiektów, jakimi są `x` oraz `y`. Tym samym jedna funkcja może w jednym przypadku wykonywać mnożenie, a w drugim powtarzanie. Python pozostawia obiektem zrobienie czegoś, co ma sens w przypadku podanej składni. Tak naprawdę znak `*` jest po prostu mechanizmem przekazującym kontrolę do przetwarzanych obiektów.

Taki rodzaj zachowania uzależnionego od typów znany jest pod nazwą *polimorfizmu* — konsepcji, z którą pierwszy raz spotkaliśmy się w rozdziale 4. i która oznacza, że operacja uzależniona jest od obiektów, na jakich jest wykonywana. Ponieważ Python jest językiem o typach dynamicznych, szerzy się w nim polimorfizm. Tak naprawdę w Pythonie każda operacja jest polimorficzna — wyświetlanie, indeksowanie, operator * i wiele, wiele innych.

Jest to celowe i z takiego rozwiązania wynika duża część związkowości oraz elastyczności Pythona. Pojedyncza funkcja może na przykład zostać automatycznie zastosowana do szerokiej gamy typów obiektów. Dopóki obiekty te obsługują oczekiwany interfejs (inaczej mówiąc: protokół), funkcja może je przetworzyć. Oznacza to, że jeśli obiekty przekazane do funkcji mają oczekiwane metody oraz operatory wyrażeń, są z miejsca zgodne z logiką funkcji.

Nawet w naszej prostej funkcji `times` oznaczało to, że *dowolne* dwa obiekty obsługujące operator * będą działać — bez względu na to, czym są, i na to, kiedy zostaną zapisane w kodzie. Funkcja ta będzie działała na dwóch liczbach (wykonując mnożenie) lub łańcuchu znaków i liczbie (wykonując powtórzenie) bądź dowolnej kombinacji obiektów obsługujących oczekiwany interfejs — nawet obiektów opartych na klasach, których kod jeszcze nie powstał.

Co więcej, jeśli przekazane obiekty *nie* obsługują oczekiwanej interfejsu, Python wykryje błąd przy wykonywaniu wyrażenia z operatorem * i automatycznie zwróci wyjątek. Samodzielne sprawdzanie błędów w kodzie jest zatem bezcelowe. Tak naprawdę takie działanie obniżyłoby użyteczność funkcji, ponieważ byłaby ona ograniczona do obiektów, na których jej działanie zostało przetestowane.

Okazuje się to kluczową różnicą w filozofii Pythona i języków z typami statycznymi, takich jak C++ i Java. W Pythonie określone typy danych *mają nie mieć znaczenia* dla kodu. Jeśli tak nie jest, kod będzie ograniczony do działania wyłącznie na typach przewidzianych w momencie pisania go i nie będzie obsługiwał innych zgodnych typów obiektów, które mogą powstać w przyszłości. Choć można testować kod pod kątem działania na określonych typach za pomocą wbudowanych narzędzi, takich jak funkcja `type`, takie działanie niszczy elastyczność kodu. W Pythonie tworzy się kod z myślą o *interfejsach* obiektów, a nie typach danych.

Oczywiście polimorficzny model programowania oznacza, że musimy sprawdzać nasz kod w celu wykrycia błędów, zamiast udostępniać deklaracje typów, które kompilator może wykorzystać do wykrycia pewnych rodzajów błędów. W zamian za nieco więcej początkowego testowania możemy radykalnie zmniejszyć ilość kodu, jaką musimy napisać, a także radykalnie zwiększyć jego elastyczność. Jak się niebawem okaże, w praktyce oznacza to zysk netto.

Drugi przykład — przecinające się sekwencje

Przyjrzyjmy się kolejnemu przykładowi, który robi coś nieco bardziej przydatnego od mnożenia argumentów i dodatkowo ilustruje podstawy funkcji.

W rozdziale 13. utworzyliśmy pętlę `for` zbierającą elementy wspólne dla dwóch łańcuchów znaków. Zauważliśmy tam, że kod nie był tak użyteczny, jak mógłby być, ponieważ został skonfigurowany do działania jedynie na określonych zmiennych i nie mógł być wykonany ponownie później. Oczywiście moglibyśmy skopiować kod i wkleić go do każdego miejsca, w którym miałby on zostać wykonany, jednak takie rozwiązanie nie jest ani dobre, ani uniwersalne — nadal musielibyśmy dokonywać edycji każdej kopii kodu w celu obsłużenia różnych nazw sekwencji. Zmiana samego algorytmu wymagałaby wprowadzenia modyfikacji do każdej z kopii.

Definicja

W tej chwili każdy już zapewne zgadł, że rozwiążaniem tego problemu jest umieszczenie pętli `for` wewnętrz funkcji. Takie działanie ma kilka zalet.

- Umieszczenie kodu w funkcji sprawia, że uzyskujemy narzędzie, które można wykonać tyle razy, ile nam się będzie podobało.
- Ponieważ kod wywołujący może przekazywać argumenty dowolnego typu, funkcje są na tyle uniwersalne, by działały na dowolnych dwóch sekwencjach (czy innych obiektach, na których można wykonywać iterację), jakich część wspólną chcemy obliczyć.
- Kiedy logika umieszczana jest w funkcji, w momencie gdy chcemy zmienić jej sposób działania, wystarczy jedynie wprowadzić modyfikacje w jednym miejscu.
- Umieszczenie funkcji w pliku modułu oznacza, że można ją importować i użyć jej ponownie w dowolnym programie działającym na naszym komputerze.

W rezultacie opakowanie kodu w funkcję sprawia, że staje się ona wszechstronnym narzędziem do zwracania części wspólnych.

```
def intersect(seq1, seq2):  
    res = [] # Na początek pusta lista  
    for x in seq1:  
        if x in seq2:  
            res.append(x) # Przeszukanie pierwszej sekwencji  
                           # Powtarzający się element?  
                           # Dodanie na końcu listy wyników  
    return res
```

Transformacja z prostego kodu z rozdziału 13. na funkcję jest prosta — po prostu osadziliśmy oryginalny kod pod nagłówkiem `def`, a z obiektów, na których działa funkcja, zrobiliśmy przekazywane nazwy argumentów. Ponieważ funkcja ta oblicza wynik, dodaliśmy również instrukcję `return` przesyłającą ten wynik z powrotem do kodu wywołującego.

Wywołania

Zanim wywołamy funkcję, musimy ją utworzyć. By to zrobić, należy wykonać jej instrukcję `def` — albo w sesji interaktywnej, albo umieszczając ją w pliku modułu i importując ten plik. Po wykonaniu instrukcji `def` można wywołać funkcję, przekazując jej dowolne dwa obiekty sekwencji w nawiasach.

```
>>> s1 = "mielonka"  
>>> s2 = "biedronka"  
  
>>> intersect(s1, s2) # Łącuchy znaków  
['i', 'e', 'o', 'n', 'k', 'a']
```

W powyższym kodzie przekazaliśmy do funkcji dwa łańcuchy znaków i otrzymaliśmy z powrotem listę zawierającą znaki wspólne w obu łańcuchach. Algorytm wykorzystywany przez funkcję jest prosty: „W przypadku każdego elementu z pierwszego argumentu, jeśli element ten znajduje się również w drugim argumentie, dodaj ten element do wyniku”. W Pythonie jest to nieco krótsze niż w języku polskim, jednak działa tak samo.

Uczciwie należy przyznać, że nasza funkcja jest dość wolna (wykonuje zagnieżdżone pętle), nie zwraca tak naprawdę matematycznej części wspólnej (w wyniku mogą się pojawiać duplikaty), a na dodatek jest zupełnie niepotrzebna (widzieliśmy już, że typ danych zbioru w Pytho-

nie udostępnia wbudowaną operację obliczania części wspólnej). I faktycznie, funkcję tę można zastąpić prostym wyrażeniem listy składanej, gdyż działa ono zgodnie z klasycznym wzorcem pętli zbierającej elementy:

```
>>> [x for x in s1 if x in s2]
['i', 'e', 'o', 'n', 'k', 'a']
```

Kod ten spełnia jednak swoje zadanie prostego przykładu działania funkcji. Ten fragment kodu można zastosować do różnych typów obiektów, co zostanie wyjaśnione w kolejnym podrozdziale.

Raz jeszcze o polimorfizmie

Tak jak wszystkie pozostałe funkcje w Pythonie, funkcja `intersect` jest polimorficzna. Oznacza to, że działa na dowolnych typach, które obsługują oczekiwany interfejs obiektu.

```
>>> x = intersect([1, 2, 3], (1, 4))          # Typy mieszane
>>> x                                         # Zapisany obiekt wyniku
[1]
```

Tym razem do funkcji przekazaliśmy różne typy obiektów — listę oraz krotkę (typy mieszane); funkcja nadal była w stanie wybrać elementy wspólne. Ponieważ nie musimy określać z wyprzedzeniem typów argumentów, funkcja `intersect` z radością przejdzie dowolny typ sekwencji, o ile tylko obsługuje on oczekiwany interfejs.

Dla `intersect` oznacza to, że pierwszy argument musi obsługiwać pętlę `for`, a drugi test przynależności `in`. Funkcja będzie działała na dwóch dowolnych obiektach tego rodzaju bez względu na ich typ — obejmuje to zarówno sekwencje przechowywane fizycznie w pamięci, jak i łańcuchy znaków oraz listy, wszystkie typy z rozdziału 14., na których można wykonywać iterację, w tym pliki i słowniki, a nawet utworzone przez nas obiekty oparte na klasach wykorzystujące techniki przeciążania operatorów (omówimy je później, w dalszej części książki).¹

I znów, jeśli przekażemy obiekty nieobsługujące tych interfejsów (na przykład liczby), Python automatycznie wykryje niedopasowanie i zgłosi wyjątek — i dokładnie tego chcemy, ponieważ gdybyśmy tworzyli jawne testy typów, zrobilibyśmy to w taki sam sposób. Nie zapisując testów typu samodzielnie i pozwalając Pythonowi na wykrycie niedopasowania, redukujemy ilość kodu, jaki musimy zapisać, jednocześnie zwiększając jego elastyczność.

Zmienne lokalne

Najciekawszą częścią tego przykładu są chyba zmienne. Okazuje się, że zmienna `res` wewnętrz funkcji `intersect` jest czymś, co w Pythonie nazywane jest *zmienną lokalną* — nazwą widoczną jedynie dla kodu wewnętrz definicji funkcji i istniejącą jedynie w czasie wykonywania

¹ Kod ten będzie działał zawsze, jeśli będziemy sprawdzali część wspólną zawartości plików uzyskanych za pomocą funkcji `file.readlines()`. Może on jednak nie działać w przypadku części wspólnej plików wejściowych otwieranych w sposób bezpośredni, w zależności od implementacji operatora `in` w obiekcie pliku lub ogólnej iteracji. Po wczytaniu do końca pliku muszą generalnie być przewijane (na przykład za pomocą `file.seek(0)` lub kolejnego `open`). Jak zobaczymy w rozdziale 29, przy okazji omawiania przeciążania operatorów, klasy implementują operator `in` albo udostępniając określoną metodę `__contains__`, albo obsługując ogólny protokół iteracji za pomocą metody `__iter__` lub starszej `__getitem__`. Po zapisaniu w ten sposób klasy mogą definiować, co iteracja oznacza dla ich danych.

tej funkcji. Tak naprawdę, ponieważ wszystkie nazwy *przypisywane* w dowolny sposób wewnątrz funkcji są domyślnie klasyfikowane jako zmienne lokalne, prawie wszystkie nazwy z funkcji `intersect` są zmiennymi tego typu.

- Zmienna `res` jest przypisywana w jawny sposób, dlatego jest zmienną lokalną.
- Argumenty przekazywane są przez przypisanie, dlatego również `seq1` i `seq2` są zmiennymi lokalnymi.
- Pętla `for` przypisuje elementy do zmiennej, zatem nazwa `x` również jest lokalna.

Wszystkie zmienne lokalne pojawiają się w momencie wywołania funkcji i znikają, kiedy funkcja przestaje działać — instrukcja `return` na końcu funkcji `intersect` odsyła z powrotem obiekt wynikowy, jednak nazwa `res` znika. By w pełni zrozumieć pojęcie zmiennych lokalnych, trzeba jednak przejść do rozdziału 17.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadzono podstawowe koncepcje związane z definicją funkcji — składnię i działanie instrukcji `def` oraz `return`, a także zachowanie wyrażeń wywołujących funkcję — oraz koncepcję i zalety polimorfizmu w funkcjach Pythona. Jak widzieliśmy, instrukcja `def` jest kodem wykonywalnym tworzącym obiekt funkcji w czasie wykonania. Kiedy funkcję tę później się wywoła, obiekty przekazywane są do niej za pomocą przypisania (warto przypomnieć, że w Pythonie przypisanie oznacza referencję do obiektu, co wewnętrznie, jak wiadomo z rozdziału 6., oznacza wskaźnik do miejsca w pamięci); obliczone wartości odsyłane są z powrotem przez instrukcję `return`. Zaczęliśmy również zapoznawać się z koncepcją zmiennych lokalnych oraz zakresów, których szczegóły zostaną omówione w rozdziale 17. Najpierw jednak pora na quiz.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel tworzenia funkcji?
2. W którym momencie Python tworzy funkcję?
3. Co zwraca funkcja, jeśli nie ma w niej instrukcji `return`?
4. Kiedy wykonywany jest kod zagnieżdżony wewnątrz instrukcji definicji funkcji?
5. Co zlego jest w sprawdzaniu typów obiektów przekazywanych do funkcji?

Sprawdź swoją wiedzę — odpowiedzi

1. Funkcje są podstawowym sposobem unikania *powtarzalności* kodu w Pythonie. Faktoryzacja kodu w funkcje oznacza, że mamy tylko jedną kopię kodu operacji do uaktualnienia w przyszłości. Funkcje są również podstawową metodą *ponownego użycia kodu* w Pythonie — umieszczenie kodu w funkcjach powoduje, że można z niego korzystać później za-

pomocą wywołania w różnych programach. Funkcje pozwalają nam również na podzielenie złożonego systemu na części, którymi łatwiej jest zarządzać, a z których każda może być rozwijana niezależnie.

2. Funkcja jest tworzona, kiedy Python trafi na instrukcję `def` i ją wykona. Instrukcja ta tworzy obiekt funkcji i przypisuje go do nazwy funkcji. Normalnie dzieje się to, kiedy moduł ją zawierający importowany jest przez inny moduł (warto przypomnieć, że instrukcja `import` wykonuje kod pliku od góry do dołu, włączając w to wszystkie instrukcje `def`). Dzieje się tak jednak również wtedy, gdy instrukcja `def` wpisana zostanie w sesji interaktywnej lub zostaje zagnieźdzona w innych instrukcjach, takich jak `if`.
3. Funkcja domyślnie zwraca obiekt `None`, kiedy sterowanie wyjdzie poza jej ciało bez trafienia na instrukcję `return`. Takie funkcje zazwyczaj wywoływane są za pomocą wyrażenia wywołania. Gdyż przypisanie ich wyników (`None`) do zmiennych zwykłe jest bezcelowe.
4. Ciało funkcji (kod zagnieźdzony wewnętrz instrukcji definicji funkcji) wykonywane jest, kiedy funkcja jest później wywoływana za pomocą wyrażenia wywołania. Ciało funkcji z każdym wywołaniem funkcji wykonywane jest na nowo.
5. Sprawdzanie typów obiektów przekazywanych do funkcji w praktyce niszczy jej elastyczność, ograniczając funkcję do działania jedynie na określonych typach. Bez takiego sprawdzania funkcja będzie najprawdopodobniej mogła przetworzyć całą gamę typów obiektów — każdy obiekt obsługujący interfejs oczekiwany przez funkcję będzie działał. Pojęcie *interfejs* oznacza tutaj zbiór metod oraz operatorów wyrażeń wykonywanych przez kod funkcji.

Zakresy

W rozdziale 16. wprowadzono podstawowe definicje i wywołania funkcji. Jak widzieliśmy, podstawowy model funkcji Pythona jest prosty w użyciu, jednak nawet proste przykłady funkcji sprawiły, że zaczęliśmy sobie zadawać pytania dotyczące znaczenia zmiennych w kodzie. W niniejszym rozdziale przedstawimy szczegóły dotyczące zakresów Pythona — miejsc, w których zmienne są definiowane i wyszukiwane. Jak się przekonamy, miejsce, w którym zmienna jest przypisana w kodzie, ma kluczowe znaczenie w ustaleniu, co zmienna ta oznacza. Dowiemy się także, że korzystanie z zakresów może mieć duży wpływ na wysiłek związany z utrzymywaniem kodu. Nadużywanie zmiennych globalnych jest na przykład złym zwyczajem.

Podstawy zakresów w Pythonie

Skoro już zaczęliśmy tworzyć własne funkcje, musimy sprawdzić, co od strony formalnej oznaczają nazwy (zmienne) w Pythonie. Kiedy w programie użyjemy jakieś nazwy, Python tworzy, zmienia lub wyszukuje nazwę w czymś, co znane jest jako *przestrzeń nazw* (ang. *namespace*) — miejscu, w którym „żyją” nazwy. Kiedy mówimy o szukaniu wartości nazwy w odniesieniu do kodu, do przestrzeni nazw odnosi się *zakres* (ang. *scope*). Lokalizacja przypisania nazwy w kodzie określa zakres, w jakim nazwa ta jest w kodzie widoczna.

Prawie wszystko, co wiąże się z nazwami, w tym kwalifikacja do zakresów, odbywa się w Pythonie w czasie przypisania. Jak widzieliśmy, nazwy w Pythonie zaczynają istnieć, kiedy pierwszy raz przypisuje się do nich wartości, co musi się odbyć przed ich pierwszym użyciem. Ponieważ nazw nie deklaruje się z wyprzedzeniem, Python wykorzystuje lokalizację przypisania do nazwy do *powiązania* jej z określona przestrzenią nazw. Innymi słowy, miejsce, w którym przypisujemy nazwę w kodzie źródłowym, określa przestrzeń nazw, w której nazwa ta będzie istniała, a tym samym również zakres jej widoczności.

Poza łączeniem kodu w pakiety funkcje dodają do naszych programów dodatkową warstwę przestrzeni nazw. Domyślnie wszystkie nazwy przypisane wewnątrz funkcji wiązane są tylko z przestrzenią nazw tej funkcji i żadną inną. Oznacza to, że:

- Nazwy zdefiniowane wewnątrz instrukcji `def` mogą być widoczne jedynie dla kodu wewnątrz tej instrukcji. Nie można się do nich odnosić w żaden sposób poza tą funkcją.

- Nazwy zdefiniowane wewnątrz instrukcji `def` nie wchodzą w konflikt ze zmiennymi spoza tej instrukcji, nawet jeśli tak samo się nazywają. Zmienna `X` przypisana poza definicją funkcji (na przykład w innej definicji funkcji lub na najwyższym poziomie pliku modułu) jest czymś zupełnie innym od zmiennej `X` przypisanej w tej funkcji.

W każdym przypadku zakres zmiennej (miejsce, w którym może ona zostać użyta) jest zawsze ustalany przez to, w którym miejscu kodu źródłowego zostaje ona przypisana, i nie ma nic wspólnego z tym, która funkcja wywołuje którą zmienną. Jak zresztą dowiemy się z niniejszego rozdziału, zmienne można przypisywać w trzech różnych miejscach, odpowiadających trzem różnym zakresom:

- Jeśli zmienna zostanie przypisana wewnątrz instrukcji `def`, staje się zmienną *lokalną* dla tej funkcji.
- Jeśli zmienna przypisana jest wewnątrz instrukcji `def` zawierającej inną funkcję, staje się zmienną *nielokalną* dla tej funkcji.
- Jeśli zmienna przypisana jest poza wszystkimi instrukcjami `def`, staje się zmienną *globalną* dla całego pliku.

Nazywamy to *zakresem leksykalnym*, ponieważ zakres zmiennej uzależniony jest w całości od lokalizacji zmiennych w kodzie źródłowym plików programu, a nie od wywołań funkcji.

W poniższym pliku modułu przypisanie `X = 99` tworzy zmienną *globalną* o nazwie `X` (widoczną w całym pliku). Przypisanie `X = 88` natomiast zmienną *lokalną* `X` (widoczną jedynie wewnątrz instrukcji `def`).

```
X = 99  
def func():  
    X = 88
```

Pomimo iż obie zmienne noszą nazwę `X`, ich zakresy sprawiają, że są one czymś innym. W rezultacie zakres funkcji pozwala unikać konfliktów nazw w Pythonie i pomaga zamienić funkcje w bardziej samowystarczalne jednostki programu.

Reguły dotyczące zakresów

Zanim zaczeliśmy tworzyć funkcje, cały kod przez nas pisany był kodem najwyższego poziomu modułu (niezagieźdzonym wewnątrz instrukcji `def`), dlatego wykorzystywane przez nas nazwy albo były częścią modułu, albo były wbudowane w samego Pythona (jak na przykład `open`). Funkcje udostępniają zagnieźdzone przestrzenie nazw (zakresy), zawierające nazwy dla siebie lokalne, tak by zmienne te nie pozostawały w konflikcie z tymi spoza zakresu funkcji (znajdującymi się w module czy innej funkcji). Funkcje definiują *zakres lokalny*, natomiast moduły — *zakres globalny*. Te dwa zakresy wiążą się ze sobą w następujący sposób:

- **Moduł zawierający funkcję jest zakresem globalnym.** Każdy moduł jest zakresem globalnym, czyli przestrzenią nazw, w której znajdują się zmienne tworzone (przypisane) na najwyższym poziomie pliku modułu. Zmienne globalne stają się atrybutami obiektu modułu dla świata zewnętrznego, jednak wewnątrz samego modułu mogą być używane jako proste zmienne.
- **Zakres globalny rozciąga się jedynie na jeden plik.** Nie należy dać się wprowadzić w błąd słowem „globalny” — zmienne przypisane na najwyższym poziomie pliku są globalne jedynie w odniesieniu do kodu tego jednego pliku. W Pythonie nie istnieje coś takiego, jak

jeden obejmujący wszystko globalny zakres oparty na plikach. Zamiast tego zmienne dziedziczone są na moduły i jeśli chcemy z nich skorzystać, musimy moduł zawsze w jawnym sposobie zaimportować. Kiedy słyszymy „globalny” w odniesieniu do Pythona, naszym pierwszym skojarzeniem powinien być „moduł”.

- **Każde wywołanie funkcji tworzy nowy zakres lokalny.** Za każdym razem, gdy wywołujemy funkcję, tworzymy nowy zakres lokalny — to znaczy przestrzeń nazw, w której znajdują się zmienne utworzone w tej funkcji. Każdą instrukcję `def` (i każde wyrażenie `lambda`) można sobie wyobrazić jako tworzącą nowy zakres lokalny, jednak ponieważ Python pozwala na wywoływanie funkcji przez nie same w celu wykonania pętli (jest to zaawansowana technika nazywana *rekurencją*), zakres lokalny tak naprawdę odpowiada wywołaniu funkcji. Krótko mówiąc, każde wywołanie tworzy nową lokalną przestrzeń nazw. Rekurencja przydaje się przy przetwarzaniu struktur, których kształtu nie możemy przewidzieć z wyprzedzeniem.
- **Przypisane nazwy są lokalne, o ile nie zostaną zadeklarowane jako globalne lub nielokalne.** Domyślnie wszystkie nazwy przypisane wewnętrz definicji funkcji umieszczane są w zakresie lokalnym (przestrzeni nazw powiązanej z wywołaniem funkcji). Jeśli potrzebujemy przypisać zmienną istniejącą na najwyższym poziomie modułu zawierającego funkcję, możemy to zrobić, deklarując to w instrukcji `global` wewnętrz funkcji. Jeśli potrzebujemy przypisać zmienną istniejącą w instrukcji `def` zawierającej inną funkcję, od Pythona 3.0 możemy to zrobić, deklarując to w instrukcji `nonlocal`.
- **Wszystkie pozostałe nazwy są lokalne dla zakresu zawierającego, globalne lub wbudowane.** Nazwy, które nie zostały przypisane wewnętrz definicji funkcji, są lokalne dla zawierającego je zakresu (w przypadku instrukcji `def` zawierającej tę funkcję), globalne (należące do przestrzeni nazw modułu) lub wbudowane (znajdujące się w udostępnianym przez Pythona module `builtins`).

Należy tu zwrócić uwagę na kilka subtelności. Po pierwsze, trzeba pamiętać, że kod wpisany w *interaktywnym wierszu poleceń* stosuje się do tych samych reguł. Może to jeszcze nie być oczywiste, ale kod wykonywany interaktywnie w wierszu poleceń jest tak naprawdę wprowadzany do wbudowanego modułu o nazwie `_main_`. Moduł ten działa tak samo jak zwykły plik modułu, jednak dane wyjściowe są wyświetlane od razu w miarę przechodzenia dalej w kodzie. Z tego powodu zmienne utworzone w sesji interaktywnej również są częścią modułu i podlegają normalnym regułom dotyczącym zakresów — są one globalne dla sesji interaktywnej. Więcej informacji na temat modułów znajdziesz się w kolejnej części książki.

Warto także zauważyć, że *dowolny typ przypisania* wewnętrz funkcji klasyfikuje zmienną jako lokalną. Dotyczy to między innymi instrukcji ze znakiem `=`, zmiennych modułów w instrukcji `import`, zmiennych funkcji w instrukcji `def`, a także przekazywania argumentów. Jeśli przypiszemy zmienną w jakikolwiek sposób wewnętrz instrukcji `def`, stanie się ona lokalna dla tej funkcji.

Należy również zwrócić uwagę na to, że *modyfikacje obiektów w miejscu* nie klasyfikują zmiennych jako lokalnych — jest tak jedynie w przypadku właściwego przypisywania zmiennej. Jeśli na przykład nazwa `L` zostanie przypisana do listy na najwyższym poziomie modułu, instrukcja taka jak `L.append(X)` wewnętrz funkcji nie zaklasyfikuje `L` jako zmiennej lokalnej, natomiast `L = X` już tak. W tym pierwszym przypadku zmieniamy obiekt listy, do którego odwołuje się zmienna `L`, a nie samą zmienną `L`. `L` zostanie jak zwykle odnaleziona w zakresie globalnym, a Python z radością ją zmodyfikuje bez wymagania deklaracji `global` lub `nonlocal`. Jak zwykle należy jasno rozróżnić nazwy (zmienne) i obiekty — modyfikacja obiektu nie jest przypisaniem go do zmiennej.

Rozwiązywanie konfliktów w zakresie nazw — reguła LEGB

Jeśli omówione wcześniej kwestie nie są wystarczająco jasne, warto dodać, że sprowadzają się one do trzech reguł. W instrukcji `def`:

- Referencje do nazw przeszukują cztery zakresy — lokalny, następnie funkcji zawierających tę funkcję (jeśli takie istnieją), globalny i na końcu wbudowany.
- Przypisania nazw domyślnie tworzą lub modyfikują nazwy lokalne.
- Deklaracje `global` i `nonlocal` odwzorowują przypisane nazwy na zakres modułu zawierającego oraz funkcji zawierającej.

Innymi słowy, wszystkie nazwy przypisywane wewnątrz instrukcji `def` funkcji (lub wyrażenia `lambda`, z którym spotkamy się nieco później) są domyślnie lokalne. Funkcje mogą swobodnie wykorzystywać nazwy przypisane w funkcjach je zawierających, a także z zakresu globalnego, jednak by je zmodyfikować, muszą deklarować zmienne nielokalne oraz globalne.

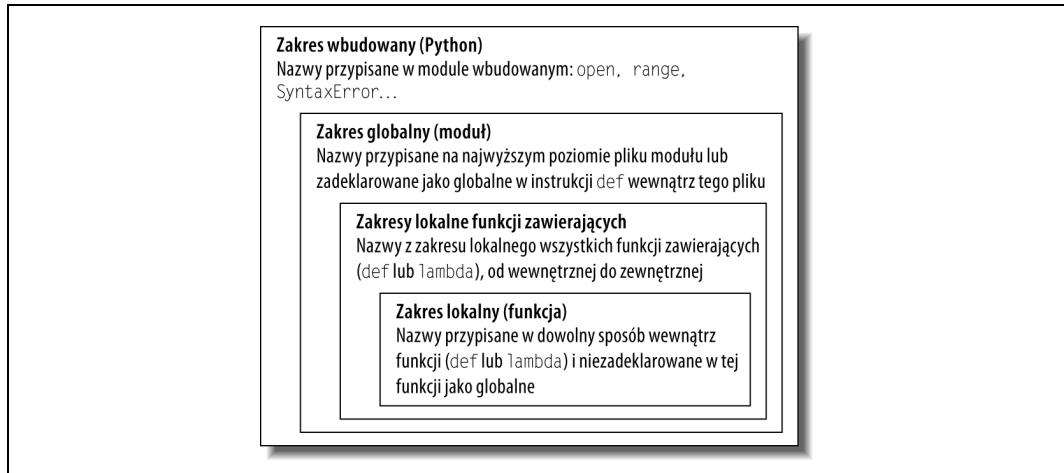
Rozwiązywanie konfliktów w zakresie nazw w Pythonie czasami nazywane jest *regułą LEGB* — od angielskich nazw kolejnych zakresów:

- Kiedy wewnątrz funkcji użyjemy nazwy bez kwalifikatora, Python przeszuka cztery zakresy — lokalny (*L*, od ang. *local*), następnie zakres lokalny instrukcji `def` lub wyrażenia `lambda` zawierających daną funkcję (*E*, od ang. *enclosing*), później globalny (*G*, od ang. *global*), a na końcu wbudowany (*B*, od ang. *built-in*), zatrzymując się w pierwszym miejscu, w którym nazwa ta zostanie odnaleziona. Jeśli nazwa nie zostanie znaleziona, Python zgłasza błąd. Jak wiemy z rozdziału 6., nazwy muszą być przypisane przed pierwszym użyciem.
- Kiedy przypisujemy zmienną wewnątrz funkcji (zamiast odnieść się do niej w wyrażeniu), Python zawsze tworzy lub modyfikuje tę zmienną w zakresie lokalnym, o ile nie została ona w tej funkcji zadeklarowana jako globalna lub nielokalna.
- Kiedy przypisujemy zmienną poza jakąkolwiek funkcją (na przykład na najwyższym poziomie pliku modułu czy w sesji interaktywnej), zakres lokalny jest tym samym co zakres globalny — przestrzenią nazw modułu.

Na rysunku 17.1 przedstawiono cztery zakresy Pythona. Warto zauważyć, że druga warstwa przeszukiwania — zakres *E* (czyli zakres instrukcji `def` lub wyrażeń `lambda` funkcji zawierającej kolejną funkcję) — może z technicznego punktu widzenia odpowiadać większej liczbie warstw. Taka sytuacja pojawia się tylko wtedy, gdy zagnieżdżamy funkcje w innych funkcjach, i zajmuje się nią instrukcja `nonlocal`.¹

Należy również pamiętać, że reguły te mają zastosowanie jedynie do prostych nazw *zmiennych* (takich jak `spam`). W piątej i szóstej części książki zobaczymy, że kwalifikowane nazwy *atrybutów* (takie jak `object.spam`) istnieją w poszczególnych obiektach i obowiązują je zupełnie

¹ Reguła ta w pierwszym wydaniu książki nosiła nazwę „reguły LGB”. Warstwa zawierającej instrukcję `def` została w Pythonie dodana później, by zlikwidować konieczność jawnego przekazywania nazw za pomocą argumentów domyślnych w zakresie zawierającym — zagadnienie to jest na tyle mało znaczące dla osób początkujących, że odłożymy jego omówienie do dalszej części rozdziału. Ponieważ problem ten w Pythonie 3.0 został rozwiązany za pomocą instrukcji `nonlocal`, reguła LEGB mogłaby teraz zostać przemianowana na LNGB, ale na szczęście zachowanie zgodności z poprzednimi wersjami ma znaczenie także w książkach!



Rysunek 17.1. Reguła przeszukiwania zakresów LEGB. Kiedy w kodzie pojawia się referencja do zmiennej, Python szuka tej zmiennej w następującej kolejności: zakres lokalny, zakres lokalny funkcji zawierających tę funkcję, zakres globalny i na końcu zakres wbudowany. Wygrywa pierwsze wystąpienie. Zakres zazwyczaj uzależniony jest od miejsca przypisania zmiennej w kodzie. W Pythonie 3.X deklaracje nonlocal mogą także wymusić odwzorowanie zmiennych na zakresy funkcji zawierającej, bez względu na to, czy są one przypisane inne reguły wyszukiwania niż omówione tutaj koncepcje związane z zakresami. Referencje do atrybutów (nazw następujących po kropce) przeszukują obiekt lub obiekty, a nie zakresy, i mogą wywołać coś o nazwie dziedziczenia (omówionego w szóstej części książki).

Przykład zakresu

Przyjrzyjmy się większemu przykładowi demonstrującemu koncepcje związane z zakresami. Założmy, że w pliku modułu zapiszemy poniższy kod:

```
# Zakres globalny
X = 99 # X i func przypisane w module: globalne

def func(Y):
    # Zakres lokalny
    Z = X + Y # Y i Z przypisane w funkcji: lokalne
    return Z # X jest globalne

func(1) # func w module: wynik = 100
```

Moduł i umieszczona w nim funkcja wykorzystują kilka zmiennych. Za pomocą reguł zakresu Pythona możemy zmienne sklasyfikować w następujący sposób:

Nazwy globalne — X, func

Zmienna X jest globalna, ponieważ zostaje przypisana na najwyższym poziomie pliku modułu. Można się do niej odnieść ze środka funkcji bez deklarowania jej jako globalnej. Nazwa func jest globalna z tego samego powodu; instrukcja def przypisuje obiekt funkcji do nazwy func na najwyższym poziomie modułu.

Nazwy lokalne — Y, Z

Zmienne Y oraz Z są lokalne dla funkcji (i istnieją tylko w czasie jej działania), ponieważ do obu przy definiowaniu funkcji przypisaliśmy wartości — do Z za pomocą instrukcji =, natomiast do Y dlatego, że argumenty zawsze przekazywane są przez przypisanie.

Celem takiego schematu rozdzielenia nazw zmiennych jest to, że zmienne lokalne służą jako tymczasowe nazwy potrzebne tylko w czasie wykonywania funkcji. W powyższym przykładzie argument `Y` oraz wynik dodawania `Z` istnieją jedynie wewnątrz funkcji. Nazwy te nie wchodzą w konflikt z przestrzenią nazw zawierającego funkcję modułu (ani z żadną inną funkcją).

Rozróżnienie zmiennych na lokalne i globalne sprawia również, że funkcje łatwiej jest zrozumieć, ponieważ większość nazw wykorzystywanych przez funkcję pojawia się tylko w niej, a nie w dowolnym miejscu modułu. Ponieważ możemy być pewni, że nazwy lokalne nie zostaną zmodyfikowane przez jakąś inną funkcję programu, zazwyczaj pozwala to na łatwiejsze debogowanie i modyfikowanie kodu.

Zakres wbudowany

Dotychczas zakres wbudowany traktowaliśmy jak abstrakcję, jednak w rzeczywistości jest on łatwiejszy, niż można przypuszczać. Tak naprawdę zakres wbudowany to po prostu wbudowany moduł o nazwie `builtins`. Żeby jednak użyć tego modułu, należy zaimportować `builtins`, ponieważ sama nazwa `builtins` nie jest zmienną wbudowaną.

Naprawdę, nie żartuję! Zakres wbudowany zaimplementowany został jako jeden moduł biblioteki standardowej o nazwie `builtins`, jednak nazwa ta nie została umieszczona w zakresie wbudowanym, dlatego by przejrzeć zawartość tego modułu, trzeba go zaimportować. Po zrobieniu tego możemy wywołać funkcję `dir` w celu sprawdzenia, jakie nazwy zostały zdefiniowane w tym module. W Pythonie 3.0:

```
>>> import builtins
>>> dir(builtins)
['ArithError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
 ...pominięto wiele nazw...
 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr',
 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

Nazwy z tej listy składają się na zakres wbudowany w Pythonie. Mniej więcej pierwsza połowa to wbudowane wyjątki, natomiast druga to wbudowane funkcje. Na liście tej można także znaleźć nazwy specjalne `None`, `True` oraz `False`, choć są one traktowane jako słowa zarezerwowane. Ponieważ Python automatycznie przeszukuje ten moduł jako ostatni (zgodnie z regułą przeszukiwania zakresów LEGB), wszystkie nazwy z tej listy otrzymujemy za darmo — co oznacza, że możemy z nich korzystać bez konieczności importowania jakichś modułów. Z tego powodu do funkcji wbudowanej można się tak naprawdę odnieść na dwa sposoby — korzystając z reguły LEGB lub ręcznie importując moduł `builtins`.

```
>>> zip # W normalny sposób
<class 'zip'>

>>> import builtins # W bardziej skomplikowany sposób
>>> builtins.zip
<class 'zip'>
```

To drugie rozwiązanie przydaje się czasami w bardziej zaawansowanej pracy. Uważny Czytelnik z pewnością dostrzeże również, że ponieważ procedura przeszukiwania zakresów LEGB pobiera pierwszą odnalezioną nazwę, nazwy z zakresu lokalnego mogą nadpisać zmienne

o tej samej nazwie w globalnym i wbudowanym zakresie. Zmienne globalne mogą za to nadpisać nazwy wbudowane. Funkcja może na przykład utworzyć zmienną lokalną o nazwie `open`, przypisując do niej wartość.

```
def hider():
    open = 'mielonka'                                # Zmienna lokalna, ukrywa wbudowaną
    ...
    open('data.txt')                                 # Polecenie to nie otworzy w tym zakresie pliku!
```

Takie rozwiązanie spowoduje natomiast ukrycie wbudowanej funkcji `open` umieszczonej w zakresie wbudowanym (zewnętrznym). Zazwyczaj jest to błąd, i to paskudny, ponieważ Python nie wyświetli nam żadnego ostrzeżenia (czasami w bardziej zaawansowanym programowaniu zdarza się, że naprawdę chcemy zastąpić nazwę wbudowaną, redefiniując ją w kodzie). Funkcje mogą w podobny sposób ukrywać zmienne globalne za pomocą takich samych zmiennych lokalnych:

```
X = 88                                         # Zmienna globalna X

def func():
    X = 99                                       # Zmienna lokalna X: ukrywa globalną

func()
print(X)                                       # Wyświetla 88: bez zmian
```

W powyższym kodzie instrukcja przypisania wewnętrz funkcji tworzy zmienną lokalną `X`, całkowicie odrębną od zmiennej globalnej `X` z modułu znajdującego się na zewnątrz funkcji. Z tego powodu nie można zmienić zmiennej spoza funkcji bez dodania do instrukcji `def` deklaracji `global` lub `nonlocal` (zgodnie z opisem z kolejnego podrozdziału).²



Uwaga na temat zmiany w wersji: Teraz dopiero sprawy się komplikują. Wykorzystany tutaj moduł `builtins` z Pythonem 3.0 w Pythonie 2.6 nosi nazwę `_builtin_`. A żeby było jeszcze śmieszniej, nazwa `_builtins_` (z „s” na końcu) jest z góry ustawiana w większości zakresów globalnych, w tym sesji interaktywnej, i odwołuje się do modułu znanego pod nazwą `builtins` (lub `_builtin_` w 2.6).

Tym samym po zimportowaniu `builtins` wyrażenie `_builtins_` is `builtins` zwraca w Pythonie 3.0 `True`, natomiast w Pythonie 2.6 wynik `True` zwróci `_builtins_` is `_builtin_`. W rezultacie możemy zbadać zakres wbudowany, wykonując funkcję `dir(_builtins_)` bez importowania — i to zarówno w wersji 3.0, jak i 2.6. Zalecane jest jednak wykorzystywanie w Pythonie 3.0 `builtins` dla wszystkich prawdziwych działań. Kto powiedział, że udokumentowanie tych zawiłości ma być proste?

Instrukcja `global`

Instrukcja `global` oraz jej krewny `nonlocal` są jedynymi elementami w Pythonie przypominającymi instrukcję deklaracji. Nie są to jednak deklaracje typu czy rozmiaru — to *deklaracje przestrzeni nazw*. Informują one Pythona, że funkcja zamierza zmodyfikować jedną lub większą

² Z technicznego punktu widzenia w Pythonie znajduje się jeszcze jeden zakres. Zmienne pętli w wyrażeniach generatorów i złożień są lokalne dla wyrażenia w wersji 3.X (w 2.X były lokalne w generatorach, natomiast w listach składanych nie). Jest to przypadek specjalny, egzotyczny, rzadko wpływający na prawdziwy kod, różniący się od instrukcji z pętlami `for`, w których zmienne nigdy nie są lokalne.

Koniec świata w Pythonie 2.6

Oto kolejna rzecz, jaką można zrobić w Pythonie, choć raczej się nie powinno. Ponieważ nazwy `True` i `False` w wersji 2.6 są po prostu zmiennymi z zakresu wbudowanego, a nie słowami zarezerwowanymi, możemy je przypisać za pomocą instrukcji takich jak `True = False`. Nie ma się co przejmować, takim działaniem wcale nie zniszczymy logicznej spójności świata! Instrukcja ta po prostu redefiniuje słowo `True` dla zakresu, w którym się pojawia. Wszystkie pozostałe zakresy odnajdą oryginalne nazwy w zakresie wbudowanym.

Większym dowcipem jest wpisanie w Pythonie 2.6 instrukcji `_builtin_.True = False`, która zmienia `True` na `False` dla całego procesu Pythona! Takie działanie jest w Pythonie 3.0 niemożliwe, ponieważ `True` i `False` traktowane są jako prawdziwe słowa zarezerwowane, podobnie jak `None`. W wersji 2.6 wprowadza to jednak IDLE w dziwny stan paniki, który powoduje rozpoznanie procesu kodu użytkownika od nowa.

Technika ta jest jednak użyteczna zarówno dla celów zilustrowania modelu przestrzeni nazw, jak i dla autorów narzędzi, którzy muszą zmienić funkcje wbudowane, takie jak `open`, w celu dostosowania ich zachowania do własnych potrzeb. Warto również dodać, że narzędzia zewnętrzne, takie jak PyChecker, ostrzegają nas przed często popełnianymi błędami programistycznymi, w tym przypadkowym przypisaniem do wbudowanej nazwy Pythona (w aplikacji PyChecker znane jest to jako „zacielenie” funkcji wbudowanej).

liczbę zmiennych globalnych, czyli zmiennych znajdujących się w zakresie (przestrzeni nazw) modułu obejmującego funkcję.

Wspominaliśmy już wcześniej o instrukcji `global`; oto kilka słów przypomnienia:

- Nazwy globalne to zmienne przypisane na najwyższy poziomie pliku modułu.
- Nazwy globalne muszą być deklarowane jedynie wtedy, gdy przypisywane są wewnątrz funkcji.
- Do nazw globalnych można się odnosić wewnątrz funkcji bez ich deklarowania.

Innymi słowy, instrukcja `global` pozwala nam modyfikować zmienne znajdujące się poza instrukcją `def` na najwyższym poziomie pliku modułu. Jak zobaczymy później, instrukcja `nonlocal` działa prawie identycznie, jednak ma zastosowanie do zmiennych w zakresie lokalnym zawierającej instrukcji `def`, a nie do zmiennych z modułu zawierającego.

Instrukcja `global` składa się ze słowa kluczowego `global`, po którym następuje jedna lub większa liczba nazw rozdzielonych przecinkami. Po przypisaniu lub odniesieniu wewnątrz ciała funkcji wszystkie wymienione nazwy zostaną odwzorowane na zakres modułu zawierającego funkcję, jak w przykładzie poniżej.

```
X = 88 # Zmienna globalna X

def func():
    global X
    X = 99 # Zmienna globalna X: poza def

func()
print(X) # Wyświetla 99
```

Do powyższego przykładu dodaliśmy deklarację `global`, tak by zmienna `X` wewnątrz instrukcji `def` odnosiła się teraz do zmiennej `X` spoza tej instrukcji — tym razem obie są jedną zmienną. Poniżej nieco bardziej skomplikowany przykład działania instrukcji `global`.

```
y, z = 1, 2  
def all_global():  
    global x  
    x = y + z
```

Zmienne globalne w module
Deklaracja przypisanych zmiennych globalnych
Nie trzeba przypisywać y i z — działa reguła LEGB

Tutaj zmienne `x`, `y` oraz `z` są globalne wewnętrz funkcji `all_global`. Zmienne `y` oraz `z` są globalne, ponieważ są przypisane poza funkcją, natomiast zmienna `x` jest globalna, ponieważ została wymieniona w instrukcji `global`, tak by w jawnym sposobie odwzorowywać bezpośrednio zakres globalny. Bez instrukcji `global` zmienna `x` byłaby uznawana za lokalną ze względu na miejsce przypisania.

Warto zwrócić uwagę na to, że zmienne `y` oraz `z` nie są deklarowane jako globalne. Reguły wyszukiwania LEGB Pythona automatycznie odnajdują je w module. Zmienna `x` może nie istnieć w module zawierającym funkcję przed jej wykonaniem. W tym przypadku przypisanie wewnętrz funkcji tworzy zmienną `x` w module.

Minimalizowanie stosowania zmiennych globalnych

Zmienne przypisane w funkcjach są domyślnie lokalne, jeśli zatem chcemy zmodyfikować zmienne spoza funkcji, musimy wstawić do funkcji dodatkowy kod (na przykład instrukcję `global`). Takie rozwiązanie jest celowe — tak, jak często ma to miejsce w Pythonie — jeśli chcemy zrobić coś potencjalnie niewłaściwego, musimy się bardziej postarać. Choć czasami zmienne globalne są użyteczne, zmienne przypisywane w instrukcji `def` są domyślnie lokalne, ponieważ zazwyczaj jest to najlepsze rozwiązanie. Modyfikacja zmiennych globalnych może prowadzić do znanych problemów programistycznych — ponieważ wartości zmiennych uzależnione są od kolejności wywoływania odrębnych od siebie funkcji, programy mogą być bardziej trudne do debugowania.

Rozważmy na przykład następujący plik modułu.

```
X = 99  
  
def func1():  
    global X  
    X = 88  
  
def func2():  
    global X  
    X = 77
```

Wyobraźmy sobie teraz, że naszym zadaniem jest modyfikacja lub ponowne wykorzystanie tego pliku modułu. Jaką wartość będzie miała zmienna `X`? To pytanie nie ma tak naprawdę znaczenia, dopóki nie wzbogacimy go o jakiś punkt odniesienia w czasie — wartość `X` zmienia się w czasie, ponieważ uzależniona jest od tego, która funkcja została wywołana jako ostatnia (czyli od czegoś, czego nie da się powiedzieć na podstawie tego pliku).

Rezultat jest taki, że by zrozumieć kod, musimy prześledzić sterowanie w *całym programie*. I jeśli będziemy potrzebowali ten kod zmodyfikować lub ponownie wykorzystać, musimy pamiętać o całym programie jednocześnie. W tym przypadku nie możemy tak naprawdę użyć żadnej z funkcji bez skorzystania przy okazji z tej drugiej. Są one uzależnione od zmiennej globalnej (czyli z nią *powiązane*). Jest to wada zmiennych globalnych — zazwyczaj sprawiają, że kod bardziej jest zrozumieć i używać w porównaniu z kodem składającym się z samodzielnych, odrębnych funkcji opartych na zmiennych lokalnych.

Z drugiej strony, poza wykorzystywaniem programowania zorientowanego obiektowo i klas zmienne globalne są chyba najprostszym sposobem zachowywania w Pythonie współdzielonej informacji o stanie (informacji, które funkcja musi pamiętać w celu użycia przy następnym wywołaniu) — zmienne lokalne znikają po zakończeniu funkcji, natomiast globalne nie. Inne techniki, takie jak domyślne argumenty zmienne oraz zakresy funkcji zawierających, mogą pozwolić uzyskać to samo, jednak są bardziej skomplikowane od przesłania zmiennych do zakresu globalnego w celu ich przechowania.

Niektóre programy wyznaczają jeden moduł do zbierania zmiennych globalnych; dopóki takie zachowanie jest oczekiwane, nie jest to aż tak szkodliwe. Dodatkowo programy wykorzystujące wielowątkowość do wykonywania w Pythonie przetwarzania równoległego uzależnione są od zmiennych globalnych — stają się one pamięcią współdzieloną między funkcjami wykonującymi równolegle wątki, dzięki czemu działają jak narzędzia do komunikacji.³

Na razie jednak, szczególnie jeśli dopiero zaczynamy swoją przygodę z programowaniem, należy oprzeć się pokusie wykorzystywania zmiennych globalnych, gdzie tylko się da. Zamiast tego należy spróbować komunikować się za pomocą przekazywanych argumentów i zwracanych wartości. Za sześć miesięcy my sami będziemy zadowoleni z takiego wyboru, podobnie jak nasi współpracownicy.

Minimalizacja modyfikacji dokonywanych pomiędzy plikami

Poniżej kolejna kwestia związana z zakresem. Choć możemy bezpośrednio modyfikować zmienne w innym pliku, zazwyczaj nie powinniśmy tego robić. Pliki modułów zostały przedstawione w rozdziale 3. i są omawiane bardziej szczegółowo w kolejnej części niniejszej książki. By zilustrować ich związek z zakresami, rozważmy dwa poniższe pliki modułów.

```
# Plik first.py
X = 99
# Ten kod nie wie nic o pliku second.py

# Plik second.py
import first
print(first.X)
first.X = 88
# OK: referencja do zmiennej w innym pliku
# Zmodyfikowanie jej może być zbyt subtelne i niejawne
```

Pierwszy moduł definiuje zmienną `X`, drugi wyświetla ją, a następnie modyfikuje przez przypisanie. Warto zwrócić uwagę na to, że by pobrać zmienną z pierwszego modułu, musimy najpierw zainportować go do drugiego. Jak wiemy, każdy moduł jest odrębną przestrzenią nazw (pakietem zmiennych) i by z jednego modułu zobaczyć zawartość drugiego, musimy najpierw go zainportować. To właśnie najważniejsza cecha modułów — dzięki segregowaniu zmiennych według plików pozwalają one uniknąć konfliktów w zakresie nazw pomiędzy plikami.

³ Wielowątkowość powoduje wykonywanie wywołań funkcji równolegle z resztą programu i obsługiwana jest przez moduły `_thread`, `threading` oraz `queue` biblioteki standardowej Pythona (w wersji 2.6 `thread`, `threading` oraz `Queue`). Ponieważ wszystkie funkcje wątkowane wykonywane są w tym samym procesie, zakresy globalne często służą jako pamięć współdzielona między nimi. Wątkowanie jest powszechnie wykorzystywane na potrzeby dugo trwających zadań w graficznych interfejsach użytkownika, do implementowania nieblokujących operacji, a także wykorzystania zasobów procesora. Wątki wykraczają poza tematykę niniejszej książki; więcej informacji na ich temat można znaleźć w dokumentacji biblioteki Pythona oraz w pozycjach będących jej uzupełnieniem, wymienionych w „Przedmowie” (takich jak *Programming Python*).

Tak naprawdę w kontekście tematyki niniejszego rozdziału można jednak powiedzieć, że zakres globalny pliku modułu *staje się* przestrzenią nazw atrybutów obiektu modułu po zimportowaniu. Plik importujący automatycznie zyskuje dostęp do wszystkich zmiennych globalnych pliku importowanego, ponieważ jego zakres globalny po zimportowaniu staje się przestrzenią nazw atrybutów obiektu.

Po zimportowaniu pierwszego modułu drugi moduł wyświetla jego zmienną, a następnie przypisuje do niej nową wartość. Odwołanie się do zmiennej modułu w celu wyświetlenia jej jest w porządku — w taki sposób moduły normalnie łączą się ze sobą w większy system. Problem tego przypisania polega jednak na tym, że jest ono zbyt niejawne — osoby utrzymujące pierwszy moduł czy wykorzystujące go ponownie nie mają pojęcia, że jakiś daleki od niego inny moduł z łańcucha importowania może zmodyfikować ich zmienną *X* w czasie wykonywania. Tak naprawdę drugi moduł może się znajdować w innym katalogu i być trudny do zauważenia.

Choć takie modyfikacje zmiennej w różnych plikach są w Pythonie zawsze możliwe, zazwyczaj są to o wiele bardziej subtelne zmiany, niż byśmy tego chcieli. Powoduje to powstanie zbyt mocnego *związku* między dwoma plikami — ponieważ oba uzależnione są od wartości zmiennej *X*, trudno jest zrozumieć, czy można użyć ponownie jeden plik bez drugiego. Takie niejawne zależności pomiędzy plikami mogą prowadzić w najlepszym przypadku do nieelastycznego kodu, a w najgorszym — do błędów.

I tutaj najlepszą receptą jest powstrzymanie się od takiego działania. Najlepszą metodą komunikacji ponad granicami plików jest wywoływanie funkcji, przekazywanie argumentów i otrzymywanie z powrotem wartości. W tym konkretnym przypadku lepiej byłoby utworzyć *funkcję akcesora*, która byłaby odpowiedzialna za tę zmianę.

```
# Plik first.py
X = 99

def setX(new):
    global X
    X = new

# Plik second.py
import first
first.setX(88)
```

Wymaga to większej ilości kodu i może się wydawać niewielką zmianą, ale jest to rozwiązanie znacznie lepsze pod względem czytelności i możliwości utrzymywania kodu. Kiedy osoba czytająca pierwszy moduł zobaczy funkcję, będzie wiedziała, że jest to *punkt styku*, i będzie oczekiwала modyfikacji wartości zmiennej *X*. Innymi słowy, rozwiązanie to eliminuje element niespodzianki, który w projektach informatycznych rzadko jest czymś dobrym. Choć nie możemy zapobiec występowaniu zmian pomiędzy plikami, rozsądek nakazuje, by były one minimalizowane, o ile nie jest to szeroko akceptowane w programie.

Inne metody dostępu do zmiennych globalnych

Co ciekawe, ponieważ zmienne z zakresu globalnego stają się atrybutami załadowanego obiektu modułu, możemy emulować instrukcję *global*, importując moduł ją zawierający i przypisując do jego atrybutów, jak w poniższym przykładowym pliku modułu. Kod tego pliku importuje najpierw moduł zawierający za pomocą jego nazwy, a następnie indeksuje *sys.modules*, czyli tabelę załadowanych modułów (więcej informacji na ten temat znajduje się w rozdziale 21.).

```

# Plik thismod.py

var = 99                                # Zmienna globalna == atrybut modułu

def local():
    var = 0                                # Modyfikacja zmiennej lokalnej var

def glob1():
    global var
    var += 1                                # Deklaracja zmiennej globalnej (normalnej)
                                                # Modyfikacja zmiennej globalnej var

def glob2():
    var = 0                                # Modyfikacja zmiennej lokalnej var
    import thismod
    thismod.var += 1                         # Zimportowanie siebie
                                                # Modyfikacja zmiennej globalnej var

def glob3():
    var = 0                                # Modyfikacja zmiennej lokalnej var
    import sys
    glob = sys.modules['thismod']           # Zimportowanie tabeli systemowej
    glob.var += 1                            # Pobranie obiektu modułu (można też użyć __name__)
                                                # Modyfikacja zmiennej globalnej var

def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)

```

Po wykonaniu tego kodu do zmiennej globalnej zostanie dodane 3 (jedynie pierwsza funkcja nie ma wpływu na tę zmienną).

```

>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102

```

Takie rozwiązanie działa i służy jako ilustracja tego, że zmienne globalne są odpowiednikami atrybutów modułów. Wymaga ono jednak więcej pracy niż umieszczenie w kodzie instrukcji global.

Jak widzieliśmy, instrukcja global pozwala nam modyfikować zmienne w modułach poza funkcjami. Ma ona również krewniaka, instrukcję nonlocal, którą można także wykorzystać do zmiany zmiennych w funkcjach zawierających, jednak by zrozumieć, jak ona działa, musimy najpierw ogólnie omówić funkcje zawierające.

Zakresy a funkcje zagnieżdżone

Dotychczas pomijałem jedną z części reguł zakresu Pythona (celowo, ponieważ w praktyce spotyka się ją stosunkowo rzadko). Czas jednak przyjrzeć się uważniej literze *E* z reguły LEGB. Warstwa *E* jest stosunkowo nowa (została dodana w Pythonie 2.2); przybiera ona postać lokalnych zakresów wszystkich instrukcji def zawierających funkcję. Zakresy tego typu czasami nazywane są również *zakresami zagnieżdżonymi statycznie*. Tak naprawdę zagnieżdżenie to jest leksykalne — zagnieżdżone zakresy odpowiadają fizycznie i składniowo zagnieżdżonym strukturom kodu w kodzie źródłowym programu.

Szczegóły dotyczące zakresów zagnieżdzonych

Po dodaniu zakresów funkcji zagnieżdzonych reguły wyszukiwania zmiennych stały się nieco bardziej skomplikowane. Wewnątrz funkcji:

- Referencja (X) wyszukuje zmienną X najpierw w zakresie lokalnym (funkcji), a później w zakresach lokalnych wszystkich funkcji leksykalnie zawierających tę funkcję w kodzie źródłowym, od wewnętrznej do zewnętrznej. Następnie szuka w bieżącym zakresie globalnym (pliku modułu) i wreszcie w zakresie wbudowanym (module builtins). Deklaracje global sprawiają, że wyszukiwanie rozpoczyna się zamiast tego w zakresie globalnym (pliku modułu).
- Przypisanie ($X = \text{wartość}$) domyślnie tworzy bądź modyfikuje zmienną X w bieżącym zakresie lokalnym. Jeśli zmienna ta jest wewnątrz funkcji deklarowana jako *globalna*, przypisanie to zamiast tego tworzy lub modyfikuje zmienną X w zakresie modułu zawierającego funkcję. Jeśli z kolei zmienna ta jest wewnątrz funkcji deklarowana jako *nielokalna*, przypisanie modyfikuje zmienną X w zakresie lokalnym najbliższej funkcji zawierającej.

Warto zauważyć, że deklaracja *global* nadal odwzorowuje zmienne na moduł je zawierający. Kiedy obecne są funkcje zagnieżdzone, do zmiennych z funkcji zawierających inne można się odnosić, jednak by je modyfikować, niezbędna jest deklaracja *nonlocal*.

Przykład zakresu zagnieżdzonego

By wyjaśnić te kwestie w praktyce, zilustrujemy je za pomocą prawdziwego kodu. Poniżej widoczny jest przykład zakresu funkcji zawierającej.

```
X = 99 # Zmienna z zakresu globalnego — nieużywana

def f1():
    X = 88 # Zmienna lokalna z zakresu zawierającego
    def f2():
        print(X) # Referencja w zagnieżdzonej instrukcji def
    f2()
f1() # Wyświetla 88 — zmienną lokalną funkcji zawierającej
```

Po pierwsze — jest to poprawny kod w Pythonie. Instrukcja `def` jest instrukcją wykonywalną; może pojawić się w każdym miejscu, w którym pojawiają się inne instrukcje, w tym wewnątrz innej instrukcji `def`. W powyższym kodzie zagnieżdzona instrukcja `def` wykonywana jest w momencie wywołania funkcji `f1`. Generuje ona funkcję `f2` i przypisuje ją do nazwy `f2` — lokalnej zmiennej z zakresu lokalnego funkcji `f1`. W pewnym sensie `f2` jest tymczasową funkcją istniejącą jedynie w czasie wykonywania zawierającej ją funkcji `f1` i widoczną tylko dla kodu wewnątrz tej funkcji.

Warto jednak zwrócić uwagę na to, co dzieje się wewnątrz funkcji `f2`. Kiedy wyświetla ona zmienną X , odnosi się do X istniejącej w zakresie lokalnym zawierającej ją funkcji `f1`. Ponieważ funkcje mają dostęp do nazw w fizycznie je zawierających instrukcjach `def`, zmienna X z funkcji `f2` jest automatycznie odwzorowana na zmienną X z funkcji `f1` za pomocą reguły wyszukiwania LEGB.

Wyszukiwanie w zakresie zawierającym ma miejsce nawet wtedy, gdy funkcja zawierająca zakończyła już swoje działanie i zwróciła wartość. Poniższy kod definiuje na przykład funkcję tworzącą i zwracającą inną funkcję.

```
def f1():
    X = 88
    def f2():
        print(X)
        return f2
    action = f1()
    action() # Utworzenie i zwrócenie funkcji
              # Teraz wywołanie jej — wyświetla 88
```

W powyższym kodzie wywołanie `action` tak naprawdę wykonuje funkcję o nazwie `f2` po wykonaniu funkcji `f1`. Funkcja `f2` pamięta zmienną `X` z zakresu zawierającej ją funkcji `f1` nawet wtedy, gdy funkcja `f1` nie jest już aktywna.

Funkcje fabryczne

W zależności od tego, kogo zapytamy, ten rodzaj zachowania nazywany jest czasami *domknięciem* (ang. *closure*), *funkcją fabryczną* lub *funkcją fabryki* (ang. *factory function*). Pojęcia te odnoszą się do obiektu funkcji pamiętającego wartości z zakresów go zawierających bez względu na to, czy zakresy te istnieją jeszcze w pamięci. Choć klasy (opisane w szóstej części książki) najlepiej się chyba nadają do pamiętania stanu, ponieważ robią to w sposób jawnego za pomocą przy pisania atrybutów, takie funkcje są ciekawą alternatywą.

Funkcje fabryczne są na przykład czasami wykorzystywane przez programy, które muszą generować programy obsługi zdarzeń w locie w odpowiedzi na warunki w momencie wykonania (na przykład informacje od użytkowników, których nie da się przewidzieć). Przyjrzymy się na przykład poniższej funkcji.

```
>>> def maker(N):
...     def action(X):
...         return X ** N
...     return action
... 
```

Kod ten definiuje funkcję zewnętrzną, która po prostu generuje i zwraca funkcję zagnieżdżoną bez wywoływania jej. Jeśli wywołamy funkcję zewnętrzną:

```
>>> f = maker(2) # Przekazanie 2 do N
>>> f
<function action at 0x014720B0>
```

z powrotem otrzymamy referencję do wygenerowanej funkcji zagnieżdżonej, utworzonej przez wykonanie zagnieżdżonej instrukcji `def`. Jeśli teraz wywołamy to, co otrzymaliśmy z funkcji zewnętrznej:

```
>>> f(3) # Przekazanie 3 do X, N pamięta 2 — 3 ** 2
9
>>> f(4) # 4 ** 2
16
```

wywołana zostanie funkcja zagnieżdżona — nazywana w funkcji `maker` funkcją `action`. Najbardziej niezwykłym elementem jest jednak to, że funkcja zagnieżdżona pamięta liczbę całkowitą 2 — wartość zmiennej `N` w funkcji `maker` — pomimo tego, że funkcja `maker` zwróciła wartość i zakończyła działanie, zanim jeszcze wywołaliśmy funkcję `action`. W rezultacie

zmienna `N` z lokalnego zakresu funkcji zawierającej jest przechowywana jako informacja o stanie dołączona do funkcji `action`. Dzięki temu otrzymujemy wynik będący kwadratem przekazanego argumentu.

Gdybyśmy teraz znowu wywołali funkcję zewnętrzną, otrzymalibyśmy z powrotem nową funkcję zagnieżdzoną z dołączonymi innymi informacjami o stanie. Argument zostałby tym samym podniesiony do sześciadanu, a nie do kwadratu, choć oryginalna funkcja nadal podnosi wartości do kwadratu.

```
>>> g = maker(3)          # g pamięta 3, f pamięta 2
>>> g(3)                  # 3 ** 3
27
>>> f(3)                  # 3 ** 2
9
```

Powyższy kod działa, ponieważ każde wywołanie funkcji fabrycznej takie jak powyższe otrzymuje własny zbiór informacji o stanie. W naszym przypadku funkcja przypisana do zmiennej `g` pamięta 3, natomiast `f` pamięta 2, ponieważ każda z nich ma własne informacje o stanie zachowane przez zmienną `N` w funkcji `maker`.

Jest to zaawansowana technika, którą w praktyce spotyka się raczej rzadko, z wyjątkiem kodu utworzonego przez programistów mających doświadczenie w funkcjonalnych językach programowania. Z drugiej strony, zakresy funkcji zawierającej często wykorzystywane są w wyrażeniach tworzących funkcje `lambda` (omówionych w dalszej części niniejszego rozdziału). Ponieważ są one wyrażeniami, prawie zawsze zagnieżdżane są wewnętrz `def`. Co więcej, zagnieżdżanie funkcji jest powszechnie wykorzystywane w *dekoratorach* (przedstawionych w rozdziale 38.) — w niektórych przypadkach jest to najrozsądzniejszy wzorzec kodu.

Mówiąc ogólnie, *klasy* są lepsze do takiego zapamiętywania, ponieważ w nich stan przechowywany jest w sposób jawny w atrybutach. Poza używaniem klas to zmienne globalne, argumenty domyślne i referencje do zakresu zawierającego podobne do powyższych są najważniejszymi sposobami zachowywania informacji o stanie w funkcjach. By pokazać porównanie ich działania, w rozdziale 18. zamieszczono pełne omówienie argumentów domyślnych, jednak już kolejny podrozdział powinien być dobrym wprowadzeniem do tego zagadnienia.

Zachowywanie stanu zakresu zawierającego za pomocą argumentów domyślnych

We wcześniejszych wersjach Pythona dopiero co zaprezentowany kod nie miałby racji bytu, ponieważ zagnieżdżone instrukcje `def` nie miały nic wspólnego z zakresami — referencja do zmiennej wewnętrz funkcji `f2` przeszukałaby jedynie zakres lokalny (tej funkcji), następnie globalny (kod poza funkcją `f1`), a później wbudowany. Ponieważ pominęłaby zakresy funkcji zawierających `f2`, otrzymalibyśmy błąd. By obejść ten problem, programiści zazwyczaj wykorzystywali *domyślne wartości argumentów* do przekazania (i zapamiętania) obiektów w zakresie zawierającym.

```
def f1():
    x = 88
    def f2(x=x):
        print(x)
    f2()
# Wyświetla 88
```

Pamięta x z zakresu funkcji zawierającej z wartościami domyślnymi

Ten kod działa we wszystkich wydaniach Pythona i nadal można się z nim spotkać w istniejących programach napisanych w tym języku. W skrócie, składnia `argument = wartość` w nagłówku instrukcji `def` oznacza, że argument będzie miał podaną wartość domyślną, kiedy w wywołaniu nie zostanie do niego przekazana żadna prawdziwa wartość.

W zmodyfikowanej funkcji `f2` w powyższym kodzie zapis `x=x` oznacza, że argument `x` będzie miał wartość domyślną równą `x` w zakresie zawierającym — ponieważ drugie `x` zostanie oblizowane, zanim Python wejdzie do zagnieżdżonej instrukcji `def`, nadal będzie się odnosiło do zmiennej `x` z funkcji `f1`. W rezultacie wartość domyślna pamięta, czym zmienna `x` była w funkcji `f1` (w tej sytuacji obiektem 88).

Wszystko to jest stosunkowo skomplikowane i całkowicie uzależnione od momentu obliczenia wartości domyślnych. Tak naprawdę reguła przeszukiwania zakresu zagnieżdżonego została w Pythonie dodana po to, by ukrócić wykorzystywanie wartości domyślnych w tej roli. Dzisiaj Python automatycznie pamięta wszystkie wartości wymagane w zakresie zawierającym funkcję i może je wykorzystać w zagnieżdżonych instrukcjach `def`.

Oczywiście najlepszą receptą w przypadku większości kodu jest po prostu unikanie zagnieżdżania instrukcji `def` wewnętrz instrukcji `def`, co pozwoli uprościć nasz program. Poniżej znajduje się odpowiednik wcześniejszego przykładu, usuwający jedynie kwestię zagnieżdżenia. Warto zauważyć, że referencja robiona z wyprzedzeniem w kodzie jest poprawna — można wywoływać funkcję zdefiniowaną po funkcji zawierającej wywołanie, o ile druga instrukcja `def` wykonywana jest przed wywołaniem pierwszej funkcji. Kod wewnętrz instrukcji `def` nie jest obliczany do momentu właściwego wywołania funkcji.

```
>>> def f1():
...     x = 88
...     f2(x)
...
>>> def f2(x):
...     print(x)
...
>>> f1()
88
```

Przekazanie x zamiast zagnieżdżania
Referencja z wyprzedzeniem jest OK

Jeśli będziemy w taki sposób unikać zagnieżdżania, możemy właściwie zapomnieć o koncepcji zakresów zagnieżdżonych w Pythonie, o ile nie będziemy musieli tworzyć kodu zgodnego ze stylem omówionej wcześniej funkcji fabrycznej — przynajmniej w przypadku instrukcji `def`. Wyrażenia `lambda`, które w naturalny sposób pojawiają się zagnieżdżone w instrukcjach `def`, często uzależnione są od zakresów zagnieżdżonych, co zostanie wyjaśnione poniżej.

Zakresy zagnieżdżone a lambda

Choć zakresy funkcji zagnieżdżonych rzadziej są wykorzystywane w praktyce w samych instrukcjach `def`, zaczyna się pojawiać o wiele częściej, kiedy zaczynamy używać wyrażeń `lambda`. Wyrażenia tego nie będziemy omawiać aż do rozdziału 19., jednak, mówiąc w skrócie, jest to wyrażenie generujące nową funkcję, która ma być wywołana później, podobnie jak instrukcja `def`. Ponieważ jest to jednak wyrażenie, można je umieszczać w miejscach, w których instrukcja `def` nie mogłaby się pojawić, na przykład w literałach list i słowników.

Podobnie do instrukcji `def`, wyrażenie `lambda` wprowadza nowy zakres lokalny dla tworzonej przez siebie funkcji. Dzięki warstwie zakresu zawierającego wyrażenia `lambda` widzą wszystkie zmienne istniejące w funkcjach, w których są utworzone. Z tego powodu poniższy kod działa — tylko dzięki zastosowaniu nowych reguł dotyczących zakresów zagnieżdżonych.

```
def func():
    x = 4
    action = (lambda n: x ** n)           # x pamiętane z zakresu zawierającego
    return action

x = func()
print(x(2))                            # Wyświetla 16 (4 ** 2)
```

Przed wprowadzeniem zakresów funkcji zagnieżdżonych programiści wykorzystywali argumenty domyślne i przekazywali za ich pomocą wartości z zakresów zawierających do wyrażeń `lambda` (podobnie jak do instrukcji `def`). Poniższe rozwiązanie działa na przykład we wszystkich wersjach Pythona.

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)      # Ręczne przekazanie x
    return action
```

Ponieważ `lambda` jest wyrażeniem, w naturalny sposób zagnieżdżana jest wewnątrz instrukcji `def`. Z tego powodu wyrażenie to chyba najbardziej zyskało na dodaniu zakresu funkcji zawierającej do reguł wyszukiwania. W większości przypadków przekazywanie wartości do wyrażenia `lambda` za pomocą wartości domyślnych nie jest już konieczne.

Zakresy a domyślne wartości argumentów w zmiennych pętli

Istnieje jeden ważny wyjątek od reguły podanej powyżej. Jeśli wyrażenie `lambda` lub instrukcja `def` zdefiniowane są wewnątrz funkcji zagnieżdżonej wewnątrz pętli, a funkcja zagnieżdżona zawiera odniesienie do zmiennej z zakresu zawierającego, która modyfikowana jest przez tę pętlę, wszystkie funkcje wygenerowane w pętli będą miały tę samą wartość — wartość, jaką zmienna miała w ostatniej iteracji pętli.

Poniższy kod próbuje na przykład utworzyć listę funkcji pamiętających aktualną wartość zmiennej `i` z zakresu zawierającego.

```
>>> def makeActions():
...     acts = []
...     for i in range(5):
...         acts.append(lambda x: i ** x)      # Próbuje zapamiętać każde i
...     return acts                         # Wszystkie pamiętają to samo ostatnie i!
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B16B0>
```

Takie rozwiązanie jednak nie działa. Ponieważ zmienna z zakresu zawierającego wyszukiwana jest przy okazji późniejszego *wywołania* funkcji zagnieżdżonych, wszystkie one w rezultacie zapamiętają tę samą wartość (wartość, jaką zmienna pętli miała przy *ostatniej* iteracji). Dla każdej funkcji z listy otrzymujemy w rezultacie liczbę 4 do kwadratu, ponieważ we wszystkich funkcjach zmienna `i` ma tę samą wartość.

```
>>> acts[0](2)                        # Wszystkie to 4 ** 2, wartość ostatniego i
16
>>> acts[2](2)                        # Powinno być 2 ** 2
```

16

>>> acts[4](2)

16

Powinno być $4^{**} 2$

To jedyny przypadek, w którym musimy przechować wartości z zakresu funkcji zawierającej w sposób jawny za pomocą argumentów domyślnych — zamiast referencji do zakresu funkcji zawierającej. By ten kod działał, musimy przekazać aktualną wartość zmiennej zakresu funkcji zawierającej za pomocą domyślnej wartości argumentu. Ponieważ wartości domyślne obliczane są przy tworzeniu funkcji zagnieżdzonej (a nie przy jej późniejszym wywołaniu), każda pamięta swoją własną wartość dla zmiennej *i*.

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                      # Użycie domyślnych wartości argumentów
...         acts.append(lambda x, i=i: i ** x) # Pamiętanie bieżącej wartości zmiennej i
...     return acts
...
>>> acts = makeActions()
>>> acts[0](2)                            # 0 ** 2
0
>>> acts[2](2)                            # 2 ** 2
4
>>> acts[4](2)                            # 4 ** 2
16
```

To dość niezwykły przypadek, ale może przydać się w praktyce, w szczególności w kodzie generującym funkcje zwrotne dla kilku widgetów z graficznego interfejsu użytkownika (na przykład programów obsługi naciśnięcia przycisków). Więcej informacji na temat domyślnych wartości argumentów można znaleźć w rozdziale 18., natomiast o wyrażeniach lambda będziemy mówić w rozdziale 19., dlatego do niniejszego podrozdziału można również spróbować wrócić później.⁴

Dowolne zagnieżdżanie zakresów

Przed zakończeniem omówienia tej kwestii należy jeszcze odnotować, że zakresy mogą być zagnieżdżane w dowolny sposób, jednak przeszukiwane są jedynie instrukcje `def` funkcji zawierających (a nie klasy, opisane w szóstej części książki).

```
>>> def f1():
...     x = 99
...     def f2():
...         def f3():
...             print(x)                  # Znalezione w zakresie lokalnym f1!
...         f3()
...     f2()
...
>>> f1()
99
```

Python przeszukuje zakresy lokalne *wszystkich* instrukcji `def` zawierających funkcję, od wewnętrznej do zewnętrznej, po sprawdzeniu zakresu lokalnego funkcji i przed sprawdzeniem zakresu globalnego modułu. Ten rodzaj kodu raczej nie pojawia się w praktyce. W Pythonie

⁴ W części poświęconej pułapkom związanym z funkcjami, w rozdziale 20., zobaczymy, że używanie obiektów zmiennych, jak listy i słowniki, w postaci argumentów domyślnych (na przykład `def f(a=[])`) może być problematyczne. Ponieważ wartości domyślne implementowane są jako pojedyncze obiekty dołączane do funkcji, zmienne wartości domyślne zachowują stan pomiędzy wywołaniami, a nie są inicjalizowane od nowa z każdym wywołaniem. W zależności od tego, kogo o to zapytamy, jest to uznawane albo za opcję umożliwiającą przechowywanie stanu, albo za dziwną pułapkę języka. Więcej informacji na ten temat znajduje się na końcu rozdziału 20.

uważa się, że *plaska struktura jest lepsza od zagnieżdzonej* — z wyjątkiem pewnych specjalnych kontekstów nasze życie, jak i życie naszych współpracowników zdecydowanie zyska na minimalizacji liczby zagnieżdzonych definicji funkcji.

Instrukcja nonlocal

W poprzednim podrozdziale omawialiśmy sposoby *odwoływania* się w funkcjach zagnieżdzonych do zmiennych z zakresu funkcji zawierającej, nawet jeśli funkcja ta zwróciła już wartość. Okazuje się, że od Pythona 3.0 możemy także *modyfikować* tego typu zmienne z zakresu funkcji zawierającej, dopóki będziemy je deklarować w instrukcjach `nonlocal`. Dzięki tej instrukcji zagnieżdzone instrukcje `def` mają dostęp do odczytu i zapisu zmiennych znajdujących się w funkcjach je zawierających.

Instrukcja `nonlocal` jest bliskim krewnym omówionej wcześniej instrukcji `global`. Podobnie jak `global`, deklaruje ona zmienną, która zostanie zmodyfikowana w zakresie funkcji zawierającej. W przeciwieństwie do `global` instrukcja `nonlocal` ma zastosowanie do zmiennej z zakresu funkcji zawierającej, a nie do zakresu modułu poza wszystkimi instrukcjami `def`. I inaczej niż w `global` — w przypadku `nonlocal` zmienne muszą w momencie deklaracji istnieć już w zakresie funkcji zawierającej — mogą istnieć jedynie w funkcjach zawierających i nie mogą być tworzone za pomocą pierwszego przypisania w zagnieżdzonej instrukcji `def`.

Innymi słowy, instrukcja `nonlocal` zarówno pozwala na przypisywanie do zmiennych w zakresach funkcji zawierającej, jak i ogranicza przeszukiwanie zakresów dla zmiennych tego typu jedynie do instrukcji `def` zawierających funkcję. W rezultacie otrzymujemy bardziej bezpośrednią i niezawodną implementację zmieniającej się informacji o stanie dla programów, które nie chcą bądź nie potrzebują klas z atrybutami.

Podstawy instrukcji nonlocal

W Pythonie 3.0 wprowadzono nową instrukcję `nonlocal`, która ma znaczenie jedynie wewnętrz funkcji:

```
def func():
    nonlocal zmienna1, zmienna2, ...
```

Powyzsza instrukcja pozwala, by funkcja zagnieżdzona modyfikowała jedną lub większą liczbę zmiennych zdefiniowanych w zakresie funkcji zawierającej. W Pythonie 2.X (w tym w wersji 2.6), kiedy instrukcja `def` jednej funkcji była zagnieżdzona w innej, funkcja zagnieżdzona mogła odwoływać się do dowolnych zmiennych zdefiniowanych przez przypisanie w zakresie funkcji zawierającej, jednak nie mogła ich modyfikować. W wersji 3.0 zadeklarowanie zmiennych z zakresów funkcji zawierającej w instrukcji `nonlocal` pozwala funkcji zagnieżdzonej na przypisanie i tym samym również modyfikację takich zmiennych.

W ten sposób funkcje zawierające udostępniają informacje o stanie *do zapisu*, pamietane w czasie późniejszego wywołania funkcji zagnieżdzonych. Możliwość zmiany stanu sprawia, że jest on bardziej użyteczny dla funkcji zagnieżdzonej (wyobraźmy sobie na przykład licznik w zakresie funkcji zawierającej). W wersji 2.X programiści osiągali ten sam cel za pomocą klas lub innych metod. Ponieważ funkcje zagnieżdzone stały się teraz często wykorzystywany wzorcem kodu służącym do zachowywania stanu, instrukcja `nonlocal` sprawia, że zadanie to jest łatwiejsze do wykonania.

Poza umożliwieniem modyfikacji zmiennych w instrukcjach `def` funkcji zawierających instrukcja `nonlocal` wymusza także pewną kwestię w związku z referencjami. Podobnie jak instrukcja `global`, `nonlocal` powoduje, że wyszukiwanie zmiennej w instrukcji rozpoczyna się w zakresie instrukcji `def` funkcji zawierających, a nie w zakresie lokalnym dla funkcji deklarującej. Instrukcja `nonlocal` oznacza zatem: „Całkowicie pomiń mój zakres lokalny”.

Tak naprawdę zmienne wymienione w `nonlocal` w momencie dotarcia do `nonlocal` muszą być wcześniej zdefiniowane w instrukcji `def` funkcji zawierającej — inaczej zwrócony zostanie błąd. Rezultat jest podobny do `global` — instrukcja `global` oznacza, że zmienne znajdują się w module zawierającym, natomiast `nonlocal` oznacza, że znajdują się one w funkcji zawierającej. Instrukcja `nonlocal` jest przy tym nawet bardziej rygorystyczna — przeszukiwanie zakresów ograniczone jest tylko do funkcji zawierających. Zmienne nielokalne mogą się pojawić jedynie w instrukcjach `def` funkcji zawierających, a nie w zakresie globalnym modułu czy we wbudowanych modułach poza instrukcjami `def`.

Dodanie `nonlocal` nie zmienia ogólnych reguł zakresów referencji zmiennych — nadal działają one tak jak poprzednio, zgodnie z opisaną wcześniej regułą LEGB. Instrukcja `nonlocal` służy przede wszystkim do umożliwienia modyfikacji zmiennych znajdujących się w zakresach funkcji zawierających, a nie tylko odwoływania się do nich. Zarówno `global`, jak i `nonlocal` ograniczają jednak nieco reguły wyszukiwania, kiedy używa się ich w funkcji:

- Instrukcja `global` sprawia, że przeszukiwanie zakresów rozpoczyna się w zakresie modułu zawierającego, i pozwala na przypisywanie zmiennych tam się znajdujących. Jeśli zmienna nie znajduje się w module, przeszukiwanie zakresów jest kontynuowane w zakresie wbudowanym, jednak przypisania do zmiennych globalnych zawsze tworzą lub modyfikują je w zakresie modułu.
- Instrukcja `nonlocal` ogranicza przeszukiwanie zakresów do instrukcji `def` funkcji zawierających, wymaga, by zmienne już tam istniały, i pozwala na przypisywanie ich. Przeszukiwanie zakresów nie jest kontynuowane w zakresach globalnym czy wbudowanym.

W Pythonie 2.6 referencje do zmiennych z zakresów funkcji zawierającej są dozwolone, natomiast przypisania nie. Można jednak nadal wykorzystywać klasy z jawnymi atrybutami do uzyskania tego samego efektu zmieniających się informacji o stanie jak w przypadku zmiennych nielokalnych (i w niektórych kontekstach będzie to lepsze rozwiązanie). Zmienne globalne oraz atrybuty funkcji także mogą czasami posłużyć do tych samych celów. Więcej informacji na ten temat za moment — na razie przyjrzyjmy się jakiemuś działającemu kodowi, by nieco skonkretyzować podane wiadomości.

Instrukcja `nonlocal` w akcji

Przejdzmy do przykładów — wszystkich wykonywanych w Pythonie 3.0. Referencje do zakresów instrukcji `def` funkcji zawierających działają tak samo jak w wersji 2.6. W poniższym kodzie funkcja `tester` tworzy i zwraca funkcję `nested`, która będzie wywołana później, a referencja do zmiennej `state` w funkcji `nested` odwzorowana zostaje na zakres lokalny funkcji `tester` za pomocą normalnych reguł przeszukiwania zakresów:

```
C:\\\\misc>c:\\python30\\python
>>> def tester(start):
...     state = start
...     def nested(label):
...         # Referencja do zmiennej nielokalnej działa normalnie
```

```

...     print(label, state)      # Pamięta stan w zakresie funkcji zawierającej
...     return nested
...
>>> F = tester(0)
>>> F('mielonka')
mielonka 0
>>> F('szynka')
szynka 0

```

Modyfikacja zmiennej w zakresie instrukcji def funkcji zawierającej nie jest jednak domyślnie dozwolona — tak samo jest również w wersji 2.6:

```

>>> def tester(start):
...     state = start
...     def nested(label):
...         print(label, state)
...         state += 1
...     return nested
...
>>> F = tester(0)
>>> F('mielonka')
UnboundLocalError: local variable 'state' referenced before assignment

```

Domyślnie nie może się zmienić (w 2.6 też nie)

Użycie zmiennych nielokalnych w celu modyfikacji

Obecnie w Pythonie 3.0, jeśli zadeklarujemy zmienną state w zakresie funkcji tester jako nonlocal wewnętrz funkcji nested, możemy ją zmodyfikować również wewnętrz funkcji zagnieżdzonej. Takie rozwiązanie działa, mimo że funkcja tester zwróciła wartość i zakończyła działanie, zanim wywołaliśmy funkcję nested za pośrednictwem zmiennej F:

```

>>> def tester(start):
...     state = start
...     def nested(label):
...         nonlocal state
...         print(label, state)
...         state += 1
...     return nested
...
>>> F = tester(0)
>>> F('mielonka')          # Inkrementuje state z każdym wywołaniem
mielonka 0
>>> F('szynka')
szynka 1
>>> F('jajka')
jajka 2

```

Każde wywołanie otrzymuje własną zmienną state

Pamięta state z zakresu funkcji zawierającej

Można zmienić, jeśli nonlocal

Jak zwykle w przypadku referencji do zakresów funkcji zawierającej — funkcję fabryczną tester możemy wywołać kilka razy w celu otrzymania kilku kopii jej stanu w pamięci. Obiekt state w zakresie funkcji zawierającej jest dołączony do zwracanego obiektu funkcji nested. Każde wywołanie tworzy nowy, odrębny obiekt state w taki sposób, że uaktualnienie stanu jednej funkcji nie będzie miało wpływu na inną. Poniższy kod stanowi kontynuację poprzedniej interakcji:

```

>>> G = tester(42)          # Utworzenie nowej funkcji tester rozpoczynającej się od 42
>>> G('mielonka')
mielonka 42
>>> G('jajka')            # Informacje o stanie uaktualnione do 43
jajka 43
>>> F('bekon')             # Dla F pozostają takie, jakie były: 3
bekon 3

```

Przypadki graniczne

Istnieje jednak kilka elementów, na które należy uważać. Po pierwsze, w przeciwnieństwie do instrukcji global zmienne nonlocal naprawdę muszą być wcześniej przypisane w zakresie instrukcji def funkcji zawierającej, kiedy instrukcja nonlocal będzie obliczana, gdyż inaczej otrzymamy błąd — nie możemy ich tworzyć dynamicznie, przypisując na nowo w zakresie funkcji zawierającej:

```
>>> def tester(start):
...     def nested(label):
...         nonlocal state           # Zmienne nielokalne muszą już istnieć w zakresie funkcji zawierającej!
...         state = 0
...         print(label, state)
...     return nested
...
SyntaxError: no binding for nonlocal 'state' found

>>> def tester(start):
...     def nested(label):
...         global state            # Zmienne globalne nie muszą istnieć przy deklarowaniu ich
...         state = 0               # To tworzy teraz zmienną w module
...         print(label, state)
...     return nested
...
>>> F = tester(0)
>>> F('abc')
abc 0
>>> state
0
```

Po drugie, instrukcja nonlocal ogranicza przeszukiwanie zakresów jedynie do instrukcji def funkcji zawierających. Zmienne nielokalne nie są szukane w zakresie globalnym modułu zawierającego czy zakresie globalnym poza instrukcjami def, nawet jeśli tam już są:

```
>>> spam = 99
>>> def tester():
...     def nested():
...         nonlocal spam          # Musi być w instrukcji def, nie w module!
...         print('Aktualna wartość=', spam)
...         spam += 1
...     return nested
...
SyntaxError: no binding for nonlocal 'spam' found
```

Powyższe ograniczenia mają sens, jeśli sobie uświadomimy, że Python nie wiedziałby, w którym zakresie funkcji zawierającej miałby utworzyć nową zmienną. I tak w powyższym przykładzie — czy zmienna spam powinna być przypisana w funkcji tester, czy też może w zawierającym ją module? Ponieważ nie jest to oczywiste, Python musi znać zmienne nielokalne w momencie *tworzenia* funkcji, a nie jej *wywołania*.

Czemu służą zmienne nielokalne?

Biorąc pod uwagę dodatkowy poziom skomplikowania funkcji zagnieżdżonych, można się zastanawiać, po co to całe zamieszanie. Choć trudno to zobaczyć w prostych przykładach, informacje o stanie w wielu programach stają się kluczowe. Istnieje wiele sposobów „pamiętania” informacji pomiędzy wywołaniami funkcji i metod w Pythonie. Choć każdy z nich wiąże się z pewnymi kompromisami, nonlocal poprawia sytuację w przypadku referencji do zakresów

funkcji zawierających — instrukcja `nonlocal` pozwala na przechowywanie w pamięci większej liczby kopii zmieniającego się stanu, a także zaspokaja proste potrzeby w zakresie przechowywania stanu w sytuacjach, w których stosowanie klas może nie być uzasadnione.

Jak widzieliśmy w poprzednim podrozdziale, poniższy kod pozwala na przechowywanie stanu, a także modyfikowanie go w zakresie funkcji zawierającej. Każde wywołanie funkcji `tester` tworzy niewielki samodzielny pakiet zmieniających się informacji, w którym zmienne nie wchodzą w konflikt z żadnymi innymi częściami programu:

```
def tester(start):
    state = start
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
    return nested

F = tester(0)
F('mielonka')
```

Każde wywołanie otrzymuje własną zmienną state
Pamięta state z zakresu funkcji zawierającej
Można zmienić, jeśli nonlocal

Niestety, powyższy kod działa jedynie w Pythonie 3.0. Dla osób pracujących w Pythonie 2.6 w zależności od potrzeb dostępne są inne opcje. W kolejnych trzech podrozdziałach prezentujemy pewne alternatywy.

Współdzielony stan i zmienne globalne

Jednym z często spotykanych przepisów na uzyskanie efektu instrukcji `nonlocal` w Pythonie 2.6 i wcześniejszych wersjach jest po prostu przeniesienie stanu do *zakresu globalnego* (modułu zawierającego):

```
>>> def tester(start):
...     global state
...     state = start
...     def nested(label):
...         global state
...         print(label, state)
...         state += 1
...     return nested
...
>>> F = tester(0)
>>> F('mielonka')                                     # Każde wywołanie inkrementuje współdzieloną zmienną globalną state
mielonka 0
>>> F('jajka')
jajka 1
```

Takie rozwiązanie w tym przypadku działa, jednak wymaga deklaracji `global` w obu funkcjach i jest podatne na konflikty nazw zmiennych w zakresie globalnym (co będzie, jeśli nazwa zmiennej `state` jest już wykorzystywana?). Poważniejszym i bardziej subtelnym problemem jest to, że rozwiązanie to pozwala na istnienie tylko *jednej współdzielonej kopii* informacji o stanie w zakresie modułu. Jeśli znowu wywołamy funkcję `tester`, przywróci się zmiana `state` modułu do początkowej wartości, a poprzednie wywołania zobaczą, że ich zmiana `state` została nadpisana:

```
>>> G = tester(42)                                    # Przywraca wartość jednej kopii zmiennej state w zakresie globalnym
>>> G('tost')
tost 42
>>> G('bekon')
bekon 43
>>> F('szynka')                                       # Oj — mój licznik został nadpisany!
szynka 44
```

Jak pokazano wcześniej, przy użyciu `nonlocal` w miejsce `global` każde wywołanie funkcji `tester` zapamiętuje własną, unikalną kopię obiektu `state`.

Stan i klasy (zapowiedź)

Inną receptą na uzyskanie zmiennych informacji o stanie w Pythonie 2.6 oraz wcześniejszych wersjach jest wykorzystanie *klas z atrybutami*, dzięki czemu dostęp do informacji o stanie odbywa się w sposób bardziej jawnym niż w przypadku niejawnej magii reguł przeszukiwania zakresów. Dodatkową zaletą jest to, że każda instancja klasy otrzymuje świeżą kopię informacji o stanie, będącą naturalnym produktem ubocznym modelu obiektów Pythona.

Nie omawialiśmy jeszcze klas zbyt szczegółowo, jednak tytułem wstępu poniżej zamieszczamy wcześniejszy wykorzystane funkcje `tester` i `nested` utworzone jako klasy. Stan jest zapisywany w obiektach w sposób jawnym w miarę ich tworzenia. By poniższy kod miał sens, musimy wiedzieć, że instrukcja `def` wewnętrz `class` działa dokładnie tak samo jak instrukcja `def` poza `class`, z wyjątkiem faktu, iż argument `self` funkcji automatycznie otrzymuje domniemany podmiot wywołania (obiekt instancji utworzony przez wywołanie samej klasy).

```
>>> class tester:
...     def __init__(self, start):
...         self.state = start
...     def nested(self, label):
...         print(label, self.state)
...         self.state += 1
...
>>> F = tester(0)
>>> F.nested('mielonka')
mielonka 0
>>> F.nested('szynka')
szynka 1
>>> G = tester(42)
>>> G.nested('tost')
tost 42
>>> G.nested('bekon')
bekon 43
...
>>> F.nested('jajka')
# Alternatywa oparta na klasach (patrz część VI)
# Przy tworzeniu obiektu stan jest...
# ...zapisywany w nowym obiekcie w sposób jawnym
jajka 2
>>> F.state
# Jawną referencję do zmiennej state
# Modyfikacja jest zawsze dozwolona
3
# Tworzenie instancji, wywołanie __init__
# F przekazywane jest do self
# Każda instancja otrzymuje nową kopię zmiennej state
# Zmiana jednej nie wpływa na pozostałe
...
# Zmienna state dla F jest taka, na jakiej skończyliśmy
# Dostęp do state poza klasą
```

Dzięki dodatkowej odrobinie magii, którą wyjaśnimy w dalszej części książki, możemy także za pomocą przeciążania operatorów sprawić, że nasza klasa będzie przypominała wywoływalną funkcję. Metoda `__call__` przechwytyuje bezpośrednie wywołania dla instancji, dzięki czemu nie musimy wywoływać nazwanej metody:

```
>>> class tester:
...     def __init__(self, start):
...         self.state = start
...     def __call__(self, label):
...         print(label, self.state)
...         self.state += 1
...
>>> H = tester(99)
>>> H('sok')
# Przechwycenie bezpośrednich wywołań instancji
# .nested() nie jest wymagane
sok 99
>>> H('naleśniki')
# Wywołanie __call__
naleśniki 100
```

Nie ma co zajmować się szczegółami powyższego kodu na tym etapie książki — klasy omówimy dogłębnie w szóstej części, a narzędziom służącym do przeciążania operatorów, takim jak `_call_` przyjrzymy się w rozdziale 29., dlatego warto zachować ten kod na później. Najważniejsze jest to, że klasy mogą sprawić, iż informacje o stanie staną się bardziej oczywiste, wykorzystując do tego jawne przypisywanie atrybutów w miejsce przeszukiwania zakresów.

Choć wykorzystywanie klas na potrzeby informacji o stanie jest z reguły dobrym zwyczajem, w klasach tego typu, gdzie stan to pojedynczy licznik, może to być przesadą. Tak trywialne przypadki stanu są o wiele bardziej powszechnie, niż mogłoby się wydawać. W takich kontekstach zagnieżdżone instrukcje `def` są często lżejszym rozwiązaniem od tworzenia klas, zwłaszcza w przypadku osób niezaznajomionych z programowaniem zorientowanym obiektowo. Istnieją także sytuacje, w których zagnieżdżone instrukcje `def` mogą wręcz działać lepiej od klas (przykład znacznie wykraczający poza zakres tego rozdziału znajduje się w rozdziale 38., przy okazji omawiania *dekoratorów metod*).

Stan i atrybuty funkcji

W ostatniej opcji zachowania stanu efekt zmiennych nielokalnych możemy czasami uzyskać za pomocą *atributów funkcji* — zmiennych zdefiniowanych przez użytkownika dołączanych bezpośrednio do funkcji. Oto ostatnia wersja naszego przykładu oparta na tej technice. Zastępuje ona zmienną nielokalną atrybutem dołączonym do funkcji zagnieżdżonej. Choć rozwiązanie to może się niektórym osobom wydawać nieintuicyjne, pozwala ono na dostęp do zmiennej stanu *spoza* funkcji zagnieżdżonej (za pomocą zmiennych nielokalnych możemy jedynie widzieć zmienne stanu wewnętrz zagnieżdżonej instrukcji `def`):

```
>>> def tester(start):
...     def nested(label):
...         print(label, nested.state)           # nested jest w zakresie funkcji zawierającej
...         nested.state += 1                  # Miana atrybutu, a nie samej nested
...         nested.state = start              # Początkowy stan po zdefiniowaniu funkcji
...         return nested
...
>>> F = tester(0)
>>> F('mielonka')                      # F to nested z dołączonym stanem
mielonka 0
>>> F('szynka')
szynka 1
>>> F.state                            # Można także uzyskać dostęp do stanu spoza funkcji
2
>>>
>>> G = tester(42)
>>> G('jajka')                         # G ma własny stan, nie nadpisuje stanu F
jajka 42
>>> F('szynka')
szynka 2
```

Powyższy kod oparty jest na fakcie, że nazwa funkcji `nested` jest zmienną lokalną w zakresie funkcji `tester` zawierającym `nested`. Tym samym można się do niej swobodnie odwoływać wewnętrz `nested`. Kod ten polega także na tym, że modyfikacja obiektu w miejscu nie jest przypisaniem do zmiennej. Kiedy inkrementowane jest `nested.state`, modyfikowana jest część obiektu, do którego odwołuje się `nested`, a nie sama zmienna `nested`. Ponieważ tak naprawdę nie przypisujemy zmiennej w zakresie funkcji zawierającej, nie potrzebujemy instrukcji `nonlocal`.

Jak widać, zmienne globalne, nielokalne, klasy oraz atrybuty funkcji oferują możliwość przechowania stanu. Zmienne globalne obsługują jedynie dane współdzielone, klasy wymagają

podstawowej znajomości programowania zorientowanego obiektowo, a i klasy, i atrybuty funkcji pozwalają na dostęp do stanu spoza samej funkcji zagnieżdżonej. Jak zawsze wybór najlepszego narzędzia na potrzeby programu uzależniony jest od jego celów.⁵

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy jedną z dwóch kluczowych koncepcji związanych z funkcjami — *zakresy* (sposób wyszukiwania zmiennych, kiedy są one wykorzystywane). Jak już wiemy, zmienne są uznawane za lokalne dla definicji funkcji, w której zostały przypisane, o ile nie zostały specjalnie zadeklarowane jako globalne lub nielokalne. Omówiliśmy również kilka bardziej zaawansowanych koncepcji związanych z zakresami, w tym zakresy funkcji zagnieżdżonych oraz atrybuty funkcji. Wreszcie przyjrzelismy się kilku ogólnym koncepcjom związanym z zakresami, takim jak unikanie zmiennych globalnych i modyfikacji pomiędzy plikami.

W kolejnym rozdziale będziemy kontynuować omówienie funkcji, zgłębiając drugą kluczową koncepcję z nimi związaną — przekazywanie argumentów. Jak zobaczymy, argumenty przekazywane są do funkcji przez przypisanie, jednak Python udostępnia także narzędzia pozwalające na pewną elastyczność w zakresie przekazywania elementów. Przed przejściem do tych zagadnień czas zająć się quizem sprawdzającym kwestie związane z zakresami omówione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Jaki będzie wynik poniższego kodu i dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
...     print(X)  
...  
>>> func()
```

2. Jaki będzie wynik poniższego kodu i dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
...     X = 'NI'  
...  
>>> func()  
>>> print(X)
```

3. Co wyświetla poniższy kod i dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
...     X = 'NI'  
...     print(X)
```

⁵ Atrybuty funkcji są obsługiwane zarówno w Pythonie 2.6, jak i 3.X. Omówimy je bardziej szczegółowo w rozdziale 19., a w rozdziale 38. powrócimy do wszystkich omówionych tutaj opcji przechowywania stanu w bardziej realistycznym kontekście. Warto zauważyć, że można także modyfikować zmienne obiekty w zakresach funkcji zawierającej w wersjach 2.X oraz 3.X bez deklarowania zmiennej jako nielokalnej (na przykład state=[start] w tester, state[0]+=1 w nested), choć jest to nieco bardziej niejasne od atrybutów funkcji lub instrukcji nonlocal z wersji 3.X.

```
...  
>>> func()  
>>> print(X)
```

4. Jakie dane wyjściowe zwraca poniższy kod i ponownie: dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
...     global X  
...     X = 'NI'  
...  
>>> func()  
>>> print(X)
```

5. A co z tym kodem? Jaki będzie jego wynik i dlaczego?

```
>>> X = 'Mielonka'  
>>> def func():  
...     X = 'NI'  
...     def nested():  
...         print(X)  
...     nested()  
...  
>>> func()  
>>> X
```

6. Co z tym przykładem? Jaki będzie jego wynik w Pythonie 3.0 i dlaczego?

```
>>> def func():  
...     X = 'NI'  
...     def nested():  
...         nonlocal X  
...         X = 'Mielonka'  
...     nested()  
...     print(X)  
...  
>>> func()
```

7. Należy podać trzy lub większą liczbę sposobów przechowania informacji o stanie w funkcji Pythona.

Sprawdź swoją wiedzę — odpowiedzi

- Odpowiedź brzmi: 'Mielonka', ponieważ funkcja odnosi się do zmiennej globalnej z modułu ją zawierającego (ponieważ zmienna nie jest przypisana wewnątrz funkcji, uznawana jest za globalną).
- Odpowiedź znowu brzmi: 'Mielonka', ponieważ przypisanie zmiennej wewnątrz funkcji sprawia, że jest ona lokalna i w rezultacie ukrywa zmienną globalną o tej samej nazwie. Instrukcja print odnajduje niezmodyfikowaną zmienną w zakresie globalnym (modułu).
- Wyświetla 'NI' w pierwszym wierszu, a 'Mielonka' w drugim, ponieważ referencja do zmiennej wewnątrz funkcji odnajduje przypisaną zmienną lokalną, natomiast referencja z print odnajduje zmienną globalną.
- Tym razem wyświetla tylko 'NI', ponieważ deklaracja globalna sprawia, że zmienna przypisana wewnątrz funkcji odnosi się do zmiennej z zawierającego tę funkcję zakresu globalnego.

5. Wynikiem znowu będzie 'NI' w pierwszym wierszu i 'Mielonka' w drugim, ponieważ instrukcja print z zagnieżdżonej funkcji odnajduje nazwę w lokalnym zakresie funkcji zawierającej, natomiast instrukcja print na końcu odnajduje zmienną w zakresie globalnym.
6. Przykład ten wyświetla 'Mielonka', ponieważ instrukcja nonlocal (dostępna w Pythonie 3.0, ale nie w 2.6) powoduje, że przypisanie do X wewnętrz funkcji zagnieżdżonej modyfikuje X w zakresie lokalnym funkcji zawierającej. Bez tej instrukcji przypisanie to zaklasyfikowałoby X jako zmienną lokalną dla funkcji zagnieżdżonej, przez co byłaby to inna zmienna. Kod wyświetliłby wtedy 'NI'.
7. Choć wartości zmiennych lokalnych znikają, kiedy funkcja zwraca wartość, informacje o stanie można w funkcji Pythona zachować za pomocą współdzielonych zmiennych globalnych, referencji do zakresów funkcji zawierających wewnętrz funkcji zagnieżdżonych lub domyślnych wartości argumentów. Atrybuty funkcji pozwalają czasami dołączać stan do samej funkcji, zamiast wyszukiwać go w zakresach. Alternatywa polegająca na zastosowaniu programowania zorientowanego obiektywnego i klas obsługuje czasami zachowanie informacji o stanie w lepszy sposób od wymienionych technik opartych na zakresach, ponieważ odbywa się to w sposób jawny za pomocą przypisania atrybutów; opcję tę omówimy w szóstej części książki.

Argumenty

W rozdziale 17. omówiliśmy szczegóły dotyczące *zakresów Pythona* — miejsc, w których zmienne są definiowane i wyszukiwane. Wiemy już, że miejsce zdefiniowania zmiennej w kodzie określa w dużej mierze jej znaczenie. W niniejszym rozdziale kontynuujemy omawianie funkcji, przechodząc do koncepcji *przekazywania argumentów* w Pythonie, czyli sposobu, w jaki obiekty przesyłane są do funkcji w charakterze danych wejściowych. Jak się niebawem przekonamy, argumenty (inaczej nazywane parametrami) przypisywane są do zmiennych w funkcji, jednak mają więcej wspólnego z referencjami do obiektów niż z zakresami zmiennych. Zobaczmy także, że Python udostępnia dodatkowe narzędzia, takie jak słowa kluczowe, wartości domyślne oraz kolektory dowolnych argumentów, pozwalające na większą swobodę w przesyłaniu argumentów do funkcji.

Podstawy przekazywania argumentów

Wcześniej w tej części książki napisałem, że argumenty przekazywane są przez *przypisanie*. Ma to kilka konsekwencji, które nie zawsze są oczywiste dla osób poczynających, a które wyjaśnię w niniejszym podrozdziale. Poniżej znajduje się przegląd najważniejszych kwestii związanych z przekazywaniem argumentów do funkcji.

- **Argumenty przekazywane są przez automatyczne przypisanie obiektów do nazw zmiennych lokalnych.** Argumenty funkcji — referencje do (potencjalnie) współdzielonych obiektów, do których odnosi się wywołujący — są kolejnym przykładem działania przypisania w Pythonie. Ponieważ referencje implementowane są jako wskaźniki, wszystkie argumenty są w rezultacie przekazywane za pomocą wskaźników. Obiekty przekazane jako argumenty nigdy nie są automatycznie kopowane.
- **Przypisanie do nazw argumentów wewnętrz funkcji nie wpływa na wywołującego.** Nazwy argumentów w nagłówku funkcji stają się w czasie wykonywania funkcji nowymi zmiennymi lokalnymi w zakresie tej funkcji. Nazwy argumentów funkcji i nazwy zmiennych w zakresie wywołującego nie są aliasami.
- **Modyfikacja zmiennego obiektu argumentu w funkcji może mieć wpływ na wywołującego.** Z drugiej strony, ponieważ argumenty są po prostu przypisywane do przekazywanych obiektów, funkcje mogą modyfikować w miejscu przekazywane obiekty typu zmiennego, a wynik może wpływać na wywołującego. Argumenty zmienne mogą być danymi wejściowymi oraz wyjściowymi funkcji.

Więcej informacji na temat *referencji* można znaleźć w rozdziale 6. Wszystkie informacje z tego rozdziału mają zastosowanie również do argumentów funkcji, choć przypisanie do nazw argumentów jest automatyczne i niejawne.

Model przekazywania przez przypisanie w Pythonie nie jest do końca tym samym co opcja przekazywania argumentów przez referencje z języka C++, jednak w praktyce okazuje się podobny do modelu przekazywania argumentów z języka C.

- **Argumenty niezmienne przekazywane są przez wartość.** Obiekty takie, jak liczby całkowite oraz łańcuchy znaków, przekazywane są przez referencję do obiektu, a nie kopiowanie. Ponieważ jednak nie można zmodyfikować obiektów zmiennych w miejscu, efekt jest taki, jakbyśmy sporządzili kopię.
- **Argumenty zmienne przekazywane są przez wskaźnik.** Obiekty takie, jak listy i słowniki, są również przekazywane przez referencję do obiektu, co przypomina sposób przekazywania tablic jako wskaźników w języku C. Obiekty zmienne mogą być modyfikowane w miejscu w funkcji, podobnie do tablic z języka C.

Oczywiście dla osób, które nigdy nie używały języka C, tryb przekazywania argumentów z Pythona będzie się wydawał jeszcze prostszy — obejmuje on po prostu przypisanie obiektów do nazw i działa tak samo bez względu na to, czy obiekty są zmienne, czy niezmienne.

Argumenty a współdzielone referencje

W celu zilustrowania działania właściwości przekazywania argumentów rozważmy poniższy kod:

```
>>> def f(a):                                # a przypisane zostaje (referencja) do przekazanego obiektu
...     a = 99                                # Modyfikacja tylko zmiennej lokalnej a
...
>>> b = 88
>>> f(b)                                    # a i b oba początkowo odwołują się do tego samego obiektu 88
>>> print(b)                                # b się nie zmienia
88
```

W powyższym przykładzie w momencie wywołania funkcji za pomocą `f(b)` do zmiennej `a` przypisany zostaje obiekt 88, jednak `a` istnieje jedynie wewnątrz wywołanej funkcji. Modyfikacja `a` wewnątrz funkcji nie ma wpływu na miejsce, w którym funkcja ta jest wywołana — po prostu zmienia zmienią lokalną `a` na zupełnie inny obiekt.

To właśnie mamy na myśli, mówiąc o braku *aliasów* zmiennych — przypisanie do nazwy argumentu wewnątrz funkcji (na przykład `a=99`) nie zmienia magicznie zmiennej takiej jak `b` w zakresie wywołania funkcji. Nazwy argumentów mogą początkowo współdzielić przekazywane obiekty (są one tak naprawdę wskaźnikami do tych obiektów), jednak tylko tymczasowo, za pierwszym wywołaniem funkcji. Po ponownym przypisaniu nazwy argumentu ten związek zanika.

Tak właśnie jest przynajmniej w przypadku przypisywania do samych *nazw* argumentów. Kiedy do argumentów przekazywane są obiekty *zmienne*, takie jak listy i słowniki, musimy także mieć świadomość tego, że modyfikacje takich *obiektów* w miejscu mogą istnieć również po wyjściu z funkcji i tym samym wpływają na kod wywołujący. Oto przykład demonstrujący takie działanie:

```
>>> def changer(a, b):                      # Do argumentów przypisano referencje do obiektów
...     a = 2                                  # Zmiana wartości jedynie zmiennej lokalnej
...     b[0] = 'mielonka'                       # Zmiana współdzielonego obiektu w miejscu
```

```

...
>>> X = 1
>>> L = [1, 2]
>>> changer(X, L)
# Wywołujący
# Przekazanie obiektów niezmiennych i zmiennych
>>> X, L
# X bez zmian, L jest inne
(1, ['mielonka', 2])

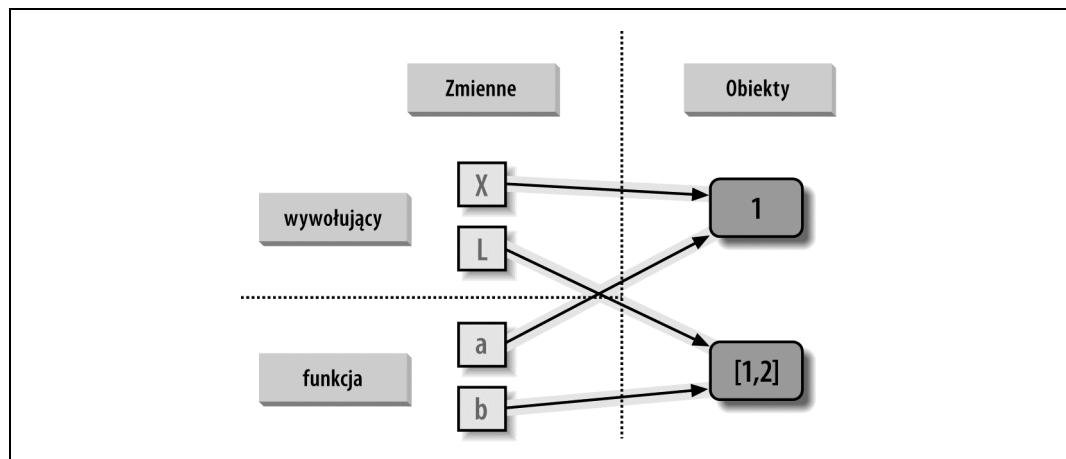
```

W powyższym kodzie funkcja `changer` przypisuje wartości do samego argumentu a oraz komponentu obiektu, do którego odnosi się argument b. Te dwa przypisania wewnątrz funkcji jedynie nieznacznie różnią się składnią, jednak dają zupełnie inne rezultaty.

- Ponieważ a jest zmienną lokalną w zakresie funkcji, pierwsze przypisanie nie ma wpływu na kod wywołujący — po prostu modyfikuje ono zmienną lokalną a, tak by odwoływała się do zupełnie innego obiektu, i nie zmienia wiązania nazwy X w zakresie wywołującym. Działa to tak samo jak w poprzednim przykładzie.
- Argument b również jest zmienną lokalną, jednak przekazuje się do niego obiekt zmienny (listę, do której w kodzie wywołującym odwołuje się L). Ponieważ drugie przypisanie jest modyfikacją obiektu w miejscu, wynik przypisania do b[0] w funkcji ma wpływ na wartość L po zwróceniu tej funkcji.

Tak naprawdę druga instrukcja przypisania w funkcji `changer` nie modyfikuje b, zmienia jedynie część obiektu, do którego aktualnie odnosi się b. Ta zmiana w miejscu ma wpływ jedynie na kod wywołujący, ponieważ zmodyfikowany obiekt istnieje dłużej od wywołania funkcji. Nazwa L także się nie zmieniła — nadal odwołuje się do tego samego, zmodyfikowanego obiektu — jednak wydaje się, jakby L było inne po wywołaniu, ponieważ wartość, do której odwołuje się ta zmienna, została zmodyfikowana wewnątrz funkcji.

Na rysunku 18.1 zilustrowano wiązania między nazwami a obiektami istniejące tuż po wywołaniu funkcji, a przed wykonaniem jej kodu.



Rysunek 18.1. Referencje a argumenty. Ponieważ argumenty przekazywane są przez przypisanie, nazwy argumentów w funkcji mogą współdzielić obiekty ze zmiennymi z zakresu wywołania. Z tego powodu modyfikacja zmiennych argumentów funkcji w miejscu może mieć wpływ na kod wywołujący. Na rysunku widać, że zmienne a i b z funkcji na początku są referencjami do obiektów, do których przy pierwszym wywołaniu funkcji odnoszą się zmienne X oraz L. Modyfikacją listy przez zmienną b sprawia, że po zakończeniu wywołania lista L okazuje się inna

Jeśli przykład ten nadal wydaje się niezrozumiały, może nam pomóc wiedza, że wynik automatycznego przypisania przekazanych argumentów jest taki sam jak wykonanie serii prostych instrukcji przypisania. Jeśli chodzi o pierwszy argument, przypisanie nie ma wpływu na wywołującego.

```
>>> X = 1
>>> a = X
>>> a = 2
>>> print(X)
1
```

Współdzielę jeden obiekt
Przestawia tylko 'a', 'X' nadal jest równe 1

Przypisanie z drugiego argumentu wpływa natomiast na zmienną w wywołującym, ponieważ jest to modyfikacja obiektu w miejscu.

```
>>> L = [1, 2]
>>> b = L
>>> b[0] = 'mielonka'
>>> print(L)
['mielonka', 2]
```

Współdzielę ten sam obiekt
Modyfikacja w miejscu: 'L' także widzi zmianę

Jeśli przypomnimy sobie naszą dyskusję na temat współdzielonych obiektów zmiennych z rozdziałów 6. i 9., rozpoznamy tutaj działanie tego samego zjawiska. Modyfikacja zmiennego obiektu w miejscu może mieć wpływ również na inne referencje do tego obiektu. Tutaj rezultat polega na sprawieniu, by jeden z argumentów działał zarówno jako dane wejściowe, jak i *dane wyjściowe* funkcji.

Unikanie modyfikacji zmiennych argumentów

Takie działanie modyfikacji zmiennych argumentów w miejscu nie jest błędem — to po prostu sposób przekazywania argumentów w Pythonie. Argumenty domyślnie są w Pythonie przekazywane do funkcji przez referencję (czyli wskaźnik), ponieważ takiego czegoś normalnie chcemy. Oznacza to, że możemy przekazywać w programach duże obiekty bez wykonywania przy okazji kilku ich kopii. Możemy również z łatwością po drodze uaktualnić te obiekty. Tak naprawdę, jak zobaczymy w szóstej części książki, model klas Pythona jest *uzależniony* od modyfikacji przekazanych argumentów `self` w miejscu w celu uaktualnienia stanu obiektu.

Jeśli jednak nie życzymy sobie, by modyfikacje w miejscu wewnętrz funkcji wpływały na obiekty, które do niej przekazujemy, możemy po prostu w jawny sposób wykonywać kopie obiektów zmiennych, w taki sam sposób jak robiliśmy to w rozdziale 6. W przypadku argumentów funkcji możemy zawsze skopiować listę w momencie wywołania.

```
L = [1, 2]
changer(X, L[:])
```

Przekazanie kopii, tak by 'L' się nie zmieniło

Mogimy również dokonać kopiowania wewnętrz samej funkcji, jeśli nigdy nie chcemy modyfikować przekazanych obiektów bez względu na sposób wywołania funkcji.

```
def changer(a, b):
    b = b[:]
```

Kopia listy w celu uniknięcia wpływu na wywołującego

```
    a = 2
    b[0] = 'mielonka'
```

Modyfikacja tylko kopii listy

Oba schematy kopiowania nie zatrzymują modyfikacji obiektu w funkcji, a jedynie zapobiegają przeniesieniu tych zmian na wywołującego. By naprawdę zapobiec modyfikacjom, zawsze możemy przekształcić obiekty na niezmienne. Przykładowo krótki zwracają wyjątek, kiedy próbuje się na nich dokonać zmian.

```
L = [1, 2]
changer(X, tuple(L))
```

Przekazanie krotki — zmiany będą błędami

Takie rozwiązanie wykorzystuje wbudowaną funkcję `tuple`, która tworzy nową krotkę ze wszystkich elementów sekwencji (a tak naprawdę z dowolnego obiektu, na którym można wykonywać iterację). To nieco ekstremalne rozwiązanie — ponieważ wymusza napisanie funkcji w taki sposób, by nigdy nie modyfikowała ona przekazywanych argumentów, rozwiązanie to może narzucać funkcji więcej ograniczeń, niż powinno, dlatego należy go raczej unikać. Nigdy nie wiemy, kiedy modyfikacja argumentów może się w przyszłości przydać do innych wywołań. Korzystanie z tej techniki sprawi również, że funkcja straci możliwość wywoływania na argumentach metod list, w tym tych, które wcale nie modyfikują obiektu w miejscu.

Najważniejsze, co należy zapamiętać, to to, że funkcje mogą aktualniać przekazane do nich zmienne obiekty (na przykład listy i słowniki). Nie musi to być problemem, jeśli się tego spodziewamy, i często jest przydatne. Co więcej, funkcje modyfikujące przekazane zmienne obiekty w miejscu zostały najprawdopodobniej celowo zaprojektowane do takiego działania — zmiana ta jest częścią dobrze zdefiniowanego API, którego nie powinniśmy naruszać, wykonując kopie.

Trzeba o tym jednak pamiętać. Jeśli jakiś obiekt niespodziewanie się nam zmienia, należy sprawdzić, czy winowiącą nie jest wywołana funkcja, i w razie potrzeby wykonać kopie przekazywanych obiektów.

Symulowanie parametrów wyjścia

Omówiliśmy już instrukcję `return` i wykorzystaliśmy ją nawet w kilku przykładach. A oto kolejny sposób wykorzystania tej instrukcji. Ponieważ instrukcja `return` może zwrócić dowolny rodzaj obiektu, może również zwracać *wiele wartości*, pakując je w krotkę czy inny typ kolekcji. Tak naprawdę, choć Python nie obsługuje czegoś, co niektóre języki programowania nazywają przekazywaniem argumentów z „wywołaniem przez referencję”, zazwyczaj możemy to symulować, zwracając krotki i przypisując wyniki z powrotem do oryginalnych zmiennych z kodu wywołującego.

```
>>> def multiple(x, y):
...     x = 2
...     y = [3, 4]
...     return x, y
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)
```

Modyfikacja jedynie zmiennych lokalnych

Zwrócenie nowych wartości w krotce

Przypisanie wyników do zmiennych wywołującego

Wygląda to tak, jakby kod zwracał dwie wartości, jednak tak naprawdę zwraca jedną — krotkę dwuelementową z pominiętymi opcjonalnymi nawiasami. Po zwróceniu wartości z funkcji możemy wykorzystać przypisanie krotek do rozpakowania części zwracanej krotki. Jeśli zapomnieliśmy w międzyczasie, dlaczego takie rozwiązanie działa, warto wrócić do podrozdziału „Krotki” z rozdziału 4., rozdziału 9. oraz podrozdziału „Instrukcje przypisania” z rozdziału 11. Rezultatem takiego wzorca programistycznego będzie symulowanie parametrów wyjścia z innych języków programowania za pomocą jawnego przypisania. Zmienne `X` i `L` mogą się zmienić po wywołaniu, ale tylko dlatego, że jest to nakazane w kodzie.



Rozpakowywanie argumentów w Pythonie 2.X: Poprzedni przykład rozpakowuje krotkę zwróconą przez funkcję za pomocą przypisania krotki. W Pythonie 2.6 można również automatycznie rozpakowywać krotki w argumentach przekazanych do funkcji. W wersji 2.6 funkcja zdefiniowana za pomocą poniższego nagłówka:

```
def f((a, (b, c))):
```

może być wywołana z krotkami, które pasują do oczekiwanej struktury — `f((1, (2, 3)))` przypisuje `a`, `b` oraz `c` do, odpowiednio, `1`, `2` i `3`. Oczywiście przekazana krotka może także być obiektem utworzonym przed wywołaniem (`f(T)`). Taka składnia instrukcji `def` nie jest już obsługiwana w Pythonie 3.0. Zamiast tego funkcję należy zapisać w poniższy sposób:

```
def f(T): (a, (b, c)) = T
```

w celu rozpakowania krotki w jawnej instrukcji przypisania. Ta jawnia postać działa zarówno w wersji 3.0, jak i 2.6. Rozpakowywanie argumentów jest nieco niejasną i rzadko wykorzystywaną opcją w Pythonie 2.X. Co więcej, nagłówki funkcji w wersji 2.6 obsługują jedynie przypisanie sekwencji z wykorzystaniem krotki. Bardziej uniwersalne przypisania sekwencji (na przykład `def f((a, [b, c])):`) zwracają błędy składni także w Pythonie 2.6 i wymagają stosowania formy z jawnym przypisaniem.

Składnia argumentów z rozpakowywaniem krotek jest również zabroniona w wersji 3.0 w listach argumentów funkcji instrukcji `lambda`. Przykład znajduje się w ramce „Znaczenie list składanych oraz funkcji `map`” w rozdziale 20. Nieco asymetryczne jest to, że przypisanie z rozpakowaniem krotki nadal jest jednak automatyczne w pętlach `for` Pythona 3.0; przykłady takiego działania można znaleźć w rozdziale 13.

Specjalne tryby dopasowania argumentów

Jak widzieliśmy przed chwilą, argumenty są w Pythonie zawsze przekazywane przez *przypisanie*. Nazwy z nagłówka `def` są przypisywane do przekazywanych obiektów. Na bazie tego modelu Python udostępnia dodatkowe narzędzia modyfikujące sposób *dopasowywania* obiektów argumentów z wywołania do nazw argumentów z nagłówka przed przypisaniem. Wszystkie te narzędzia są opcjonalne, jednak pozwalały na tworzenie funkcji obsługujących bardziej elastyczne wzorce wywołania. Możliwe jest także napotkanie ich w niektórych bibliotekach.

Domyślnie argumenty dopasowywane są za pomocą pozycji, od lewej do prawej strony, i musimy przekazać dokładnie taką samą liczbę argumentów co nazw argumentów z nagłówka funkcji. Można jednak również określić dopasowanie po nazwie, wartościach domyślnych oraz kolektorach w celu podania dodatkowych argumentów.

Podstawy

Zanim zagłębimy się w szczegóły składniowe, chciałbym podkreślić, że tryby specjalne są opcjonalne i dotyczą jedynie dopasowania obiektów do nazw. Mechanizmem przekazywania po wykonaniu dopasowania nadal jest przypisanie. Tak naprawdę niektóre z tych narzędzi przeznaczone są bardziej dla osób piszących własne biblioteki niż dla programistów aplikacji. Ponieważ jednak można na nie trafić, nawet jeśli nie tworzy się ich samodzielnie, poniżej znajduje się podsumowanie dostępnych narzędzi.

Pozycyjne — dopasowanie od lewej do prawej strony

Normalny przypadek, z którego najczęściej dotychczas korzystaliśmy. Wartości przekazanych argumentów dopasowywane są do nazw argumentów w nagłówku funkcji zgodnie z pozycją, od lewej do prawej strony.

Słowa kluczowe — dopasowanie po nazwie argumentu

Kod wywołujący może również określać, który argument funkcji ma otrzymać wartość, wykorzystując w wywołaniu jego nazwę za pomocą składni *nazwa=wartość*.

Wartości domyślne — określenie wartości dla argumentów, które nie zostały przekazane

Same funkcje mogą określać wartości domyślne, jakie mają otrzymać argumenty, kiedy wywołanie przekaże za mało wartości. Znów w użyciu pozostaje składnia *nazwa=wartość*.

Nieznana liczba argumentów (zbieranie) — zebranie dowolnej liczby argumentów zgodnie z pozycją lub słowem kluczowym

Funkcje mogą wykorzystywać argumenty specjalne poprzedzone jednym lub dwoma znakami * w celu zebrania dowolnej liczby dodatkowych argumentów (czasami określa się takie argumenty mianem *varargs*, od opcji o tej samej nazwie w języku C, również obsługującej listy argumentów o zmiennej długości).

Nieznana liczba argumentów (rozpakowywanie) — przekazanie dowolnej liczby argumentów zgodnie z pozycją lub słowem kluczowym

Kod wywołujący może również użyć składni z * do rozpakowania kolekcji argumentów na pojedyncze, osobne argumenty. Jest to przeciwieństwo użycia * w nagłówku funkcji — w nagłówku oznacza zebranie dowolnej liczby argumentów, natomiast w wywołaniu jest to przekazanie dowolnej liczby argumentów.

Argumenty mogące być tylko słowami kluczowymi — argumenty, które muszą być przekazywane przez nazwę

W Pythonie 3.0 (jednak nie w wersji 2.6) funkcje mogą określać argumenty, które muszą być przekazywane przez nazwę za pomocą argumentów ze słowami kluczowymi, a nie zgodnie z pozycją. Argumenty tego typu są zazwyczaj wykorzystywane do definiowania opcji konfiguracyjnych obok zwykłych argumentów.

Składnia dopasowania

W tabeli 18.1 przedstawiono składnię wywołującą specjalne tryby dopasowania argumentów.

Powyższe specjalne tryby dopasowywania argumentów dzielą się na wywołania i definicje funkcji w następujący sposób:

- W *wywołaniu funkcji* (pierwszych czterech wierszach tabeli) proste wartości dopasowywane są po pozycji, jednak użycie formy *nazwa=wartość* mówi Pythonowi, że ma zamiast tego dopasować do argumentów po nazwie — są to zatem argumenty ze słowami kluczowymi. Użycie form **sekwencja* lub ***słownik* w wywołaniu pozwala nam na spakowanie dowolnej liczby obiektów pozycyjnych lub słów kluczowych, odpowiednio, w sekwencjach lub słownikach i rozpakowanie ich do postaci odrębnych, pojedynczych argumentów przy przekazaniu ich do funkcji.
- W *nagłówku funkcji* (reszta tabeli) prosta nazwa dopasowywana jest po pozycji lub nazwie, w zależności od sposobu przekazania przez kod wywołujący. Forma *nazwa=wartość* określa domyślną wartość argumentu. Forma **nazwa* zbiera dodatkowe niedopasowane

Tabela 18.1. Formy dopasowywania argumentów funkcji

Składnia	Lokalizacja	Interpretacja
<code>func(wartość)</code>	Wyołujący	Normalny argument — dopasowanie po pozycji
<code>func(nazwa=wartość)</code>	Wyołujący	Słowo kluczowe — dopasowanie po nazwie
<code>func(*sekwencja)</code>	Wyołujący	Przekazanie wszystkich obiektów z sekwencji jako pojedynczych argumentów pozycyjnych
<code>func(**słownik)</code>	Wyołujący	Przekazanie wszystkich par klucz-wartość ze słownika jako pojedynczych argumentów-słów kluczowych
<code>def func(nazwa)</code>	Funkcja	Normalny argument — dopasowuje przekazane wartości po pozycji lub nazwie
<code>def func(nazwa=wartość)</code>	Funkcja	Domyślna wartość argumentu wykorzystana, jeśli wartość nie została przekazana w wywołaniu
<code>def func(*nazwa)</code>	Funkcja	Dopasowuje i zbiera pozostałe argumenty pozycyjne (w krotce)
<code>def func(**nazwa)</code>	Funkcja	Dopasowuje i zbiera pozostałe argumenty-słowa kluczowe (w słowniku)
<code>def func(*argumenty, nazwa)</code>	Funkcja	Argumenty, które w wywołaniu muszą być przekazane za pomocą słowa kluczowego (Python 3.0)
<code>def func(*, nazwa=wartość)</code>		

argumenty pozycyjne w krotkę, a forma `**nazwa` zbiera dodatkowe argumenty-słowa kluczowe w słownik. W Pythonie 3.0 oraz wersjach późniejszych wszystkie normalne nazwy argumentów lub argumenty z wartościami domyślnymi znajdujące się po `*nazwa` lub samym znaku `*` są argumentami mogącymi być tylko *słowa kluczowymi* i muszą w wywołaniach być przekazywane za pomocą słów kluczowych.

Z tych wszystkich propozycji w kodzie napisanym w Pythonie najczęściej używane są argumenty-słowa kluczowe oraz wartości domyślne. Z obu form korzystaliśmy już w sposób nieformalny we wcześniejszych częściach książki:

- *Słowa kluczowe* wykorzystaliśmy już do podawania opcji funkcji `print` z wersji 3.0, jednak mają one także bardziej uniwersalne zastosowanie. Słowa kluczowe pozwalają nam na opatrzenie dowolnego argumentu jego nazwą, tak by wywołania były bardziej znaczące.
- Z *wartościami domyślnymi* również spotkaliśmy się już wcześniej, jako ze sposobem przekazywania wartości z zakresu funkcji zawierającej, jednak tak naprawdę także one są bardziej uniwersalne. Pozwalają nam na sprawienie, że każdy argument funkcji będzie opcjonalny, i podają wartość domyślną w definicji funkcji.

Jak zobaczymy wkrótce, połączenie wartości domyślnych w nagłówkach funkcji oraz słów kluczowych w wywołaniach pozwala nam wybierać, które wartości domyślne chcemy nadpisać.

Podsumowując, specjalne tryby dopasowania pozwalają na pewną swobodę w zakresie tego, ile argumentów musimy przekazać do funkcji. Jeśli funkcja określa wartości domyślne, będą one wykorzystane, kiedy przekażemy *zbyt małą* liczbę argumentów. Jeśli funkcja wykorzystuje formę zmiennej listy argumentów z `*`, możemy przekazać *zbyt dużą* liczbę argumentów. Forma ta zbiera dodatkowe argumenty w strukturę danych w celu przetworzenia ich w funkcji.

Dopasowywanie argumentów — szczegóły

Jeśli zdecydujemy się użyć specjalnych trybów dopasowywania argumentów i połączyć je ze sobą, Python wymaga przestrzegania następujących reguł dotyczących kolejności.

- W wywołaniu funkcji najpierw muszą zostać podane wszystkie argumenty pozycyjne (*nazwa*), po których następują wszystkie argumenty ze słowami kluczowymi (*nazwa=wartość*), a po nich forma **sekwencja* i na końcu forma ***słownik*.
- W nagłówku funkcji argumenty muszą się pojawiać w następującej kolejności — normalne argumenty (*nazwa*), po nich argumenty z wartościami domyślnymi (*nazwa=wartość*), potem forma **nazwa* (lub * w wersji 3.0), jeśli jest ona obecna, następnie argumenty mogące być tylko słowami kluczowymi *nazwa* lub *nazwa=wartość* (z wersji 3.0), a na końcu ***nazwa*.

Zarówno w wywołaniach, jak i nagłówkach forma ***argumenty* (o ile jest obecna) musi się pojawiać na samym końcu. Jeśli pomieszamy argumenty w innej kolejności, otrzymamy błąd składni, ponieważ inne kombinacje mogą być niejednoznaczne. Python wewnętrznie wykonuje następujące kroki mające na celu dopasowanie argumentów przed przypisaniem:

1. Przypisuje argumenty niebędące słowami kluczowymi zgodnie z pozycją.
2. Przypisuje argumenty będące słowami kluczowymi poprzez dopasowanie nazw.
3. Przypisuje dodatkowe argumenty niebędące słowami kluczowymi do krotki **nazwa*.
4. Przypisuje dodatkowe argumenty będące słowami kluczowymi do słownika ***nazwa*.
5. Przypisuje do argumentów nieprzypisanych wartości domyślne z nagłówka.

Później Python sprawdza, czy do każdego argumentu przypisano dokładnie jedną wartość. Jeśli tak nie jest, zgłaszany jest błąd. Po zakończeniu dopasowywania Python przypisuje nazwy argumentów do obiektów do nich przekazanych.

Sam algorytm dopasowywania wykorzystywany w Pythonie jest nieco bardziej skomplikowany (musi również wziąć pod uwagę na przykład argumenty będące słowami kluczowymi z wersji 3.0), dlatego po dokładny opis odsyłam do dokumentacji języka. Nie jest to lektura obowiązkowa, jednak prześledzenie mechanizmu dopasowania Pythona pozwoli nam lepiej zrozumieć bardziej zawiłe przypadki, w szczególności takie z mieszanymi trybami.



W Pythonie 3.0 nazwy argumentów w nagłówku funkcji mogą także mieć wartości *adnotacji*, podane w formie *nazwa:wartość* (lub *nazwa:wartość=domyślna*, kiedy obecne są wartości domyślne). Jest to po prostu dodatkowa składnia przeznaczona dla argumentów, która nie rozszerza ani nie zmienia opisanych powyżej reguł kolejności argumentów. Sama funkcja także może mieć wartość adnotacji, podaną w postaci *f()>wartość*. Omówienie adnotacji funkcji można znaleźć w rozdziale 19.

Przykłady ze słowami kluczowymi i wartościami domyślnymi

Wszystko to w kodzie jest prostsze, niż mógłby sugerować powyższy opis. Jeśli nie korzystamy ze specjalnej składni dopasowania, Python dopasowuje nazwy po pozycji, od lewej do prawej strony, tak jak większość pozostałych języków programowania. Jeśli na przykład zdefiniujemy funkcję wymagającą trzech argumentów, musimy wywołać ją z trzema argumentami.

```
>>> def f(a, b, c): print(a, b, c)
...
```

Tutaj przekazujemy je po pozycji — a dopasowywane jest do 1, b do 2, natomiast c do 3 (działa to tak samo w Pythonie 3.0 i 2.6, jednak w wersji 2.6 wyświetlane są dodatkowe nawiasy krotki, ponieważ korzystamy z wywołania `print` z 3.0).

```
>>> f(1, 2, 3)
1 2 3
```

Słowa kluczowe

W Pythonie można jednak bardziej dokładnie określić, co ma gdzie trafić, kiedy wywoujemy funkcję. Argumenty ze słowami kluczowymi pozwalają na dopasowanie po *nazwie*, a nie pozycji.

```
>>> f(c=3, b=2, a=1)
1 2 3
```

Element `c=3` w tym wywołaniu oznacza na przykład przesyłanie wartości 3 do argumentu o nazwie `c`. Z bardziej formalnego punktu widzenia Python dopasowuje nazwę `c` w wywołaniu do argumentu o nazwie `c` w nagłówku definicji funkcji, a następnie przekazuje wartość 3 do tego argumentu. Rezultat tego wywołania będzie taki, jak poprzedniego, jednak warto zwrócić uwagę na to, że uporządkowanie argumentów od lewej do prawej strony w przypadku użycia słów kluczowych nie ma znaczenia — argumenty dopasowywane są po nazwie, a nie pozycji. Można nawet połączyć argumenty pozycyjne i słowa kluczowe w jednym wywołaniu. W takim przypadku wszystkie argumenty pozycyjne dopasowywane są od lewej do prawej strony nagłówka, zanim słowa kluczowe zostaną dopasowane po nazwach.

```
>>> f(1, c=3, b=2)
1 2 3
```

Kiedy większość osób pierwszy raz widzi taki zapis, zastanawia się, do czego można go użyć. Słowa kluczowe pełnią w Pythonie dwie role. Po pierwsze, sprawiają, że wywołania są nieco lepiej opisane (zakładając, że w kodzie funkcji użyliśmy bardziej znaczących nazw od `a`, `b` i `c`). Na przykład wywołanie w tej formie:

```
func(name='Teofil', age=40, job='programista')
```

jest bardziej znaczące od wywołania z trzema górnymi wartościami rozdzielonymi przecinkami — słowa kluczowe służą jako podpisy do danych z wywołania. Druga ważna dziedzina zastosowania słów kluczowych pojawia się w połączeniu z wartościami domyślnymi, do czego za chwilę dojdziemy.

Wartości domyślne

Mówiliśmy już nieco o wartościach domyślnych, kiedy omawialiśmy zakres funkcji zagnieżdzonych. W skrócie, wartości domyślne pozwalają nam uczynić wybrane argumenty funkcji opcjonalnymi. Jeśli do argumentu takiego nie przekaże się wartości, przed wykonaniem funkcji zostaje do niego przypisana wartość domyślna. Poniżej znajduje się przykład funkcji wymagającej jednego argumentu z dwoma wartościami domyślnymi.

```
>>> def f(a, b=2, c=3): print(a, b, c)
...

```

Kiedy wywołujemy tę funkcję, musimy podać wartość zmiennej `a` albo po pozycji, albo za pomocą słowa kluczowego. Podanie wartości dla zmiennych `b` oraz `c` jest opcjonalne. Jeśli nie przekażemy wartości do tych dwóch zmiennych, będą one miały wartości, odpowiednio, 2 i 3.

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
```

Jeśli przekażemy dwie wartości, jedynie c otrzymuje swoją wartość domyślną. W przypadku podania trzech wartości nie zostaną użyte żadne wartości domyślne.

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Wreszcie poniżej widać sposób interakcji pomiędzy słowami kluczowymi a wartościami domyślnymi. Ponieważ słowa kluczowe odwracają normalne odwzorowanie oparte na pozycji od lewej do prawej strony, pozwalają one na przeskoczenie argumentów z wartościami domyślnymi.

```
>>> f(1, c=6)
1 2 6
```

W powyższym kodzie zmienna a otrzymuje wartość 1 zgodnie z pozycją, zmienna c otrzymuje wartość 6 ze względu na użycie słowa kluczowego, natomiast znajdująca się pomiędzy nimi zmienna b ma wartość domyślną równą 2.

Należy uważać, by nie pomylić specjalnej składni *nazwa=wartość* w nagłówku funkcji i wywołaniu funkcji. W wywołaniu oznacza ona argument słowa kluczowego dopasowanego po nazwie, natomiast w nagłówku określa wartość domyślną opcjonalnego argumentu. W obu przypadkach nie jest to instrukcja przypisania (pomimo że tak wygląda), a jedynie specjalna składnia dla tych dwóch kontekstów, modyfikująca domyślny mechanizm dopasowywania argumentów.

Łączenie słów kluczowych i wartości domyślnych

Poniżej znajduje się nieco większy przykład demonstrujący działanie słów kluczowych oraz wartości domyślnych. W poniższym kodzie obiekt wywołujący musi zawsze przekazać co najmniej dwa argumenty (by pasowały one do zmiennych spam i eggs), jednak dwa pozostałe są opcjonalne. Jeśli zostają pominięte, Python przypisuje do argumentów toast i ham wartości domyślne podane w nagłówku.

```
def func(spam, eggs, toast=0, ham=0):          # Pierwsze dwa wymagane
    print((spam, eggs, toast, ham))

func(1, 2)                                      # Wynik: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                          # Wynik: (1, 0, 0, 1)
func(spam=1, eggs=0)                           # Wynik: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)                  # Wynik: (3, 2, 1, 0)
func(1, 2, 3, 4)                                # Wynik: (1, 2, 3, 4)
```

Warto ponownie zwrócić uwagę na to, że kiedy w wywołaniu wykorzystane są argumenty ze słowami kluczowymi, kolejność wymienionych argumentów nie ma znaczenia. Python dopasowuje je po nazwie, a nie pozycji. Kod wywołujący musi podać wartości dla argumentów spam oraz eggs, jednak mogą one zostać dopasowane albo po pozycji, albo po nazwie. Warto również pamiętać, że forma *nazwa=wartość* oznacza co innego w wywołaniu i w instrukcji def (w wywołaniu słowo kluczowe, a w nagłówku funkcji — wartość domyślną).

Przykład dowolnych argumentów

Ostatnie dwa rozszerzenia dopasowania, * oraz **, zaprojektowane zostały z myślą o obsłudze funkcji, które przyjmują dowolną liczbę argumentów. Oba mogą się pojawiać albo w definicji funkcji, albo w jej wywołaniu i mają w tych lokalizacjach powiązane cele.

Zbieranie argumentów

Pierwsze zastosowanie — w definicji funkcji — zbiera niedopasowane argumenty *pozycyjne* w krotkę.

```
>>> def f(*args): print(args)
...
...
```

Kiedy funkcja zostaje wywołana, Python zbiera wszystkie argumenty w nową krotkę i przypisuje zmienną `args` do tej krotki. Ponieważ jest to normalny obiekt krotki, można go indeksować czy na przykład przechodzić za pomocą pętli `for`.

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1,2,3,4)
(1, 2, 3, 4)
```

Opcja z `**` jest podobna, jednak działa tylko dla argumentów ze *słowami kluczowymi* — zbiera je w nowy słownik, który może następnie zostać przetworzony za pomocą normalnych narzędzi słowników. W pewnym sensie forma z `**` pozwala na konwersję ze słów kluczowych na słowniki, które można następnie przechodzić na przykład za pomocą wywołań `keys` czy iteratorów słowników.

```
>>> def f(**args): print(args)
...
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Wreszcie nagłówki funkcji mogą łączyć normalne argumenty z `*` oraz `**` w celu zaimplementowania przedziwnie elastycznych sygnatur wywołań. Przykładowo w poniższym kodzie 1 przekazywane jest do `a` po pozycji, 2 i 3 zbierane są w krotkę pozycyjną `pargs`, natomiast `x` i `y` trafiają do słownika słów kluczowych `kargs`:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
...
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

Tak naprawdę opcje te można łączyć na jeszcze bardziej skomplikowane sposoby, które mogą się na pierwszy rzut oka wydawać niejednoznaczne — zajmiemy się tym w dalszej części niniejszego rozdziału. Najpierw jednak zobaczymy, co stanie się, kiedy `*` oraz `**` umieszczone są w wywołaniach funkcji, a nie ich definicjach.

Rozpakowywanie argumentów

W nowszych wersjach Pythona składni z `*` można również użyć przy wywoływaniu funkcji. W tym kontekście jej znaczenie jest odwrotnością znaczenia z definicji funkcji — rozpakowuje kolekcję argumentów, zamiast ją budować. Możemy na przykład przekazać do funkcji cztery argumenty w krotce i pozwolić Pythonowi na rozpakowanie krotki na pojedyncze argumenty.

```
>>> def func(a, b, c, d): print(a, b, c, d)
...
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)
1 2 3 4
```

W podobny sposób składnia z `**` w wywołaniu funkcji rozpakowuje słownik par klucz-wartość na pojedyncze argumenty ze słowami kluczowymi.

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)
1 2 3 4
```

I znów w wywołaniu możemy na wiele elastycznych sposobów połączyć normalne argumenty pozycyjne oraz argumenty ze słowami kluczowymi.

```
>>> func(*(1, 2), **{'d': 4, 'c': 4})
1 2 4 4

>>> func(1, *(2, 3), **{'d': 4})
1 2 3 4

>>> func(1, c=3, *(2,), **{'d': 4})
1 2 3 4

>>> func(1, *(2, 3), d=4)
1 2 3 4

>>> f(1, *(2,), c=3, **{'d':4})
1 2 3 4
```

Ten rodzaj kodu jest wygodny, kiedy nie możemy w momencie pisania skryptu przewidzieć liczb argumentów, które będą przekazane do funkcji. Możemy zamiast tego zbudować kolekcję argumentów w czasie wykonania i w ten sposób wywołać funkcję w sposób uniwersalny. Ponownie nie należy mylić składni z `*` oraz `**` w nagłówku funkcji i jej wywołaniu. W nagłówku służy do zebrania dowolnej liczby argumentów, natomiast w wywołaniu rozpakowuje dowolną liczbę argumentów.



Jak widzieliśmy w rozdziale 14., forma `*pargs` w wywołaniu jest *kontekstem iteracyjnym*, dlatego przyjmuje ona dowolny obiekt, na którym można wykonywać iterację, a nie tylko krotki i inne sekwencje, zgodnie z przykładami zaprezentowanymi wyżej. Po znaku `*` można na przykład zastosować obiekt pliku, co powoduje rozpakowanie jego wierszy do pojedynczych argumentów (na przykład `func(*open('plik'))`).

Taka uniwersalność obsługiwana jest zarówno w Pythonie 3.0, jak i 2.6, jednak jest prawdziwa tylko w przypadku *wywołań*. Forma `*pargs` w wywołaniu pozwala na zastosowanie dowolnego obiektu, na którym można wykonywać iterację, jednak ta sama forma w nagłówku instrukcji `def` zawsze zbiera dodatkowe argumenty w *krotce*. Takie zachowanie nagłówka jest podobne w duchu i składni do `*` w rozszerzonych formach przypisów rozpakowujących sekwencje z Pythona 3.0, omówionych w rozdziale 11. (na przykład `x, *y = z`), choć opcja ta zawsze tworzy listy, a nie krotki.

Uniwersalne stosowanie funkcji

Przykłady z poprzedniego podrozdziału mogą się wydawać nieco nieciekawe, jednak wykorzystywane są częściej, niż można by się tego spodziewać. Niektóre programy muszą wywoływać różne funkcje w sposób uniwersalny, nie znając ich nazw czy argumentów z wyprzedzeniem. Tak naprawdę prawdziwa siła specjalnej składni wywołania z dowolnymi argumentami tkwi w braku konieczności posiadania przed napisaniem skryptu wiedzy, ile argumentów wymaga wywołanie funkcji. Przykładowo możemy wykorzystać logikę `if` do wybrania ze zbioru funkcji i list argumentów, a następnie wywołania ich w sposób uniwersalny:

```

if <test>:
    action, args = func1, (1,)           # W tym przypadku wywołanie func1 z 1 argumentem
else:
    action, args = func2, (1, 2, 3)     # Tutaj wywołanie func2 z 3 argumentami
...
action(*args)                          # Uniwersalne wywołanie

```

Mówiąc bardziej ogólnie, składnia wywołań z dowolną liczbą argumentów przydatna jest zawsze wtedy, gdy nie możemy przewidzieć listy argumentów. Jeśli użytkownik wybierze na przykład dowolną funkcję za pośrednictwem interfejsu użytkownika, niemożliwe może się okazać napisanie wywołania funkcji w czasie tworzenia skryptu. By obejść to ograniczenie, wystarczy zbudować listę argumentów za pomocą działań na sekwencjach i wywołać ją z nazwami ze znakami * w celu rozpakowania argumentów.

```

>>> args = (2,3)
>>> args += (4,)
>>> args
(2, 3, 4)
>>> func(*args)

```

Ponieważ lista argumentów przekazywana jest tutaj jako krotka, program może budować ją w czasie wykonania. Technika ta przydaje się także w przypadku funkcji, które sprawdzają lub mierzą czas innych funkcji. Przykładowo w poniższym kodzie obsługujemy dowolną funkcję z dowolną liczbą argumentów, przekazując dowolne argumenty, jakie zostały przesłane:

```

def tracer(func, *pargs, **kargs):          # Przyjmuje dowolne argumenty
    print('wywoływanie:', func.__name__)
    return func(*pargs, **kargs)              # Przekazuje dowolne argumenty

def func(a, b, c, d):
    return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))

```

Po wykonaniu powyższego kodu argumenty zbierane są przez funkcję `tracer`, a następnie *propagowane* dalej za pomocą składni wywołania z dowolną liczbą argumentów:

```

wywoływanie: func
10

```

Większe przykłady takich ról zobaczymy w dalszej części książki. Warto zwrócić uwagę zwłaszcza na przykład z poniarem sekwencji z rozdziału 20. oraz różne narzędzia dekoratorów omówione w rozdziale 38.

Zlikwidowana funkcja wbudowana apply (Python 2.6)

Przed Pythonem 3.0 efekt działania składni z dowolną liczbą argumentów `*argumenty` oraz `**argumenty` można było uzyskać za pomocą funkcji wbudowanej o nazwie `apply`. Technika ta została usunięta z wersji 3.0, ponieważ jest teraz zbędna (w wersji tej usunięto wiele zakurzonych narzędzi, które nagromadziły się przez lata). Jest ona jednak nadal dostępna w Pythonie 2.6 i można na nią natrafić w starszym kodzie napisanym w wersjach 2.X.

Mówiąc w skrócie, poniższe wiersze są równoważne w Pythonie przed wersją 3.0:

```

func(*pargs, **kargs)                      # Nowsza składnia wywołania: func(*sekwencja, **słownik)
apply(func, pargs, kargs)                   # Zlikwidowana funkcja wbudowana: apply(func, sekwencja, słownik)

```

Przykładowo zastanówmy się nad poniższą funkcją, która przyjmuje dowolną liczbę argumentów pozycyjnych lub będących słowami kluczowymi:

```
>>> def echo(*args, **kwargs): print(args, kwargs)
...
>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

W Pythonie 2.6 możemy wywołać ją w sposób uniwersalny za pomocą apply lub z użyciem składni wywołania wymaganej obecnie w wersji 3.0:

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}

>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}

>>> echo(*pargs, **kargs)
(1, 2) {'a': 3, 'b': 4}
```

Składnia wywołania z rozpakowaniem jest nowsza od funkcji apply, ogólnie zalecana i wymagana w wersji 3.0. Poza symetrią z formami kolektorów *pargs i **kargs z nagłówków instrukcji def oraz faktem, iż wymaga mniej pisania, nowsza składnia wywołania pozwala także na przekazywanie dodatkowych argumentów bez konieczności ręcznego rozszerzania sekwencji bądź słowników argumentów.

```
>>> echo(0, c=5, *pargs, **kargs) # Normalny, słowo kluczowe, *sekwencja, **słownik
(0, 1, 2) {'a': 3, 'c': 5, 'b': 4}
```

Oznacza to, że składnia wywołania jest *bardziej uniwersalna*. Ponieważ jest też wymagana w wersji 3.0, teraz należy wyrzucić z głowy całą swoją wiedzę na temat funkcji apply (o ile oczywiście nie pojawiła się ona w kodzie Pythona 2.X, z którego musimy korzystać lub który musimy utrzymywać...).

Argumenty mogące być tylko słowami kluczowymi z Pythona 3.0

W Pythonie 3.0 uogólniono reguły kolejności w nagłówkach funkcji w taki sposób, by możliwe było podawanie *argumentów mogących być tylko słowami kluczowymi* (ang. *keyword-only argument*) — czyli argumentów, które trzeba przekazywać za pomocą słowa kluczowego i które nigdy nie zostaną wypełnione przez jakikolwiek argument pozycyjny. Jest to przydatne, jeśli chcemy, by funkcja zarówno przetwarzała dowolną liczbę argumentów, jak i przyjmowała możliwie opcjonalne opcje konfiguracyjne.

Z punktu widzenia składni argumenty tego typu zapisywane są w kodzie jako nazwane argumenty, które na liście argumentów pojawiają się po formie **argumenty*. Wszystkie muszą być przekazywane za pomocą składni ze słowami kluczowymi w wywołaniu. Przykładowo w poniższym kodzie a można przekazać za pomocą nazwy lub pozycji, b zbiera wszystkie dodatkowe argumenty pozycyjne, natomiast c można przekazać jedynie przez słowo kluczowe:

```
>>> def kwonly(a, *b, c):
...     print(a, b, c)
...
>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() needs keyword-only argument c
```

Możemy również wykorzystać sam znak * w liście argumentów w celu wskazania, że funkcja nie przyjmuje listy argumentów o zmiennej długości, natomiast nadal oczekuje, że wszystkie argumenty pojawiające się po * zostaną przekazane jako słowa kluczowe. W poniższej funkcji a można znowu przekazać za pomocą nazwy lub pozycji, natomiast b oraz c muszą być słowami kluczowymi i nie są dozwolone żadne dodatkowe argumenty pozycyjne:

```
>>> def kwonly(a, *, b, c):
...     print(a, b, c)
...
>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
>>> kwonly(1)
TypeError: kwonly() needs keyword-only argument b
```

W przypadku argumentów mogących być tylko słowami kluczowymi nadal możemy wykorzystać wartości domyślne, choć pojawiają się one po znaku * w nagłówku funkcji. W poniższym kodzie a można przekazać za pomocą nazwy lub pozycji, natomiast b oraz c są opcjonalne, jednak jeśli zostaną użyte, muszą być przekazane za pomocą słowa kluczowego:

```
>>> def kwonly(a, *, b='mielonka', c='szynka'):
...     print(a, b, c)
...
>>> kwonly(1)
1 mielonka szynka
>>> kwonly(1, c=3)
1 mielonka 3
>>> kwonly(a=1)
1 mielonka szynka
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)
```

Tak naprawdę argumenty mogące być tylko słowami kluczowymi z wartościami domyślnymi są opcjonalne, jednak te bez wartości domyślnych stają się wymaganymi słowami kluczowymi dla funkcji:

```
>>> def kwonly(a, *, b, c='mielonka'):
...     print(a, b, c)
...
>>> kwonly(1, b='jajka')
1 jajka mielonka
>>> kwonly(1, c='jajka')
TypeError: kwonly() needs keyword-only argument b
>>> kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)

>>> def kwonly(a, *, b=1, c, d=2):
...     print(a, b, c, d)
...
>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() needs keyword-only argument c
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
```

Reguły dotyczące kolejności

Wreszcie warto podkreślić, że argumenty mogące być tylko słowami kluczowymi muszą być podawane po pojedynczym znaku *, a nie dwóch — po formie z dowolnymi słowami kluczowymi **argumenty nie mogą się pojawiać nazwane argumenty. Znaki ** nie mogą też pojawiać się samodzielnie w liście argumentów. Oba zapisy generują błąd składni:

```
>>> def kwonly(a, **pargs, b, c):  
SyntaxError: invalid syntax  
>>> def kwonly(a, **, b, c):  
SyntaxError: invalid syntax
```

Oznacza to, że w *nagłówku* funkcji argumenty mogące być tylko słowami kluczowymi muszą być zapisywane przed formą z dowolnymi słowami kluczowymi **argumenty, a po formie *argumenty z dowolnymi argumentami pozycyjnymi — o ile obie są obecne. Za każdym razem, gdy przed *argumenty pojawia się nazwa argumentu, będzie to raczej domyślny argument pozycyjny, a nie argument mogący być tylko słowem kluczowym:

```
>>> def f(a, *b, **d, c=6): print(a, b, c, d) # Przed ** argumenty mogące być tylko słowami  
# kluczowymi!  
SyntaxError: invalid syntax  
  
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # Zebranie argumentów w nagłówku  
...  
>>> f(1, 2, 3, x=4, y=5) # Użyte wartości domyślne  
1 (2, 3) 6 {'y': 5, 'x': 4}  
  
>>> f(1, 2, 3, x=4, y=5, c=7) # Nadpisanie wartości domyślnych  
1 (2, 3) 7 {'y': 5, 'x': 4}  
  
>>> f(1, 2, 3, c=7, x=4, y=5) # W dowolnym miejscu w słowach kluczowych  
1 (2, 3) 7 {'y': 5, 'x': 4}  
  
>>> def f(a, c=6, *b, **d): print(a, b, c, d) # c nie jest argumentem mogącym być tylko słowem  
# kluczowym!  
...  
>>> f(1, 2, 3, x=4)  
1 (3,) 2 {'x': 4}
```

Tak naprawdę podobne reguły kolejności są prawdziwe dla *wywołań* funkcji. Kiedy przekazywane są argumenty mogące być tylko słowami kluczowymi, muszą się one pojawiać przed formą **argumenty. Argument mogący być tylko słowem kluczowym może jednak być zapisany albo przed formą *argumenty, albo po niej, i może się mieścić w formie **argumenty:

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d) # Pomiedzy * a ** tylko słowa kluczowe  
...  
>>> f(1, *(2, 3), **dict(x=4, y=5)) # Rozpakowanie argumentów przy wywoaniu  
1 (2, 3) 6 {'y': 5, 'x': 4}  
  
>>> f(1, *(2, 3), **dict(x=4, y=5), c=7) # Słowa kluczowe przed **argumenty!  
SyntaxError: invalid syntax  
  
>>> f(1, *(2, 3), c=7, **dict(x=4, y=5)) # Nadpisanie wartości domyślnych  
1 (2, 3) 7 {'y': 5, 'x': 4}  
  
>>> f(1, c=7, *(2, 3), **dict(x=4, y=5)) # Przed lub po *  
1 (2, 3) 7 {'y': 5, 'x': 4}  
  
>>> f(1, *(2, 3), **dict(x=4, y=5, c=7)) # Argument mogący być tylko słowem kluczowym w **  
1 (2, 3) 7 {'y': 5, 'x': 4}
```

Czemu służą argumenty mogące być tylko słowami kluczowymi?

Po co nam więc argumenty mogące być tylko słowami kluczowymi? W skrócie, umożliwiają one funkcji przyjmowanie zarówno dowolnej liczby argumentów pozycyjnych do przetwarzania, jak i przekazywanie opcji konfiguracyjnych w postaci słów kluczowych. Choć ich stosowanie jest opcjonalne, bez argumentów mogących być tylko słowami kluczowymi podanie wartości domyślnych dla takich opcji i upewnienie się, że nie przekazano żadnych nadmiarowych słów kluczowych, może wymagać dodatkowej pracy.

Wyobraźmy sobie funkcję przetwarzającą zbiór przekazanych obiektów i pozwalającą na przekazanie opcji śledzącej:

```
process(X, Y, Z)                                # Wykorzystanie wartości domyślnej opcji  
process(X, Y, notify=True)                      # Nadpisanie wartości domyślnej opcji
```

Bez argumentów mogących być tylko słowami kluczowymi musimy użyć zarówno formy `*argumenty`, jak i `**argumenty` i ręcznie badać słowa kluczowe, natomiast po wykorzystaniu rozwiązania z argumentami mającymi być tylko słowami kluczowymi wymagana jest mniejsza ilość kodu. Poniższy kod gwarantuje, że żadne argumenty pozycyjne nie zostaną błędnie dopasowane do `notify`, i wymaga, by przy przekazaniu było to słowo kluczowe:

```
def process(*args, notify=False): ...
```

Ponieważ bardziej realistyczne przykłady tej techniki zobaczymy w dalszej części rozdziału, w punkcie „Emulacja funkcji print z Pythona 3.0”, resztę informacji zamieścimy właśnie tam. Dodatkowy przykład działania argumentów mogących być tylko słowami kluczowymi znajduje się w studium przypadku z poniarem opcji iteracji w rozdziale 20. Dodatkowe ulepszenia definicji funkcji w Pythonie 3.0 znajdują się przy omówieniu składni adnotacji funkcji w rozdziale 19.

Przykład z funkcją obliczającą minimum

Czas na coś bardziej realistycznego. By jakoś skonkretyzować koncepcje przedstawione w niniejszym rozdziale, wykonajmy ćwiczenie demonstrujące praktyczne zastosowanie narzędzi dopasowujących argumenty.

Załóżmy, że chcemy utworzyć kod funkcji potrafiącej obliczyć wartość minimalną dla dowolnego zbioru argumentów i dowolnego zbioru typów danych obiektów. Funkcja ta powinna zatem przyjmować zero lub większą liczbę argumentów — tyle, ile będziemy jej chcieli przekazać. Co więcej, funkcja ta powinna działać na wszystkich rodzajach typów obiektów Pythona: liczbach, łańcuchach znaków, listach, listach słowników, plikach, a nawet `None`.

Pierwsze wymaganie staje się naturalnym polem do zastosowania opcji z `*` — możemy zebrać argumenty w krotkę i przejść każdy z nich po kolej za pomocą prostej pętli `for`. Druga część definicji problemu jest prosta: ponieważ każdy typ obiektu obsługuje porównania, nie musimy dostosowywać funkcji do specyficznych wymagań poszczególnych typów (zastosowanie polymorfizmu) — możemy po prostu ślepo zestawić obiekty i pozwolić Pythonowi na wykonanie odpowiedniego dla nich porównania.

Pełne rozwiązanie

Poniższy plik pokazuje trzy sposoby zapisania tej operacji w kodzie, z których co najmniej jeden został zasugerowany przez jednego z moich studentów.

- Pierwsza funkcja pobiera pierwszy argument (args jest krotką) i przechodzi resztę, odcinając pierwszy argument (nie ma sensu porównywać obiektu z samym sobą, w szczególności jeśli może on być większą strukturą).
- Druga wersja pozwala Pythonowi na automatyczne wybranie pierwszego i pozostałych argumentów, zatem unika indeksowania i wycinka.
- Trzecia metoda dokonuje konwersji krotki na listę za pomocą wywołania wbudowanej funkcji list i stosuje metodę listy sort.

Metoda sort napisana jest jako kod języka C, dlatego czasami może być szybsza od pozostałych rozwiązań, jednak w większości przypadków liniowe przeglądanie z dwóch pierwszych technik będzie działać szybciej.¹ Plik *mins.py* zawiera kod wszystkich trzech rozwiązań.

```
def min1(*args):  
    res = args[0]  
    for arg in args[1:]:  
        if arg < res:  
            res = arg  
    return res  
  
def min2(first, *rest):  
    for arg in rest:  
        if arg < first:  
            first = arg  
    return first  
  
def min3(*args):  
    tmp = list(args) # Lub w Pythonie 2.4+: return sorted(args)[0]  
    tmp.sort()  
    return tmp[0]  
  
print(min1(3,4,1,2))  
print(min2("bb", "aa"))  
print(min3([2,2], [1,1], [3,3]))
```

Wszystkie trzy rozwiązania dają po wykonaniu pliku ten sam wynik. Można spróbować wpisać kilka wywołań w sesji interaktywnej w celu samodzielnego eksperymentowania z trzema funkcjami.

```
% python mins.py  
1  
aa  
[1, 1]
```

¹ Tak naprawdę jest to dość skomplikowane. Procedura sort w Pythonie została napisana w języku C i wykorzystuje bardzo zoptymalizowany algorytm, który próbuje korzystać z częściowego uporządkowania elementów do sortowania. Nosi on nazwę *timsort* od imienia twórcy, Tim'a Petersa, i zgodnie z dokumentacją ma w niektórych sytuacjach mieć „ponadnaturalną wydajność” (całkiem nieźle, jak na sortowanie!). Mimo to sortowanie to operacja potencjalnie wykładnicza (musi wiele razy pociąć sekwencję na kawałki i ponownie ją złożyć), a inne wersje rozwiązania po prostu wykonują jedno liniowe przeglądnięcie kolekcji argumentów. W rezultacie sortowanie jest szybsze, jeśli argumenty są częściowo posortowane, jednak w innym przypadku prawdopodobnie będzie wolniejsze. Wydajność Pythona może jednak z czasem się zmieniać, a sam fakt, że sortowanie zaimplementowane jest w języku C, może znacznie pomóc. By dokonać dokładnej analizy, powinniśmy zmierzyć czas działania alternatywnych rozwiązań za pomocą modułów time i timeit omówionych w rozdziale 20.

Warto zauważyć, że żaden z trzech wariantów nie sprawdza przypadku, w którym nie przekazano żadnych argumentów. Mogłyby to robić, ale nie ma to tutaj sensu — we wszystkich trzech rozwiązaniach Python automatycznie zgłasza wyjątek, jeśli do funkcji nie przekazano żadnych argumentów. Pierwszy wariant zwraca wyjątek, kiedy próbujemy pobrać element zerowy, drugi — kiedy Python wykrywa niedopasowanie listy argumentów, natomiast trzeci — kiedy próbujemy na końcu zwrócić element zerowy.

Dokładnie o to nam chodzi — ponieważ funkcje obsługują dowolny typ danych, nie istnieje poprawna wartość ostrzegawcza, którą moglibyśmy przekazać w celu oznaczenia błędu. Istnieją wyjątki od tej reguły (na przykład jeśli musimy przed dojściem do błędu wykonać kosztowne działanie), jednak lepiej jest zakładać w kodzie funkcji, że argumenty będą działały, i pozwolić Pythonowi na zgłoszenie wyjątków, kiedy tak nie będzie.

Dodatkowy bonus

Punkty dodatkowe można uzyskać za modyfikację funkcji w taki sposób, by obliczały one wartości *maksymalne*, a nie minimalne. To proste zadanie — pierwsze dwie wersje wymagają jedynie zmiany < na >, natomiast trzecia wymaga zwrócenia `tmp[-1]` w miejsce `tmp[0]`. W celu uzyskania dodatkowych punktów należy również pamiętać o zmianie nazwy funkcji na `max` (choć jest to całkowicie opcjonalne).

Można również utworzyć jedną uogólnioną funkcję obliczającą wartość minimalną lub maksymalną poprzez obliczenie łańcuchów znaków wyrażenia porównania za pomocą narzędzi takich, jak wbudowana funkcja `eval` (więcej informacji w dokumentacji biblioteki standardowej) lub przekazania dowolnej funkcji porównującej. Plik `minmax.py` pokazuje, w jaki sposób można zaimplementować ten ostatni mechanizm.

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y                      # Zobacz również lambda
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Kod testu
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

% python minmax.py
1
6
```

Funkcje są kolejnym typem obiektu, jaki może zostać przekazany do funkcji, podobnie jak w kodzie wyżej. By zrobić z tego funkcję `max` (czy dowolną inną), moglibyśmy na przykład przekazać właściwy rodzaj funkcji `test`. Może się to wydawać dodatkową pracą, jednak najważniejszą zaletą uogólniania funkcji (w miejsce kopowania i wklejania w celu zmiany jednego znaku) jest to, że w przyszłości będziemy musieli modyfikować tylko jeden kod, a nie dwa.

Puenta

Wszystko to było oczywiście jedynie ćwiczeniem programistycznym. Tak naprawdę nie ma powodu tworzyć w Pythonie funkcji `min` czy `max`, ponieważ obie są wbudowane w sam język! Spotkaliśmy je w rozdziale 5. w połączeniu z narzędziami działającymi na liczbach, a także w rozdziale 14. przy omawianiu kontekstów iteracyjnych. Wersje wbudowane działają prawie dokładnie tak samo jak nasze, jednak w celu uzyskania optymalnej szybkości napisane zostały w języku C i przyjmują albo jeden, albo większą liczbę argumentów, na których można wykonywać iterację. Nawet jeśli w tym kontekście kod ten był nieco zbędny, sam wykorzystany tutaj wzorzec programowania może się przydać w innych scenariuszach.

Uogólnione funkcje działające na zbiorach

Przyjrzyjmy się teraz bardziej użytecznemu przykładowi działania specjalnych trybów dopasowywania argumentów. Na końcu rozdziału 16. napisaliśmy funkcję zwracającą część wspólną dwóch sekwencji (wybierała ona elementy pojawiające się w obu sekwencjach). Poniżej znajduje się jej wersja obliczająca część wspólną dla dowolnej liczby sekwencji (jednej lub większej ich liczby), wykorzystującą postać dopasowania ze zmienną liczbą argumentów (`*args`) do zebrania wszystkich przekazanych argumentów. Ponieważ argumenty trafiają do funkcji w postaci krotki, możemy je przetworzyć za pomocą prostej pętli `for`. Dla zabawy napiszemy funkcję obliczającą sumę zbiorów i również przyjmującą dowolną liczbę argumentów oraz zbiegającą elementy pojawiające się w dowolnym z argumentów.

```
def intersect(*args):
    res = []
    for x in args[0]:
        for other in args[1:]:
            if x not in other: break
            else:
                res.append(x)
    return res

def union(*args):
    res = []
    for seq in args:
        for x in seq:
            if not x in res:
                res.append(x)
    return res
```

Przejście pierwszej sekwencji
Dla wszystkich pozostałych argumentów
Element w każdym z nich?
Nie — wyjście z pętli
Tak — dodanie elementów na końcu

Dla wszystkich argumentów
Dla wszystkich węzłów
Dodanie nowych elementów do wyniku

Ponieważ narzędzia te mogą się przydać w przyszłości (a na dodatek są zbyt duże, by wpisywać je ponownie w sesji interaktywnej), zachowamy te funkcje w pliku modułu o nazwie `inter2.py` (osoby, które nie pamiętają, jak działa importowanie modułów, mogą wrócić do wprowadzenia w rozdziale 3. lub poczekać na więcej informacji na temat modułów w piątej części książki). W obu funkcjach argumenty przekazane w wywołaniu przychodzą jako krotka `args`. Jak oryginalna funkcja `intersect`, obie funkcje działają na dowolnym typie sekwencji. Poniżej widać, jak funkcje te przetwarzają łańcuchy znaków, typy mieszane oraz liczbę sekwencji większą od dwóch.

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "Teodor", "Teofil", "Troll"

>>> intersect(s1, s2), union(s1, s2)          # Dwa argumenty
```

```
(['T', 'e', 'o', 'o'], ['T', 'e', 'o', 'd', 'r', 'f', 'i', 'l'])

>>> intersect([1,2,3], (1,4))          # Typy mieszane
[1]

>>> intersect(s1, s2, s3)           # Trzy argumenty
['T', 'o', 'o']

>>> union(s1, s2, s3)
['T', 'e', 'o', 'd', 'r', 'f', 'i', 'l']
```



Powiniennem podkreślić, że ponieważ Python zawiera teraz *typ obiektu zbioru* (opisany w rozdziale 5.), żaden z przykładów przetwarzania zbiorów z niniejszej książki nie jest już wymagany. Są one dołączone do książki jako demonstracje technik tworzenia kodu. Na marginesie dodam, że ponieważ Python stale jest poprawiany, można powiedzieć, iż jego twórcy w osobliwy sposób knują mi za plecami, sprawiając, że przykłady z mojej książki stają się przestarzałe i niepotrzebne!

Emulacja funkcji print z Pythona 3.0

Kończąc niniejszy rozdział, przyjrzymy się ostatniemu przykładowi działania dopasowywania argumentów. Kod zaprezentowany niżej został utworzony z myślą o wykorzystaniu w Pythonie 2.6 oraz wersjach wcześniejszych (działa on również w wersji 3.0, jednak używanie go tam nie ma większego sensu). Wykorzystuje on krotkę z dowolnymi argumentami pozycyjnymi **argumenty* oraz słownik z dowolnymi argumentami-słowami kluczowymi ***argumenty* w celu wykonania symulacji tego, co robi funkcja *print* z Pythona 3.0.

Jak wiemy z rozdziału 11., nie jest to tak naprawdę wymagane, ponieważ programiści mogą w Pythonie 2.6 włączyć obsługę funkcji *print* z wersji 3.0 za pomocą instrukcji *import* o poniższej postaci:

```
from __future__ import print_function
```

W celu zademonstrowania dopasowywania argumentów w pliku *print30.py* wykonamy to samo działanie za pomocą niewielkiej ilości kodu, który można wykorzystać ponownie.

```
"""
Emulacja działania funkcji 'print' z Pythona 3.0 w celu wykorzystania jej w wersji 2.X
sygnatura wywołania: print30(*args, sep=' ', end='\n', file=None)
"""

import sys

def print30(*args, **kwargs):
    sep = kwargs.get('sep', ' ')                      # Wartości domyślne argumentów-słów kluczowych
    end = kwargs.get('end', '\n')
    file = kwargs.get('file', sys.stdout)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

W celu przetestowania działania tego kodu należy zainportować go do innego pliku bądź do sesji interaktywnej i użyć podobnie jak funkcji *print* z Pythona 3.0. Poniżej znajduje się skrypt testowy *testprint30.py* (warto pamiętać, że funkcja musi nosić nazwę *print30*, ponieważ *print* jest w wersji 2.6 słowem zarezerwowanym):

```

from print30 import print30
print30(1, 2, 3)
print30(1, 2, 3, sep='')
print30(1, 2, 3, sep='...')
print30(1, [2], (3,), sep='...')

print30(4, 5, 6, sep='', end=' ')
print30(7, 8, 9)
print30()

import sys
print30(1, 2, 3, sep='??', end='.\n', file=sys.stderr) # Przekierowanie do pliku

```

Po wykonaniu w wersji 2.6 otrzymujemy te same wyniki co dla funkcji print Pythona 3.0:

```

C:\misc> c:\python26\python testprint30.py
1 2 3
123
1...2...3
1...[2]...(3,)
4567 8 9

1?????3.

```

Choć nie ma to większego znaczenia, po wykonaniu kodu w Pythonie 3.0 wyniki są takie same. Jak zawsze uniwersalność projektu Pythona pozwala nam na tworzenie prototypów i rozwijanie koncepcji w samym języku Python. W tym przypadku narzędzia służące do dopasowywania argumentów są w kodzie napisanym w Pythonie tak samo elastyczne jak w wewnętrznej implementacji tego języka.

Wykorzystywanie argumentów mogących być tylko słowami kluczowymi

Ciekawostką jest, że powyższy przykład można utworzyć z wykorzystaniem opisanych wcześniej w niniejszym rozdziale argumentów mogących być tylko słowami kluczowymi z Pythona 3.0, służących do automatycznego sprawdzania poprawności argumentów konfiguracyjnych.

Wykorzystanie argumentów mogących być tylko słowami kluczowymi

```

def print30(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)

```

Ta wersja kodu działa tak samo jak pierwsza i jest świetnym przykładem użyteczności argumentów mogących być tylko słowami kluczowymi. W oryginalnej wersji kodu zakładamy, że wszystkie argumenty pozycyjne mają być wyświetlane, natomiast wszystkie argumenty ze słowami kluczowymi służą do podania opcji. To właściwie wystarczy, jednak wszelkie dodatkowe argumenty-słowa kluczowe będą po cichu ignorowane. Wywołanie podobne do poniższego wygeneruje na przykład wyjątek w przypadku zastosowania formy z argumentem mogącym być tylko słowem kluczowym:

```

>>> print30(99, name='robert')
TypeError: print30() got an unexpected keyword argument 'name'

```

natomiast w oryginalnej wersji argument `name` zostanie po cichu zignorowany. W celu ręcznego wykrywania zbędnych słów kluczowych moglibyśmy użyć `dict.pop()` i usunąć pobrane wpisy, a następnie sprawdzić, czy słownik nie jest pusty. Poniżej znajduje się odpowiednik wersji z argumentami mogącymi być tylko słowami kluczowymi:

```
# Zastosowanie usunięcia argumentów-słów kluczowych z wartościami domyślnymi

def print30(*args, **kargs):
    sep = kargs.pop('sep', ' ')
    end = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('dodatkowe słowa kluczowe: %s' % kargs)
    output = ''
    first = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

Powyższy kod działa jak wersja poprzednia, jednak przechwytuje teraz również dodatkowe argumenty będące słowami kluczowymi:

```
>>> print30(99, name='robert')
TypeError: dodatkowe słowa kluczowe: {'name': 'robert'}
```

Ta wersja funkcji działa w Pythonie 2.6, jednak wymaga o cztery wiersze kodu więcej od wersji z argumentami mogącymi być tylko słowami kluczowymi. Niestety, dodatkowe wiersze kodu są w tym przypadku niezbędne — wersja z argumentami mogącymi być tylko słowami kluczowymi działa jedynie w Pythonie 3.0, co stoi w sprzeczności z powodami tworzenia tego przykładu (emulator wersji 3.0 działający jedynie w wersji 3.0 nie jest szczególnie przydatny!). W programach pisanych z myślą o Pythonie 3.0 argumenty mogące być jedynie słowami kluczowymi mogą uprościć pewną kategorię funkcji, które przyjmują zarówno argumenty, jak i opcje. Kolejny przykład działania argumentów mogących być jedynie słowami kluczowymi znajduje się w rozdziale 20. w studium przypadku pomiaru czasu wykonywania iteracji.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy drugą kluczową koncepcję związaną z funkcjami — *argumenty* (sposoby przekazywania obiektów do funkcji). Jak już wiemy, argumenty przekazywane są do funkcji przez przypisanie, co oznacza referencję do obiektu, a tak naprawdę wskaźnik. Omówiliśmy także bardziej zaawansowane rozszerzenia — na przykład argumenty ze słowami kluczowymi oraz wartościami domyślnymi, narzędzia służące do wykorzystywania dowolnej liczby argumentów, a także argumenty mogące być tylko słowami kluczowymi z wersji 3.0. Wreszcie widzieliśmy, jak zmienne argumenty mogą przejawiać takie samo zachowanie jak inne współdzielone referencje do obiektów — o ile obiekt nie jest kopowany w sposób jawnym przy przesyłaniu, modyfikacja przekazanego obiektu zmiennego może wpływać na kod wywołujący.

W kolejnym rozdziale kontynuujemy omówienie funkcji, zgłębiając bardziej zaawansowane zagadnienia — adnotacje funkcji, wyrażenia `lambda` oraz narzędzia funkcjonalne, takie jak `map` i `filter`. Wiele z tych koncepcji wynika z faktu, że funkcje są w Pythonie normalnymi obiektami, dlatego obsługują pewne zaawansowane i bardzo elastyczne tryby przetwarzania. Przed przejściem do tych zagadnień czas zająć się quizem sprawdzającym związane z argumentami kwestie omówione w niniejszym rozdziale.

Znaczenie argumentów ze słowami kluczowymi

Jak widać, zaawansowane tryby dopasowywania argumentów mogą być skomplikowane. Są również całkowicie opcjonalne. Można sobie zupełnie dobrze radzić, stosując proste dopasowanie pozycyjne, i naprawdę niezwykle na początku jest to niezły pomysł. Ponieważ jednak niektóre narzędzia Pythona z nich korzystają, dobrze jest mieć choć ogólną wiedzę na temat trybów specjalnych.

Argumenty ze słowami kluczowymi odgrywają na przykład szczególnie istotną rolę w interfejsie `tkinter`, będącym de facto standardowym API graficznego interfejsu użytkownika w Pythonie (w Pythonie 2.6 moduł ten nosi nazwę `Tkinter`). W różnych miejscach książki krótko wspominamy o module `tkinter`; jeśli chodzi o jego wzorce wywoływanego, argumenty ze słowami kluczowymi ustawiają opcje konfiguracyjne przy budowaniu komponentów GUI. Przykładowo wywołanie w formie:

```
from tkinter import *
widget = Button(text="Naciśnij mnie", command=someFunction)
```

tworzy nowy przycisk i określa jego tekst oraz funkcję zwrotną, używając do tego argumentów ze słowami kluczowymi `text` oraz `command`. Ponieważ liczba opcji konfiguracji dla widgetu może być spora, argumenty ze słowami kluczowymi pozwalają nam wybierać z nich odpowiednie. Bez tego albo trzeba wymienić listę wszystkich możliwych opcji zgodnie z ich pozycją, albo liczyć na rozsądny protokół pozycyjnych wartości domyślnych, który poradzi sobie z każdym możliwym ułożeniem opcji.

Wiele funkcji wbudowanych Pythona oczekuje, że w różnych trybach użycia przekażemy im słowa kluczowe jako opcje, które mogą mieć wartości domyślne, ale nie muszą. Jak wiemy z rozdziału 8., funkcja wbudowana `sorted` w postaci:

```
sorted(iterable, key=None, reverse=False)
```

oczekuje przekazania do sortowania obiektu, na którym można wykonać iterację, jednak pozwala nam także przekazać opcjonalne argumenty-słowa kluczowe w celu wybrania klucza sortowania słownika oraz opcji odwrócenia kierunku sortowania — domyślnie mają one wartości, odpowiednio, `None` oraz `False`. Ponieważ normalnie nie korzystamy z tych opcji, można je pominąć w celu zastosowania wartości domyślnych.

Sprawdź swoją wiedzę — quiz

1. Jaki będzie wynik poniższego kodu i dlaczego?

```
>>> def func(a, b=4, c=5):
...     print(a, b, c)
...
>>> func(1, 2)
```

2. Jaki będzie wynik poniższego kodu i dlaczego?

```
>>> def func(a, b, c=5):
...     print(a, b, c)
...
>>> func(1, c=3, b=2)
```

3. Jakie dane wyjściowe zwraca poniższy kod i ponownie: dlaczego?

```
>>> def func(a, *pargs):
...     print(a, pargs)
...
>>> func(1, 2, 3)
```

4. Co wyświetla poniższy kod i dlaczego?

```
>>> def func(a, **kargs):
...     print(a, kargs)
...
>>> func(a=1, c=3, b=2)
```

5. I ostatni raz: jaki będzie wynik poniższego kodu i dlaczego?

```
>>> def func(a, b, c=3, d=4): print(a, b, c, d)
...
>>> func(1, *(5,6))
```

6. Należy podać trzy lub większą liczbę sposobów, na jakie funkcje mogą przekazywać wyniki do kodu wywołującego.

Sprawdź swoją wiedzę — odpowiedzi

1. Wynikiem będzie '1 2 5', ponieważ 1 i 2 przekazywane są do a i b przez pozycję, natomiast c jest pominięte w wywołaniu i otrzymuje wartość domyślną 5.
2. Wynikiem tym razem będzie '1 2 3'. 1 przekazywane jest do a przez pozycję, natomiast do b oraz c przez nazwę przekazane zostają 2 i 3 (kolejność od lewej do prawej strony nie ma znaczenia, kiedy w taki sposób wykorzystujemy argumenty-słowa kluczowe).
3. Kod wyświetla '1 (2, 3)', ponieważ 1 przekazywane jest do a, natomiast *pargs zbiera pozostałe argumenty pozycyjne w nowy obiekt krotki. Krotkę z dodatkowymi argumentami pozycyjnymi możemy przejść za pomocą dowolnego narzędzia iteracyjnego (na przykład for arg in pargs: ...).
4. Tym razem kod wyświetla '1, {'c': 3, 'b': 2}', ponieważ 1 przekazane zostaje do a przez nazwę, natomiast **kargs zbiera pozostałe argumenty-słowa kluczowe w słownik. Słownik z dodatkowymi argumentami-słowami kluczowymi możemy przejść po kluczu za pomocą dowolnego narzędzia iteracyjnego (na przykład for key in kargs: ...).
5. Wynikiem będzie tym razem '1 5 6 4'. 1 dopasowane zostaje do a za pomocą pozycji, 5 i 6 dopasowane zostają do b i c za pomocą konstrukcji *nazwa (6 nadpisuje wartość domyślną c), natomiast d ma wartość domyślną 4, ponieważ do zmiennej tej nie przekazano wartości.
6. Funkcje mogą zwracać wyniki za pomocą instrukcji return, modyfikując przekazane zmienne argumenty i ustawiając zmienne globalne. Zmienne globalne nie są witane z entuzjazmem (z wyjątkiem bardzo specjalnych przypadków, jak programy wielowątkowe), ponieważ sprawiają, że kod trudniej jest zrozumieć i wykorzystywać. Instrukcje return są zazwyczaj najlepsze, a modyfikacja zmiennych obiektów jest w porządku, o ile jest oceniana. Funkcje mogą również komunikować się z narzędziami systemowymi, takimi jak pliki i gniazda, jednak wykracza to poza zakres zagadnień omawianych tutaj.

Zaawansowane zagadnienia dotyczące funkcji

Niniejszy rozdział wprowadza wiele bardziej zaawansowanych zagadnień związanych z funkcjami — funkcje rekurencyjne, atrybuty i adnotacje, wyrażenie `lambda`, narzędzia programowania funkcyjnego, takie jak `map` i `filter`. To bardziej zaawansowane narzędzia, których część Czytelników może nie napotkać w swojej codziennej pracy. Jednak z powodu ich znaczenia w niektórych dziedzinach programowania użyteczna może okazać się ich znajomość choćby na podstawowym poziomie. Na przykład konstrukcje `lambda` są dość powszechnie w programowaniu graficznych interfejsów użytkownika.

Część sztuki wykorzystywania funkcji polega na używaniu interfejsów pomiędzy nimi, dlatego przy okazji omówimy również pewne ogólne zasady projektowania funkcji. Następny rozdział kontynuuje to zagadnienie, wprowadzając zaawansowane pojęcia funkcji i wyrażeń generatorów oraz pogłębiając tematykę list składanych i ich zastosowań w kontekście narzędzi funkcyjnych przedstawionych w niniejszym rozdziale.

Koncepcje projektowania funkcji

Skoro poznaliśmy już podstawy tworzenia funkcji w Pythonie, warto powiedzieć parę słów na temat kontekstu. Gdy ktoś zaczyna na poważnie korzystać z funkcji, powinien podjąć kilka decyzji dotyczących kontekstu, na przykład sposobu rozbijania zadania na funkcje (co jest znane pod pojęciem *spójności*, ang. *cohesion*), tego, w jaki sposób funkcje powinny komunikować się wzajemnie (co jest określone jako *sprzęganie*, ang. *coupling*) i tak dalej. Należy również wziąć pod uwagę takie rzeczy, jak rozmiary funkcji, ponieważ ten czynnik ma bezpośredni wpływ na użyteczność kodu. Niektóre z tych zagadnień należą do kategorii analizy strukturalnej i projektowania i mają zastosowanie do kodu w Pythonie tak samo jak do innych języków programowania.

Niektóre zagadnienia dotyczące sprzęgania w kontekście funkcji i modułów poznaliśmy w rozdziale 17. przy okazji zakresów, poniżej przedstawiam krótkie podsumowanie tych zagadnień dla poczatkujących:

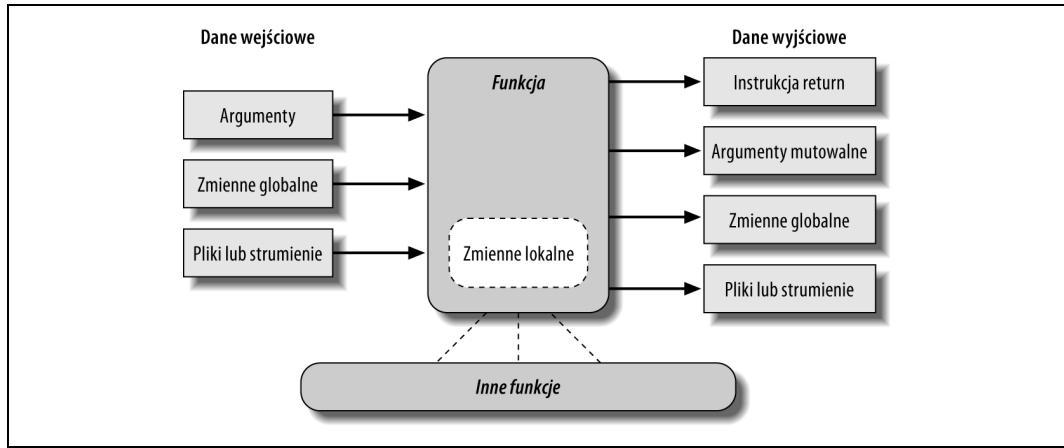
- **Sprzęganie: używanie argumentów jako danych wejściowych oraz instrukcji `return` do zwracania wyników.** Należy unikać uzależniania funkcji od jej otoczenia. Wykorzystanie argumentów i instrukcji `return` to najlepszy sposób odizolowania funkcji od zależności zewnętrznych, przy pozostawieniu ustalonych, dobrze oznaczonych punktów styku.

- **Sprzęganie: używanie zmiennych globalnych wyłącznie tam, gdzie to konieczne.** Zmienne globalne (to znaczy nazwy w module zawierającym funkcję) są z reguły niezalecanym sposobem komunikacji z funkcją. Mogą powodować skomplikowane zależności i inne trudne do wyśledzenia problemy z kodem.
- **Sprzęganie: nie należy modyfikować argumentów mutowalnych, o ile kod wywołujący tego nie oczekuje.** Funkcje mogą modyfikować część przekazywanych im obiektów mutowalnych, ale to może powodować mnóstwo sprzężeń między kodem wywołującym a wywoływanym, co z kolei powoduje, że funkcje stają się bardzo zależne od otoczenia.
- **Spójność: każda funkcja powinna mieć jeden, zunifikowany cel.** W przypadku dobrego projektu każda z funkcji powinna wykonywać pojedyncze zadanie, czynność, którą można określić jednym zdaniem oznajmującym. Jeśli to zdanie jest bardzo ogólne (na przykład: „Ta funkcja implementuje całą logikę programu”) lub złożone (na przykład: „Funkcja podnosi pracownikowi pensję i składa zamówienie na pizzę”), warto zastanowić się nad rozbiением funkcji na kilka mniejszych. W przeciwnym razie ponowne użycie funkcji może okazać się trudne z powodu skomplikowanego kontekstu jej zastosowania.
- **Rozmiar: każda funkcja powinna być relatywnie mała.** Ten wymóg wynika z poprzedniego, ale warto zwrócić uwagę na to, że jeśli funkcja zajmuje kilka ekranów na wyświetlaczu, to dobry sygnał, że warto ją rozbić na mniejsze części. Szczególnie biorąc pod uwagę spójność i czytelność jako istotniejsze cechy Pythona, rozwlekła i głęboko zagnieżdzona logika funkcji jest często symptomem problemów projektowych. Klucz do sukcesu leży w prostocie i zwięzości.
- **Sprzęganie: należy unikać bezpośredniego modyfikowania zmiennych zdefiniowanych w innym pliku modułu.** Tę koncepcję wprowadziliśmy w rozdziale 17. i pogłębimy ją w następnej części książki, przy okazji modułów. Jednak w celu zachowania kompletności tego wyliczenia należy wspomnieć o tym, że modyfikowanie zmiennych zdefiniowanych w innych modułach powoduje sprzężenie między modułami, analogiczne do sprzężenia między funkcjami, powodowanego przez użycie zmiennych globalnych. Powoduje, że moduły stają się skomplikowane i trudne do ponownego użycia. Wszędzie, gdzie to możliwe, należy stosować funkcje akcesorów zamiast bezpośrednich instrukcji przypisania.

Rysunek 19.1 zawiera podsumowanie metod komunikacji funkcji z ich otoczeniem: elementy po lewej stronie to możliwe źródła danych wejściowych, wyniki są pokazane po prawej. Dobry projekt funkcji wykorzystuje wyłącznie przekazywanie danych wejściowych w postaci argumentów, a zwracanie wyników za pomocą instrukcji return.

Oczywiście istnieje mnóstwo wyjątków od przedstawionych wyżej reguł, między innymi wynikających z technik programowania obiektowego. Jak zobaczymy w części VI, funkcjonowanie klas w Pythonie opiera się na modyfikowaniu instancji klasy przekazywanej do jej metod, innymi słowy, funkcje zdefiniowane w ciele klasy modyfikują przekazywany do nich automatycznie obiekt instancji self (na przykład w efekcie przypisania typu self.name = 'bob'). Co więcej, jeśli klasy nie są używane, najprostszym i najpowszechniejszym sposobem przekazywania stanu między wywołaniami funkcji są zmienne globalne. Efekty uboczne są niebezpieczne wyłącznie wówczas, gdy nie są spodziewane.

Jednak w ujęciu ogólnym należy starać się minimalizować zewnętrzne zależności funkcji i innych elementów programów. Im bardziej funkcja będzie niezależna od otoczenia, tym będzie łatwiejsza do zrozumienia, ponownego użycia i modyfikacji.



Rysunek 19.1. Środowisko wykonawcze funkcji. Funkcje mogą uzyskiwać dane wejściowe i zwracać wyniki na wiele sposobów, ale z reguły najłatwiej zrozumieć i rozwijać funkcje pobierające dane wejściowe wyłącznie z argumentów, a zwracające wyniki za pomocą instrukcji return, względnie w efekcie modyfikacji mutowalnych argumentów. W Pythonie 3 wynikiem funkcji mogą być również zadeklarowane zmienne nielokalne istniejące w zakresie definicji funkcji

Funkcje rekurencyjne

Przy okazji omawiania reguł zakresów na początku rozdziału 17. wspominałem o tym, że Python obsługuje *funkcje rekurencyjne*, to znaczy funkcje, które wywołują same siebie bezpośrednio lub za pośrednictwem innych funkcji. Rekurencja jest dość zaawansowanym zagadniением informatycznym i w Pythonie spotyka się ją dość rzadko. Jest to jednak użyteczna technika i warto o niej wiedzieć, ponieważ pozwala programistom przetwarzać dane cechujące się trudną do przewidzenia strukturą. Rekurencja bywa czasem alternatywą dla prostych pętli i iteracji, choć trudno nazwać ją alternatywą czytelniejszą lub łatwiejszą do zrozumienia.

Sumowanie z użyciem rekurencji

Przyjrzyjmy się kilku przykładom. Aby policzyć sumę elementów listy (lub innej sekwencji), można użyć wbudowanej funkcji `sum` lub napisać jej własną wersję. Gdyby taką funkcję napisać z użyciem rekurencji, miałaby mniej więcej taką postać:

```
>>> def mysum(L):
...     if not L:
...         return 0
...     else:
...         return L[0] + mysum(L[1:])          # Wywołanie samej siebie

>>> mysum([1, 2, 3, 4, 5])
15
```

Przy każdym wywoaniu funkcja rekurencyjnie wywołuje samą siebie w celu policzenia sumy pozostałych elementów listy, po czym ta suma jest dodawana do pierwszego odczytanego elementu. Rekurencyjne wywołania kończą się, gdy podlista jest pusta — w tym przypadku

funkcja zwraca zero. W tego typu wywołaniu rekurencyjnym każdy poziom zagnieźdżenia rekurencyjnego otrzymuje własną kopię lokalnego zakresu funkcji, co w naszym przykładzie oznacza, że `L` jest inną zmienną dla każdego wywołania.

Jeśli ten przykład jest trudny do zrozumienia (co się często zdarza w przypadku początkujących programistów), można do funkcji dodać wypisywanie wartości `L` i wywołać ją ponownie, dzięki czemu będziemy mieli wgląd w wartość tej zmiennej na każdym poziomie wywołania.

```
>>> def mysum(L):
...     print(L)
...     if not L:
...         return 0
...     else:
...         return L[0] + mysum(L[1:])
...
>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Śledzenie poziomów rekurencji
L z każdym wywołaniem ma coraz mniejszą długość

Jak widać, sumowana lista zmniejsza się dla każdego kolejnego zagnieźdżenia rekurencyjnego, aż staje się pusta, co powoduje zakończenie sekwencji rekurencyjnej. Suma jest obliczana wraz z *odwijaniem* się rekurencji.

Implementacje alternatywne

Co interesujące, alternatywnie możemy użyć trójskładnikowej operacji `if/else` (opisanej w rozdziale 12.), dzięki czemu możemy oszczędzić nieco kodu. Możemy również uogólnić funkcję, pozwalając zastosować listy dowolnych elementów, które można sumować (co okaże się łatwiejsze, jeśli przyjmiemy założenie, że dane wejściowe zawierają przynajmniej jeden element, podobnie jak to miało miejsce w rozdziale 18. w przykładzie wyznaczającym wartość minimalną). Zastosujemy rozszerzoną składnię przypisania sekwencyjnego wprowadzoną w 3.0, dzięki czemu rozpakowanie wartości będzie prostsze (szczegółowy opis tej techniki znajduje się w rozdziale 11.).

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:]) # Użycie wyrażeń trójskładnikowych

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:]) # Dowolny typ, domyślnie wartość 1

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest) # Użycie rozszerzonego przypisania
                                                       # sekwencji z 3.0
```

Zastosowane techniki nie obsługują pustych danych wejściowych, ale w zamian za to możemy użyć danych dowolnego typu obsługującego operator dodawania (+), a więc nie jesteśmy już ograniczeni do liczb.

```
>>> mysum([1]) # mysum([]) nie zadziała dla ostatnich 2 wersji funkcji
1
>>> mysum([1, 2, 3, 4, 5])
15
```

```
>>> mysum('j', 'a', 'j', 'k', 'o')           # Ale funkcja działa dla różnych typów
'jajko'
>>> mysum(['mielonka', 'szynka', 'jajka'])
'mielonkaszynkajajka'
```

Jeśli przeanalizujemy te trzy warianty, zauważymy, że dwa ostatnie obsługują również pojęty argument będący ciągiem znaków (zadziała wywołanie `mysum('mielonka')`), ponieważ ciągi znaków są sekwencjami pojedynczych znaków. Ostatni wariant obsługuje dodatkowo dowolne iteratory, jak otwarte pliki; dwa pierwsze już nie, ponieważ wykorzystują dostęp do elementów po indeksie. Wywołanie typu `mysum(first, *rest)` również nie ma sensu, ponieważ funkcja oczekuje indywidualnych elementów, nie pojedynczego iteratora.

Należy pamiętać, że rekurencja może być bezpośrednia, jak w przedstawionych przykładach, lub *pośrednia*, jak w listingu poniżej (funkcja wywołuje inną funkcję, która z kolei wywołuje tę pierwszą). Efekt jest ten sam, ale mamy do czynienia z dwoma funkcjami na każdym poziomie rekurencji, nie z jedną.

```
>>> def mysum(L):
...     if not L: return 0
...     return nonempty(L)                                # Wywołuję funkcję, która wywołała mnie
...
>>> def nonempty(L):
...     return L[0] + mysum(L[1:])                      # Rekurencja pośrednia
...
>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

Pętle a rekurencja

Rekurencja działa prawidłowo w przykładach prezentowanych w poprzednim punkcie, ale jej zastosowanie w tak prostym zadaniu to przerost formy nad treścią. W rzeczywistości rekurencja nie jest stosowana w Pythonie tak często jak w innych językach programowania, na przykład w Prologu czy Lisp, ponieważ Python kładzie nacisk na prostsze instrukcje proceduralne, jak pętle, które z reguły pozwalają uzyskać znacznie naturalniejsze rozwiązania. Na przykład pętla `while` często oferuje bardziej konkretny algorytm i nie wymaga pisania funkcji rekurencyjnej.

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
...     sum += L[0]
...     L = L[1:]
...
>>> sum
15
```

Co więcej, pętla `for` automatycznie wykonuje iterację, dzięki czemu w większości przypadków rekurencja staje się zupełnie zbędna (a co nie mniej ważne, najczęściej mniej wydajna z punktu widzenia zużycia pamięci i czasu wykonania).

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x
...
>>> sum
15
```

W przypadku pętli nie potrzebujemy kopii fragmentu sekwencji w lokalnym zakresie funkcji dla każdej iteracji, unikamy też kosztów czasowych związanych z wywołaniami funkcji (w rozdziale 20. poświęcimy nieco więcej miejsca na zagadnienia związane z pomiarem wydajności alternatywnych implementacji).

Obsługa dowolnych struktur

Z drugiej strony, rekurencja (albo analogiczne algorytmy wykorzystujące stos danych) może być wykorzystana do przeglądania danych o skomplikowanych strukturach. Jako prostym przykładem rekurencji wykorzystanej w takim kontekście posłużymy się zadaniem obliczenia sumy elementów zapisanych w wielokrotnie zagnieżdżonych listach:

```
[1, [2, [3, 4], 5], 6, [7, 8]] # Dowolnie zagnieżdżone podlisty
```

Prosta pętla nie poradzi sobie, ponieważ nie mamy do czynienia z liniową iteracją. Zagnieżdżone pętle również na nic się zdadzą, ponieważ podlisty mogą być dowolnie zagnieżdżone, czyli efektywnie mieć dowolny kształt. Zamiast pętli posłużymy się następującym kodem wykorzystującym rekurencję w przypadku napotkania po drodze zagnieżdżonych podlist.

```
def sumtree(L):
    tot = 0
    for x in L:
        if not isinstance(x, list):
            tot += x
        else:
            tot += sumtree(x)
    return tot

# Dla każdego elementu na tym poziomie
# Bezpośrednie dodawanie liczb
# Rekurencja dla podlist

L = [1, [2, [3, 4], 5], 6, [7, 8]] # Dowolne zagnieżdżanie
print(sumtree(L)) # Wypisuje 36

# Przypadki zdegenerowane
print(sumtree([1, [2, [3, [4, [5]]]]])) # Wypisuje 15 (drzewo rozrośnięte po prawej)
print(sumtree([[[], 1], 2, 3], 4, 5))) # Wypisuje 15 (drzewo rozrośnięte po lewej)
```

Warto prześledzić dwa ostatnie przykłady wywołania, aby zrozumieć, w jaki sposób rekurencja przeszukuje zagnieżdżone listy. Ten przykład jest sztuczny, ale stanowi dobre odzwierciedlenie rzeczywistej klasy problemów, jak drzewa dziedziczenia, a nawet łańcuchy importowanych modułów. W rzeczywistości w dalszej części tej książki będziemy mieli okazję wykorzystać rekurencję w różnych typach zastosowań:

- w rozdziale 24. moduł *reloadall.py* wykorzystuje rekurencję do śledzenia łańcuchów importów,
- w rozdziale 28. moduł *classtree.py* śledzi drzewa dziedziczenia klas,
- w rozdziale 30. moduł *lister.py* również śledzi drzewa dziedziczenia.

Mimo że ze względu na prostotę i wydajność kodu instrukcje pętli powinny mieć przewagę nad rekurencją w przypadku liniowych operacji, to rekurencja jest techniką kluczową w takich zastosowaniach jak wspomniane wyżej przykłady.

Co więcej, należy być świadomym potencjalnych konsekwencji *niezamierzonych* rekurencji w programach. Jak się przekonamy w dalszej części książki, również własne implementacje specjalnych metod klas, jak *__setattr__* i *__getattribute__*, bywają podatne na rekurencję, jeśli zostaną użyte w sposób niewłaściwy. Rekurencja to potężne narzędzie, ale działa ona najlepiej wówczas, gdy jest dobrze kontrolowana.

Obiekty funkcji — atrybuty i adnotacje

Funkcje w Pythonie są bardziej elastyczne, niż można by się spodziewać. Jak mieliśmy już okazję się przekonać, funkcje w Pythonie są czymś więcej niż jedynie specyfikacją operacji dla kompilatora: funkcje są pełnoprawnymi *obiektami*, zapisanymi w dedykowanych obszarach pamięci. Dzięki temu można je przekazywać w programach i wywoływać niebezpośrednio. Można również wykonywać na nich operacje, które są rzadko kojarzone z funkcjami: przypisywać atrybuty i adnotacje.

Pośrednie wywołania funkcji

Ponieważ funkcje w Pythonie są obiektami, można pisać programy, które wykorzystują je w sposób generyczny. Obiekty funkcji można przypisywać do zmiennych, przekazywać do innych funkcji jako parametry, osadzać w strukturach danych, zwracać w wyniku działania innych funkcji i wykonywać na nich wiele innych operacji, analogicznie jak to się dzieje z liczbami czy ciągami znaków. Obiekty funkcji obsługują specjalne operacje: można je wywoływać, przekazując w nawiasach parametry wywołania w wyrażeniu funkcji. Mimo tych cech szczególnych funkcje należą do tej samej kategorii co pozostałe obiekty.

We wcześniejszych przykładach mieliśmy już okazję zaobserwować przypadki generycznego użycia funkcji, ale warto zrobić małą powtórkę w celu wyraźnego podkreślenia faktu oparcia funkcji w Pythonie na modelu obiektowym. Na przykład nazwa użyta w instrukcji `def` nie jest przez Pythona traktowana w szczególny sposób. Jest to po prostu przypisanie nazwy w bieżącym zakresie traktowanym dokładnie tak samo jak w przypadku jego wykorzystania w wyrażeniu przypisania po lewej stronie znaku `=`. Po wywoaniu instrukcji `def` nazwa funkcji staje się referencją do obiektu funkcji: ten sam obiekt można przypisać do innej nazwy i wywołać tę samą funkcję przez referencję:

```
>>> def echo(message):                                # Nazwa echo przypisana do obiektu funkcji
...     print(message)
...
>>> echo('Wywołanie bezpośrednie')                 # Wywołanie obiektu przez oryginalną nazwę

Direct call
>>> x = echo                                         # Teraz również zmienna x zawiera referencję do funkcji
>>> x('Wywołanie pośrednie!')                         # Wywołanie obiektu przez nazwę przez zastosowanie ()
Wywołanie pośrednie!
```

Argumenty są przekazywane przez przypisanie obiektów, zatem funkcje można przekazywać w argumentach innych funkcji. Funkcja, której przekazano inną funkcję, może ją wywołać, dodając nawiasy do nazwy i przekazując w nich argumenty wywoywanej funkcji:

```
>>> def indirect(func, arg):                          # Wywołanie przekazanego obiektu przez zastosowanie ()
...     func(arg)
...
>>> indirect(echo, 'Wywołanie argumentu!')          # Przekazanie funkcji do innej funkcji
Wywołanie argumentu!
```

Obiektów funkcji można użyć również w charakterze elementów innych struktur danych, tak samo jak by to były liczby czy ciągi znaków. W poniższym przykładzie funkcję zapisujemy w liście krotek, tworząc coś na kształt tabeli operacji. W Pythonie tego typu struktury danych mogą zawierać dowolne typy obiektów, zatem i w tym przypadku nie dzieje się nic szczególnego:

```

>>> schedule = [(echo, 'Mielonka!'), (echo, 'Szynka!')]
>>> for func, arg in schedule:
...     func(arg)                                     # Wywołanie funkcji osadzonych w kontenerze
...
Mielonka!
Szynka!

```

Powyższy kod po prostu iteruje po liście `schedule`, wywołując funkcję `echo` z argumentem, będącym drugim elementem krotki (w deklaracji pętli następuje rozpakowanie krotki; informacje o tej technice można znaleźć w rozdziale 13.). Jak mieliśmy okazję zaobserwować w przykładach z rozdziału 17., funkcje można tworzyć w innych funkcjach i *zwracać* je w celu wywołania w innym miejscu programu:

```

>>> def make(label):                      # Utworzenie funkcji, bez wywołania
...     def echo(message):
...         print(label + ':' + message)
...     return echo
...
>>> F = make('Mielonka')                # Zostaje zapisana etykieta przekazana do funkcji w wewnętrznym zakresie
>>> F('Szynka!')                       # Wywołanie funkcji utworzonej w funkcji make
Mielonka:Szynka!
>>> F('Jajka!')
Mielonka:Jajka!

```

Elastyczny model obiektowy Pythona w połączeniu z brakiem deklaracji typów sprawia, że to niezwykle elastyczny język programowania.

Introspekcja funkcji

Funkcje są obiektami, można więc przetwarzać je z użyciem standardowych narzędzi do obsługi obiektów. W rzeczywistości funkcje są znacznie bardziej elastyczne, niż można by się spodziewać. Po utworzeniu funkcji można ją po prostu wywołać.

```

>>> def func(a):
...     b = 'mielonka'
...     return b * a
...
>>> func(5)
'mielonkamielonkamielonkamielonkamielonka'

```

Jednak wyrażenie wywołania to tylko jedna z operacji dostępnych obiektom funkcji. Możemy również dokonać inspekcji atrybutów, wykorzystując standardowe narzędzia (poniżej przykład wywołany w 3.0, wyniki dla 2.6 są zbliżone):

```

>>> func.__name__
'func'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...pominęta część wyniku...
 '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']

```

Narzędzia introspekcji pozwalają również na eksplorację szczegółów implementacyjnych: funkcje posiadają *obiekty kodu*, które zawierają takie szczegóły dotyczące implementacji, jak zmienne lokalne czy zadeklarowane argumenty.

```

>>> func.__code__
<code object func at 0x0257C9B0, file "<stdin>", line 1>
>>> dir(func.__code__)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',

```

```

...pominięta część wyniku...
'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lnotab',
'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']

>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1

```

Twórcy narzędzi mogą wykorzystać taką informację do zarządzania funkcjami (my również, w rozdziale 38., będziemy mieli okazję skorzystać z tych możliwości — do walidacji argumentów funkcji w dekoratorach).

Atrybuty funkcji

Obiekty funkcji nie są ograniczone do zdefiniowanych w systemie atrybutów wykorzystanych w powyższym punkcie. Jak mieliśmy okazję zobaczyć w rozdziale 17., istnieje możliwość definiowania własnych atrybutów funkcji:

```

>>> func
<function func at 0x0257C738>
>>> func.count = 0
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...pominięta część wyniku...
__str__, '__subclasshook__', 'count', 'handles']

```

W rozdziale 17. własne atrybuty funkcji wykorzystywaliśmy do dołączenia do obiektów funkcji *dodatkowych informacji*, zamiast używać do tego celu innych dostępnych mechanizmów, jak zmienne globalne, zmienne nielokalne czy klasy. W przeciwieństwie do zmiennych nielokalnych atrybuty funkcji są dostępne wszędzie tam, gdzie jest dostępna sama funkcja. W pewnym sensie jest to sposób implementacji odpowiednika mechanizmu „statycznych zmiennych lokalnych” dostępnego w innych językach programowania. Są to zmienne, których nazwy są lokalne dla funkcji, ale ich wartość jest zachowana po zakończeniu jej działania. Atrybuty są połączone z obiektami, nie zakresami nazw, ale efekt końcowy jest zbliżony.

Adnotacje funkcji w Pythonie 3.0

W Pythonie 3.0 (ale nie w 2.6) istnieje możliwość dołączania *dodatkowych informacji* do obiektu funkcji. Są to dowolne, zdefiniowane przez użytkownika informacje dotyczące argumentów przekazywanych do funkcji i wyników z niej zwracanych. Python oferuje specjalną składnię do definiowania adnotacji, ale nie wykorzystuje ich w jakikolwiek sposób. Adnotacje funkcji są mechanizmem zupełnie opcjonalnym, a gdy są zdefiniowane, zostają zapisane w atrybucie `__annotations__` obiektu funkcji i mogą być użyte przez inne narzędzia.

Specyficzne dla Pythona 3.0 argumenty mogące być tylko słowami kluczowymi poznaliśmy w poprzednim rozdziale, adnotacje stanowią kolejny mechanizm jeszcze bardziej uogólniający

składnię nagłówka funkcji. Weźmy na przykład następującą funkcję przyjmującą trzy argumenty i zwracającą ich sumę:

```
>>> def func(a, b, c):
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Z punktu widzenia składni adnotacje funkcji są zapisane w wierszach nagłówka funkcji jako wyrażenia związane z argumentami i zwracanymi wartościami. W przypadku argumentów adnotacja występuje po dwukropku następującym po nazwie atrybutu, w przypadku wartości zwracanych do adnotacji służy sekwencja znaków `->` następująca po liście argumentów. Poniższy przykład przedstawia wersję tej samej funkcji z adnotacjami wszystkich trzech argumentów i wyniku:

```
>>> def func(a: 'mielonka', b: (1, 10), c: float) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Wywołanie funkcji zawierającej adnotacje odbywa się tak samo jak każdej innej funkcji — w przypadku zdefiniowania adnotacji Python po prostu zapisuje je w słowniku i dołącza do obiektu funkcji. Kluczami są nazwy argumentów funkcji, adnotacja wyniku funkcji jest zapisana pod kluczem `return`. Wartościami w tym słowniku są wartości wyrażeń adnotacji:

```
>>> func.__annotations__
{'a': 'mielonka', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Adnotacje są obiektami Pythona dołączonymi do obiektu Pythona, dzięki czemu ich obsługa jest bardzo prosta. W poniższym przykładzie definiujemy adnotacje dla dwóch z trzech argumentów funkcji, po czym przeglądamy wszystkie adnotacje, wykorzystując standardowe narzędzia języka:

```
>>> def func(a: 'mielonka', b, c: 99):
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func.__annotations__
{'a': 'mielonka', 'c': 99}

>>> for arg in func.__annotations__:
...     print(arg, '=>', func.__annotations__[arg])
...
a => mielonka
c => 99
```

Warto tu zwrócić uwagę na dwa ciekawe szczegóły. Po pierwsze, korzystając z adnotacji, nie musimy rezygnować z definiowania domyślnych wartości argumentów: adnotacja (wraz ze znakiem `:`) występuje przed deklaracją wartości domyślnej (wraz z jej znakiem `=`). W poniższym przykładzie zapis `a: 'mielonka' = 4` oznacza, że wartością domyślną argumentu `a` jest 4, a argument posiada adnotację `'mielonka'`.

```
>>> def func(a: 'mielonka' = 4, b: (1, 10) = 5, c: float = 6) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

```
>>> func()                      # 4 + 5 + 6  (wartości domyślne)
15
>>> func(1, c=10)                # 1 + 5 + 10 (parametry kluczowe działają standardowo)
16
>>> func.__annotations__
{'a': 'mielonka', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Po drugie, warto zauważyc, że białe znaki są opcjonalne: pomiędzy elementami w nagłówku funkcji mogą wystąpić białe znaki, ale nie muszą, jednak pominięcie ich może zmniejszyć czytelność kodu.

```
>>> def func(a:'mielonka'=4, b:(1,10)=5, c:float=6)->int:
...     return a + b + c
...
>>> func(1, 2)                   # 1 + 2 + 6
9
>>> func.__annotations__
{'a': 'mielonka', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Adnotacje funkcji są nowym elementem języka i zostały wprowadzone w wersji 3.0. Wiele z ich potencjalnych zastosowań nadal pozostaje nieodkrytych. Łatwo wyobrazić sobie użycie adnotacji do określania ograniczeń typów lub wartości — bardziej rozbudowane API mogą wykorzystać tę cechę języka do rejestracji informacji o funkcji. W rozdziale 38. spróbujemy wykorzystać tę możliwość jako alternatywny sposób dekorowania argumentów funkcji (ogólny koncept kodowania informacji poza nagłówkiem funkcji, przez co jest on ograniczony tylko do jednej roli). Podobnie jak sam Python, adnotacje są narzędziem, którego zastosowanie jest ograniczone jedynie wyobraźnią programisty.

Na koniec należy zaznaczyć, że adnotacje działają wyłącznie w instrukcji `def`, nie są dostępne w wyrażeniach `lambda`, ponieważ składnia `lambda` sama w sobie narzuca ograniczenia definionym funkcjom. I w ten sposób przechodzimy do następnego tematu.

Funkcje anonimowe — `lambda`

Obok instrukcji `def` Python udostępnia również formę wyrażenia generującego obiekty funkcji. Ze względu na podobieństwo do narzędzia z języka LISP nosi ono nazwę `lambda`¹. Tak jak `def`, wyrażenie to tworzy funkcję, którą można wywołać później, jednak zwraca tę funkcję, zamiast przypisywać ją do nazwy. Z tego powodu wyrażenia `lambda` nazywane są czasami funkcjami *anonimowymi* (nienazwanymi). W praktyce często wykorzystywane są jako sposób skrótnego zapisania definicji funkcji lub opóźnienia wykonania fragmentu kodu.

Wyrażenia `lambda`

Na ogólną formę wyrażenia `lambda` składa się słowo kluczowe `lambda`, po którym następuje jeden lub większa liczba argumentów (dokładnie tak samo, jak argumenty umieszczone w nawiasach w nagłówku instrukcji `def`), a po nich, po dwukropku, wyrażenie.

```
lambda argument1, argument2,... argumentN : wyrażenie wykorzystujące argumenty
```

¹ Nazwa `lambda` wydaje się odstraszać ludzi bardziej, niż powinna. Reakcja ta wydaje się wynikać z samej nazwy „`lambda`” — nazwy, która pochodzi z języka LISP, który z kolei zaczerpnął ją z rachunku lambda, formy logiki symbolicznej. W Pythonie jest to jednak tylko słowo kluczowe wprowadzające wyrażenie. Odsuńmy zatem matematyczne dziedzictwo na bok, `lambda` jest prostsza w użyciu, niż może się wydawać.

Obiekty funkcji zwarcane przez wykonanie wyrażeń lambda działają dokładnie tak samo jak te utworzone i przypisane przez instrukcję def. Istnieje jednak kilka różnic sprawiających, że wyrażenie lambda staje się użyteczne w pewnych wyspecjalizowanych rolach.

- **Lambda jest wyrażeniem, a nie instrukcją.** Z tego powodu może się pojawiać w miejscach, w których użycie def w składni Pythona nie będzie dozwolone — wewnątrz literatu listy czy w wywołaniu funkcji. Jako wyrażenie lambda zwraca również wartość (nową funkcję), do której opcjonalnie można przypisać nazwę. W przeciwieństwie do tego instrukcja def zawsze przypisuje nową funkcję do nazwy z nagłówka, zamiast zwracać ją jako wynik.
- **Ciałem lambda jest pojedyncze wyrażenie, a nie blok instrukcji.** Ciało wyrażenia lambda jest podobne do tego, co umieszcza się w instrukcji return ciała instrukcji def. Wynik wypisuje się po prostu jako gołe wyrażenie, zamiast w jawnym sposobie go zwracać. Ponieważ lambda ograniczona jest do wyrażenia, jest mniej uniwersalna od def — ilość logiki, jaką można wstawić do ciała tego wyrażenia bez używania instrukcji, takich jak if, jest stosunkowo ograniczona. Taki zabieg jest celowy — ogranicza on zagnieżdżanie programu. Wyrażenie lambda zostało zaprojektowane z myślą o zapisywaniu prostych funkcji, natomiast instrukcja def zajmuje się większymi zadaniami programistycznymi.

Poza tymi różnicami def i lambda wykonują ten sam rodzaj pracy. Widzieliśmy na przykład, w jaki sposób tworzy się funkcję za pomocą instrukcji def:

```
>>> def func(x, y, z): return x + y + z
...
>>> func(2, 3, 4)
9
```

Ten sam efekt można uzyskać za pomocą wyrażenia lambda, w jawnym sposobie przypisując jego wynik do zmiennej, za pomocą której można później wywołać funkcję.

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

W powyższym kodzie zmienna f przypisywana jest do obiektu funkcji tworzonego przez wyrażenie lambda. W ten sam sposób działa instrukcja def, jednak przypisanie jest tam automatyczne.

Domyślne wartości argumentów działają w wyrażeniach lambda w taki sam sposób jak w instrukcjach def.

```
>>> x = (lambda a="raz", b="dwa", c="trzy": a + b + c)
>>> x("las")
'asdwaterzy'
```

Kod w ciele wyrażenia lambda przestrzega tych samych reguł przeszukiwania zakresów co kod wewnętrz instrukcji def. Wyrażenia lambda wprowadzają zakres lokalny — dokładnie tak, jak zagnieżdżone instrukcje def — który automatycznie (dzięki regule LEGB) widzi nazwy w funkcjach je zawierających, module oraz zakresie wbudowanym.

```
>>> def knights( ):
...     title = 'Sir'
...     action = (lambda x: title + ' ' + x) # Tytuł w instrukcji def zawierającej lambda
...     return action                         # Zwrócenie funkcji
...
>>> act = knights( )
>>> act('Robin')
'Sir Robin'
```

W powyższym przykładzie w wersjach Pythona starszych od 2.2 wartość zmiennej `title` była zamiast tego przekazywana jako domyślna wartość argumentu. Jeśli nie pamiętamy, dla czego tak było, możemy to zawsze sprawdzić, wracając do opisu zakresów z rozdziału 17.

Po co używa się wyrażenia lambda?

Wyrażenie `lambda` przydaje się jako rodzaj skrótu funkcji, który pozwala osadzać definicję funkcji w kodzie ją wykorzystującym. Wyrażenia te są całkowicie opcjonalne (zamiast nich zawsze można użyć instrukcji `def`), jednak na ogół są to proste konstrukcje kodu przydatne w sytuacjach, w których musimy osadzić niewielkie fragmenty kodu wykonywalnego.

Później zobaczymy na przykład, że programy obsługujące wywoływanie zwrotnych są często zapisywane jako wyrażenia `lambda` osadzone bezpośrednio w liście argumentów wywołania rejestrującego — zamiast opcji ze zdefiniowaniem ich za pomocą instrukcji `def` gdzieś w pliku i odniesieniem się do niej po jej nazwie (przykład takiego działania znajduje się w ramce „Znaczenie wywołań zwrotnych” w dalszej części rozdziału).

Wyrażenia `lambda` są również często wykorzystywane do tworzenia *tablic skoków* (ang. *jump table*), będących listami lub słownikami działań, które mają być wykonane na żądanie. Na przykład:

```
L = [(lambda x: x**2),           # Wewnętrzna definicja funkcji
      (lambda x: x**3),
      (lambda x: x**4)]             # Lista trzech wywoływalnych funkcji

for f in L:
    print(f(2))                  # Wyświetla 4, 8, 16

print(L[0](3))                  # Wyświetla 9
```

Wyrażenie `lambda` jest najbardziej przydatne jako skrót dla `def`, kiedy musimy zmieścić niewielkie fragmenty kodu wykonywalnego w miejscach, w których instrukcje będą niepoprawne z punktu widzenia składni. Poniższy fragment kodu tworzy na przykład listę trzech funkcji, osadzając wyrażenia `lambda` wewnątrz literała listy. Instrukcja `def` nie może się pojawić wewnątrz literala listy, ponieważ jest instrukcją, a nie wyrażeniem. Odpowiednik w postaci tradycyjnej instrukcji `def` wymagałby zastosowania tymczasowych nazw funkcji definiowanych poza kontekstem ich użycia.

```
def f1(x): return x ** 2
def f2(x): return x ** 3           # Definiujemy nazwane funkcje
def f3(x): return x ** 4

L = [f1, f2, f3]                 # Odwołanie przez nazwę

for f in L:
    print(f(2))                  # Wyświetla 4, 8, 16

print(L[0](3))                  # Wyświetla 9
```

To samo można zrobić za pomocą słowników czy innych struktur danych, które służą w Pythonie do budowania tabeli działań.

```
>>> key = 'już'
>>> {'mam': (lambda: 2 + 2),
...     'już': (lambda: 2 * 4),
...     'jeden': (lambda: 2 ** 6)}[key]()
8
```

W powyższym kodzie, kiedy Python tworzy tymczasowy słownik, każde z zagnieźdzonych wyrażeń `lambda` generuje i pozostawia funkcję, którą można wywołać później. Indeksowanie po kluczu powoduje pobranie jednej z funkcji, a nawiasy wymuszają jej wywołanie. W takiej postaci słownik staje się bardziej uniwersalnym narzędziem rozgałęziania kodu od tego, co mogłem pokazać w omówieniu instrukcji `if` z rozdziału 12.

By uzyskać to samo bez użycia wyrażeń `lambda`, musielibyśmy utworzyć gdzieś w pliku, poza słownikiem, w którym funkcje mają być użyte, trzy instrukcje `def` oraz odwoływać się do tej funkcji po nazwie.

```
>>> def f1(): return 2 + 2
...
>>> def f2(): return 2 * 4
...
>>> def f3(): return 2 ** 6
...
>>> key = 'jeden'
>>> {'mam': f1, 'już': f2, 'jeden': f3}[key]()
64
```

Takie rozwiązanie również działa, jednak instrukcje `def` mogą być w pliku mocno od siebie oddalone, nawet jeśli składają się jedynie z niewielkiego fragmentu kodu. *Bliskość kodu* udostępniana przez wyrażenia `lambda` okazuje się szczególnie wygodna w przypadku funkcji, które będą używane w tylko jednym kontekście. Jeśli trzy wymienione tu funkcje nie przydadzą się nam nigdzie indziej, osadzenie ich definicji wewnątrz słownika w postaci wyrażeń `lambda` ma sens. Co więcej, forma instrukcji wymaga nadania tym trzem niewielkim funkcjom nazw, które potencjalnie mogą pozostać w konflikcie z innymi nazwami z tego pliku.

Wyrażenia `lambda` przydają się również w listach argumentów funkcji, gdzie pozwalają na umieszczenie tymczasowych definicji funkcji nieużywanych nigdzie indziej w naszym programie. Przykłady takiego zastosowania zobaczymy nieco później, przy okazji omawiania funkcji `map`.

Jak łatwo zaciemnić kod napisany w Pythonie

Fakt, iż ciało `lambda` musi być pojedynczym wyrażeniem (a nie serią instrukcji), wydaje się nakładać na nas ograniczenia w zakresie tego, ile logiki można upakować w jednym `lambda`. Jeśli jednak wiemy, co robimy, większość instrukcji możemy w Pythonie zapisać za pomocą ich odpowiedników będących wyrażeniami.

Jeśli na przykład chcemy wyświetlić coś z ciała funkcji `lambda`, wystarczy w miejsce `print` użyć `sys.stdout.write(str(x) + '\n')` — jak pamiętamy z rozdziału 11., właśnie to robi instrukcja `print`. I podobnie, by osadzić logikę w wyrażeniu `lambda`, można użyć wyrażenia trójargumentowego z `if` oraz `else` przedstawionego w rozdziale 12. lub opisanego w tym samym miejscu (i nieco bardziej podchwytliwego) jego odpowiednika — kombinacji `and` oraz `or`. Jak już wiemy, poniższą instrukcję:

```
if a:
    b
else:
    c
```

można zastąpić jednym z odpowiadających jej wyrażeń:

```
b if a else c
((a and b) or c)
```

Ponieważ takie wyrażenia można umieszczać wewnętrz wyrażeń lambda, mogą one zostać wykorzystane do implementowania wewnętrz funkcji logiki wyboru.

```
>>> lower = (lambda x, y: x if x < y else y)
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

Co więcej, jeśli potrzebujemy wewnętrz wyrażenia lambda wykonać pętlę, możemy również osadzić w nim elementy takie, jak wywołania map czy wyrażenia list składanych (narzędzia omówione we wcześniejszych rozdziałach, do których powróćmy w dalszej części niniejszego rozdziału).

```
>>> import sys
>>> showall = (lambda x: map(sys.stdout.write, x))      # Użycie listy w3.0

>>> t = showall(['mielonka\n', 'tost\n', 'jajka\n'])
mielonka
tost
jajka

>>> showall = lambda x: [sys.stdout.write(line) for line in x]

>>> t = showall(['patrz\n', 'na\n', 'życie\n', 'z\n', 'humorem\n'])
patrz
na
życie
z
humorem
```

Skoro już zaprezentowałem te sztuczki, czuję się zobowiązany poprosić o stosowanie ich tylko w ostateczności. Bez zachowania ostrożności łatwo mogą się one stać nieczytelnym (inaczej *zaciemnionym*) kodem w Pythonie. Jak zawsze proste jest lepsze od skomplikowanego, a jawne lepsze od niejawnego. Z tego powodu pełne instrukcje są lepsze od zawiłych wyrażeń. Z drugiej strony, takie techniki mogą się nam czasem przydać, o ile używane będą z rozsądkiem.

Zagnieżdżone wyrażenia lambda a zakresy

Wyrażenia lambda najbardziej zyskały na wprowadzeniu wyszukiwania w zakresach funkcji zagnieżdżonych (czyli *E* z reguły LEGB omówionej w rozdziale 17.). W poniższym kodzie wyrażenie lambda pojawia się na przykład wewnętrz instrukcji def (typowy przypadek), dlatego możemy uzyskać dostęp do wartości, jaką zmienna *x* miała w zakresie funkcji ją zawierającej w momencie wywołania tej funkcji.

```
>>> def action(x):
...     return (lambda y: x + y)          # Utworzenie i zwrócenie funkcji, zapamiętanie x
...
>>> act = action(99)
>>> act
<function <lambda> at 0x00A16A88>
>>> act(2)                          # Wywołanie wyniku funkcji action
101
```

Czego nie było widać w omówieniu zakresów funkcji zagnieżdżonych w poprzednim rozdziale, to to, że wyrażenie lambda ma dostęp do zmiennych z zawierającego je innego wyrażenia lambda. Taki przypadek jest nieco nietypowy, ale wyobraźmy sobie sytuację, w której instrukcję def z poprzedniego przykładu zastępujemy wyrażeniem lambda.

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

W powyższym kodzie zagnieżdżona struktura lambda tworzy funkcję, która przy wywołaniu tworzy kolejną funkcję. W obu przypadkach kod zagnieżdzonego wyrażenia lambda ma dostęp do zmiennej `x` z zewnętrznego lambda. Takie rozwiązanie działa, choć jest nieco zagmatwane. W interesie czytelności kodu lepiej jest unikać zagnieżdżania wyrażeń lambda.

Znaczenie wywołań zwrotnych

Innym często wykorzystywanyem zastosowaniem wyrażeń lambda jest definiowanie funkcji zwrotnych dla API graficznego interfejsu użytkownika `tkinter` Pythona (ten moduł w 2.6 nosi nazwę `Tkinter`). Poniższy kod tworzy na przykład przycisk wyświetlający po jego naciśnięciu komunikat w konsoli (zakładając, że `tkinter` jest dostępny, jest domyślnie instalowany w Windows i innych systemach operacyjnych).

```
import sys
from tkinter import Button, mainloop # Tkinter w 2.6
x = Button(
    text ='Naciśnij mnie',
    command=(lambda:sys.stdout.write('Mielonka\n')))
x.pack()
mainloop()
```

Tutaj program obsługujący wywołanie zwrotnego rejestrowany jest za pomocą przekazania funkcji wygenerowanej przy użyciu wyrażenia `lambda` do argumentu ze słowem kluczowym `command`. Przewaga `lambda` nad `def` polega tutaj na tym, że kod obsługujący naciśnięcie przycisku jest od razu na miejscu, osadzony w wywołaniu tworzącym sam przycisk.

W rezultacie `lambda` opóźnia wykonanie programu obsługi do momentu wystąpienia zdarzenia. Wywołanie `write` odbywa się w momencie naciśnięcia przycisku, a nie przy jego utworzeniu.

Ponieważ reguły zakresów zagnieżdżonych odnoszą się również do wyrażeń `lambda`, od Pythona 2.2 są one łatwiejsze w użyciu w roli programów obsługi wywołań zwrotnych — automatycznie widzą zmienne z funkcji, w których są tworzone, i w większości przypadków nie wymagają już przekazywania wartości domyślnych. Jest to szczególnie przydatne w dostępie do specjalnego argumentu `instancji self`, który jest zmienną lokalną w funkcjach metod klasy zawierającej (więcej informacji na temat klas znajduje się w części VI książki).

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.display("mielonka")))
    def display(self, message):
        ...użycie komunikatu...
```

W poprzednich wersjach Pythona nawet `self` trzeba było przekazywać z wartościami domyślnymi.

Odwzorowywanie funkcji na sekwencje — map

Jednym z najczęściej wykonywanych działań na listach i innych sekwencjach jest zastosowanie jakiejś operacji do każdego ich elementu i zebranie wyników. Uaktualnienie wszystkich liczników w liście można na przykład z łatwością wykonać za pomocą pętli `for`.

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)                      # Dodanie 10 do każdego elementu
...
>>> updated
[11, 12, 13, 14]
```

Ponieważ jest to tak często wykonywana operacja, Python udostępnia odpowiednią funkcję wbudowaną, która jest w stanie zrobić to za nas. Funkcja `map` służy do zastosowania przekazanej funkcji na każdym elemencie sekwencji i zwraca listę zawierającą wszystkie wyniki jej wywołania, jak w poniższym przykładzie.

```
>>> def inc(x): return x + 10                  # Funkcja do wykonania
...
>>> map(inc, counters)                         # Zebranie wyników
[11, 12, 13, 14]
```

Funkcja `map` została wprowadzona w rozdziałach 13. i 14. jako narzędzie przetwarzania równoległego. Tutaj wykorzystamy ją w nieco ciekawszy sposób, przekazując jej prawdziwą funkcję, którą należy zastosować do każdego elementu listy — `map` wywołuje funkcję `inc` na każdym elemencie listy i zbiera zwieracane wartości w listę. Należy pamiętać, że w Pythonie 3.0 funkcja `map` zwraca obiekt iterowany, dlatego w celu wyświetlenia wyników należy przekształcić je na listę. Taka operacja nie jest konieczna w wersji 2.6.

Ponieważ `map` oczekuje przekazania funkcji, jest kolejnym miejscem, w którym może pojawić się wyrażenie `lambda`.

```
>>> map(lambda x: x + 3, counters)            # Wyrażenie funkcji
[4, 5, 6, 7]
```

W powyższym kodzie funkcja dodaje 3 do każdego elementu z listy `counters`. Ponieważ funkcja ta nie będzie potrzebna nigdzie indziej, została zapisana za pomocą wyrażenia `lambda`. Takie zastosowanie `map` jest odpowiednikiem pętli `for`, dlatego z użyciem niewielkiej ilości dodatkowego kodu zawsze można samodzielnie utworzyć uniwersalne narzędzie odwzorowujące.

```
>>> def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
```

Załóżmy, że funkcja `inc` ma taką postać, jak zaprezentowana ostatnio. Możemy wykonać za jej pomocą odwzorowanie na sekwencji, używając wbudowanej funkcji `map` lub naszego odpowiednika.

```
>>> map(inc, [1, 2, 3])                      # Wbudowana funkcja zwraca iterator
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])                    # Nasza wersja zwraca listę (patrz generatory)
[11, 12, 13]
```

Ponieważ jednak `map` jest funkcją wbudowaną, jest zawsze dostępna, zawsze działa w ten sam sposób i ma pewną przewagę w zakresie wydajności (jest zwykle szybsza od ręcznie napisanej

pętli `for`, co udowodnimy w następnym rozdziale). Co więcej, funkcji `map` można używać na bardziej zaawansowane, niż dotychczas pokazano, sposoby. Przykładowo po podaniu kilku argumentów będących sekwencjami przesyła ona elementy pobrane z sekwencji równolegle jako osobne argumenty funkcji.

```
>>> pow(3, 4)                                # 3**4
81
>>> map(pow, [1, 2, 3], [2, 3, 4])          # 1**2, 2**3, 3**4
[1, 8, 81]
```

W przypadku N sekwencji funkcja `map` oczekuje funkcji N -argumentowej. W powyższym kodzie funkcja `pow` w każdym wywołaniu przyjmuje dwa argumenty — po jednym z każdej sekwencji przekazanej do funkcji `map`. Zasymulowanie tego typu uogólnienia w powyższym kodzie nie wymaga wiele pracy, ale odłożę to na później, gdy poznamy narzędzie iteracyjne. Choć moglibyśmy symulować takie zachowanie, nie ma to sensu, skoro Python udostępnia szybką funkcję wbudowaną.

Wywołanie `map` jest podobne do wyrażeń list składanych omówionych w rozdziale 14., z którymi spotkamy się zresztą jeszcze w niniejszym rozdziale. Funkcja `map` stosuje jednak do każdego elementu wywołanie przekazanej funkcji, a nie dowolne wyrażenie. Ze względu na to ograniczenie jest nieco mniej uniwersalna. W niektórych przypadkach `map` jest jednak obecnie szybsza do wykonania od list składanych (na przykład, kiedy odwzorowuje się funkcję wbudowaną) i może wymagać mniejszej ilości kodu.

Narzędzia programowania funkcyjnego — `filter` i `reduce`

Funkcja `map` jest najprostszym przedstawicielem klasy wbudowanych narzędzi Pythona wykorzystywanym w *programowaniu funkcyjnym* — co w zasadzie oznacza po prostu narzędzia stosujące funkcje na sekwencjach. Jej krewniacy odfiltrują elementy w oparciu o funkcję testującą (`filter`) lub stosują funkcje do par elementów i wykonują wyniki (`reduce`). Funkcje `range` i `filter` zwracają w 3.0 obiekty iterowane, dlatego w celu wyświetlenia ich wyników należy je przekształcić na listę. Poniższe wywołanie funkcji `filter` wybiera elementy sekwencji większe od zera.

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

>>> filter(lambda x: x > 0, range(-5, 5))
[1, 2, 3, 4]
```

Elementy sekwencji, dla których funkcja zwraca `True`, dodawane są do listy wyników. Podobnie jak `map`, funkcja ta jest przybliżonym odpowiednikiem pętli `for`, jednak jest narzędziem wbudowanym, a do tego szybkim.

```
>>> res = []
>>> for x in range(-5, 5):
...     if x > 0:
...         res.append(x)
...
>>> res
[1, 2, 3, 4]
```

Funkcja `reduce`, która w 2.6 jest zwykłą funkcją wbudowaną, a w 3.0 została przeniesiona do modułu `functools`, jest bardziej skomplikowana. Poniżej widać dwa wywołania `reduce` obliczające sumę oraz iloczyn elementów listy.

```
>>> from functools import reduce      # Import tylko w 3.0, nie 2.6
>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

W każdym kroku funkcja `reduce` przekazuje bieżącą sumę lub iloczyn wraz z kolejnym elementem listy do przekazanej funkcji `lambda`. Domyślnie pierwszy element sekwencji inicjalizuje wartość początkową. Poniżej znajduje się odpowiednik pierwszego wywołania utworzony z wykorzystaniem pętli `for`, z dodawaniem zakodowanym na stałe wewnątrz pętli.

```
>>> L = [1, 2, 3, 4]
>>> res = L[0]
>>> for x in L[1:]:
...     res = res + x
...
>>> res
10
```

Utworzenie własnej wersji funkcji `reduce` jest tak naprawdę dość proste. Poniższy listing przedstawia implementację emulującą większość jej możliwości, ujawniając zasadę działania.

```
>>> def myreduce(function, sequence):
...     tally = sequence[0]
...     for next in sequence[1:]:
...         tally = function(tally, next)
...     return tally
...
>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

Funkcja wbudowana `reduce` pozwala na użycie opcjonalnego trzeciego argumentu, który jest wstawiany w puste miejsca w przypadkach, gdy niektóre sekwencje są krótsze od innych, ale to rozszerzenie pozostawiam jako ćwiczenie dla Czytelnika.

Jeśli kogoś to zainteresowało, może zwróci uwagę również na wbudowany moduł `operator` udostępniający funkcje odpowiadające wbudowanym wyrażeniom, dzięki czemu przydaje się w niektórych zastosowaniach narzędzi funkcyjnych.

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6])           # Dodawanie oparte na funkcji
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Tak jak `map`, funkcje `filter` oraz `reduce` obsługują techniki programowania funkcyjnego o dużych możliwościach. Niektórzy obserwatorzy mogą również rozszerzyć zbiór narzędzi programowania funkcyjnego w Pythonie tak, by obejmował on także `lambda`, `apply` i listy składane będące tematem kolejnego podrozdziału.

Podsumowanie rozdziału

Niniejszy rozdział był omówieniem bardziej zaawansowanych zagadnień związanych z funkcjami — funkcji rekurencyjnych, adnotacji funkcji, wyrażeń funkcji lambda, narzędzi funkcyjnych, takich jak `map`, `filter` oraz `reduce`, a także ogólnych koncepcji związanych z projektowaniem funkcji. Powróciliśmy również do iteratorów i list składanych, ponieważ są one powiązane z programowaniem funkcyjnym w takim samym stopniu jak z instrukcjami pętli. Zanim jednak przejdziemy dalej, należy upewnić się, że opanowaliśmy podstawy funkcji, wykonując quiz podsumowujący rozdział, a także ćwiczenia kończące tę część książki.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób są ze sobą powiązane wyrażenia `lambda` oraz instrukcje `def`?
2. Jaki jest cel używania wyrażeń `lambda`?
3. Porównaj działanie funkcji `map`, `filter` i `reduce`.
4. Czym są adnotacje funkcji i w jaki sposób się ich używa?
5. Czym są funkcje rekurencyjne i w jaki sposób się ich używa?
6. Jakie są ogólne zalecenia dotyczące pisania funkcji?

Sprawdź swoją wiedzę — odpowiedzi

1. Zarówno `lambda`, jak i `def` mogą tworzyć obiekty funkcji, które mogą być wywołane później. Ponieważ jednak `lambda` jest wyrażeniem, można ją wykorzystać do zagnieźdzenia definicji funkcji w miejscach, w których `def` nie jest dozwolone przez składnię. Wykorzystywanie wyrażenia `lambda` nigdy nie jest wymagane — zawsze można zamiast niego użyć instrukcji `def` i odnieść się do funkcji przez jej nazwę. Wyrażenia te przydają się jednak do osadzania niewielkich fragmentów opóźnionego kodu, z którego raczej nie będziemy korzystać w żadnym innym miejscu programu. Z punktu widzenia składni `lambda` zezwala na jedno wyrażenie zwracające wartość. Ponieważ nie obsługuje bloków instrukcji, nie nadaje się do implementowania większych funkcji.
2. Wyrażenia `lambda` pozwalają na definiowanie prostych fragmentów kodu wykonywalnego z opóźnieniem jego wykonania i możliwością zachowania stanu w postaci domyślnych argumentów i zmiennych zakresu zewnętrznego. Użycie `lambda` nigdy nie jest konieczne, zawsze istnieje możliwość zdefiniowania funkcji z użyciem instrukcji `def` i odwoływaniami się do niej po nazwie. Wyrażenia `lambda` są wygodne w sytuacjach, gdy istnieje potrzeba osadzenia kodu wykonywalnego z opóźnieniem jego wywołania, w przypadku gdy ten kod nie będzie użyty w innych miejscach aplikacji. Powszechnym zastosowaniem wyrażeń `lambda` jest kod definicji interfejsu GUI, gdzie stosuje się je z narzędziami programowania funkcyjnego, jak `map` i `filter`, które jako jednego z parametrów wymagają funkcji.
3. Te funkcje wbudowane służą do wykonywania funkcji na elementach sekwencji (obiektu iterowanego) i zwracania wyniku takiej operacji. Funkcja `map` po prostu przekazuje elementy sekwencji jako argumenty funkcji i zwraca sekwencję wyników takich wywołań, `filter`

zwraca sekwencję złożoną z elementów, dla których podana funkcja zwraca wartość prawdziwą, reduce oblicza pojedynczą wartość, wykonując funkcję na akumulatorze i kolejnych elementach sekwencji. Dwie pierwsze są funkcjami wbudowanymi, reduce od 3.0 jest dostępna w module `functools`.

4. Adnotacje funkcji są dostępne od Pythona 3.0 i pozwalają na przypisanie dodatkowych informacji argumentom funkcji oraz wartości zwracanej. Adnotacje są zapisane jako atrybut `__annotations__` obiektu funkcji. Python nie wykorzystuje adnotacji w żaden dodatkowy sposób i są one obsługiwane jako mechanizm do potencjalnego użytku przez narzędzia zewnętrzne.
5. Funkcje rekurencyjne bezpośrednio lub pośrednio wywołują same siebie w celu wykonania pętli operacji. Mogą być użyte do przetwarzania struktur danych o niezdefiniowanych kształtach, można ich również użyć jako ogólnego mechanizmu iteracyjnego (to zastosowanie jednak lepiej powierzyć pętlom, które zrealizują je szybciej i bardziej efektywnie z punktu widzenia pamięci).
6. Funkcje powinny być niewielkimi, samodzielnymi fragmentami logiki aplikacji, posiadać prosty, uogólniony cel i komunikować się z innymi elementami aplikacji za pośrednictwem swoich argumentów wywołania i zwracanych wyników. Mogą również wykorzystywać mutowalne argumenty i za ich pośrednictwem przekazywać zmiany, ale preferowane są dwa pierwsze mechanizmy komunikacji funkcji z otoczeniem.

Iteracje i składanie list — część 2.

W niniejszym rozdziale kontynuujemy omawianie zaawansowanych zagadnień związanych z mechanizmem rozwinięć i narzędziami iteracyjnymi wprowadzonymi w rozdziale 14. Ponieważ rozwinięcia list mają takie samo zastosowanie do pętli jak do mechanizmów funkcyjnych omówionych w poprzednim rozdziale (na przykład `map` oraz `filter`), w tym rozdziale omówię te zagadnienia z punktu widzenia właśnie mechanizmów funkcyjnych. Będzie to niejako inne spojrzenie na iteratory oraz wprowadzenie do tematyki funkcji generatorów i ich odpowiedników w postaci wyrażeń, pozwalających na budowanie przez programistę obiektów generujących wyniki na żądanie.

Iteracje w Pythonie mogą być również obsługiwane przez klasy definiowane przez użytkownika, ale tę część opowieści odłożę do części VI, w której zajmiemy się przeciążaniem operatorów. Ponieważ niniejszym rozdziałem zamknieniśmy omawianie wbudowanych mechanizmów iteracyjnych, znajdzie się tu podsumowanie różnych technik poznanych dotychczas, wraz z analizą wydajności wybranych z nich. Ponadto, w związku z tym, że to ostatni rozdział tej części książki, na jego końcu znajdziemy zwyczajowy zbiór pułapek czyhających na programistów oraz kilka ćwiczeń, które pomogą rozpoczęć praktyczną przygodę z poznanymi koncepcjami.

Listy składane, podejście drugie — narzędzia funkcyjne

W poprzednim rozdziale analizowaliśmy narzędzia programowania funkcyjnego, jak funkcje `map` i `filter`, które wykonują określone operacje na sekwencjach danych i gromadzą wyniki. Tego typu schemat operacji jest na tyle powszechny w Pythonie, że z czasem zaimplementowano w nim nowy typ wyrażeń: *złożenia list*, które dają poznanym narzędziom jeszcze więcej możliwości. Listy składane pozwalają bowiem wykonywać operacje na sekwencji danych w prostym *wyrażeniu*, bez konieczności użycia funkcji. Dzięki temu są narzędziem znacznie bardziej ogólnym.

Z zagadnieniem list składanych spotkaliśmy się już w rozdziale 14. w powiązaniu z instrukcjami pętli. Można je również stosować w połączeniu z narzędziami programowania funkcyjnego, jak funkcje `map` czy `filter`, zatem w niniejszym rozdziale odświeżymy ten temat, rozszerzając go o nowy kontekst. Z technicznego punktu widzenia listy składane nie są ściśle związane

z funkcjami. Jak się przekonamy, listy składane mogą być znacznie bardziej ogólnym narzędziem niż `map` i `filter`. Nowe koncepcje często łatwiej jest zrozumieć przez analogię, dlatego wprowadzę takie porównanie.

Listy składane kontra `map`

Przyjrzyjmy się przykładowi, który demonstruje podstawową wiedzę. Jak widzieliśmy w rozdziale 7., funkcja wbudowana `ord` zwraca kod liczbowy przekazanego znaku ASCII (jej odwrotnością jest funkcja `chr`, zwracająca znak na podstawie jego kodu ASCII).

```
>>> ord('s')
115
```

Załóżmy, że chcemy odczytać kody ASCII *wszystkich* znaków podanego ciągu. Najprostsze podejście polega na użyciu pętli `for`, która po kolei zapisze odpowiednie kody na liście:

```
>>> res = []
>>> for x in 'mielonka':
...     res.append(ord(x))
...
>>> res
[115, 112, 97, 109]
```

Jednak my już znamy funkcję `map`, zatem możemy pokusić się o osiągnięcie tego samego celu w jednym wywołaniu, bez konieczności inicjowania pustej listy i obsługi dopisywania do niej wartości.

```
>>> res = list(map(ord, 'mielonka'))          # Wywołanie funkcji na sekwencji
>>> res
[115, 112, 97, 109]
```

Jednak identyczny wynik osiągniemy, stosując wyrażenie listy składanej. Funkcja `map` dokonuje odwzorowania *funkcji* na sekwencji, natomiast lista składana dokonuje odwzorowania *wyrażenia* na sekwencji.

```
>>> res = [ord(x) for x in 'mielonka']        # Wywołanie wyrażenia na sekwencji
>>> res
[115, 112, 97, 109]
```

Listy składane gromadzą wyniki wywołania dowolnego wyrażenia na sekwencji wartości i zwracają nową sekwencję. Składnia list składanych wymaga użycia nawiasów kwadratowych (co sugeruje, że tworzą listę). W prostej wersji wewnętrz nawiasów kwadratowych definiuje się wyrażenie składające się z nazwy zmiennej, po której następuje coś, co przypomina uproszczenie deklaracji pętli `for`, w którym jest użyta ta sama zmienna. Python zgromadzi wyniki wykonania wyrażenia na elementach odczytanych z sekwencji w wewnętrznej pętli `for`.

Wynik wykonania powyższego wyrażenia przypomina wynik zastosowania pętli `for` lub funkcji `map`. Listy składane stają się wygodniejsze wówczas, gdy na sekwencji chcemy wykonać bardziej skomplikowane wyrażenia.

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

W powyższym kodzie obliczamy kwadraty liczb od 0 do 9 (pozwoliliśmy iteratorowi wyświetlić wynik, ale można go przypisać do zmiennej, jeśli chcemy go użyć w dalszej części kodu). Aby uzyskać podobny efekt za pomocą funkcji `map`, musielibyśmy sami napisać prostą funkcję

obliczającą kwadrat podanej liczby. Jeśli funkcja byłaby potrzebna tylko w tym jednym miejscu, moglibyśmy zdefiniować ją w miejscu za pomocą instrukcji `lambda`, zamiast kodować nazwaną funkcję za pomocą instrukcji `def`.

```
>>> list(map((lambda x: x ** 2), range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Powyższy kod wykonuje dokładnie to samo zadanie i jest zaledwie o kilka znaków dłuższy od odpowiadającej mu wersji używającej listy składanej. Jest też niewiele bardziej skomplikowany (oczywiście pod warunkiem że ktoś rozumie działanie instrukcji `lambda`). Jednak w przypadku bardziej skomplikowanych wyrażeń listy składane często pozwalają zaoszczędzić sporo pisania. Zademonstrujemy to w kolejnym punkcie.

Dodajemy warunki i pętle zagnieżdżone — filter

Listy składane są narzędziem jeszcze ogólniejszym, niż pokazaliśmy to dotychczas. Jak zauważyliśmy w rozdziale 14., w wyrażeniu listy składanej po deklaracji pętli `for` możemy dodać warunek `if`, który uzupełnia nasze wyrażenie o logikę wyboru danych. Listy składane zawierające warunek `if` można traktować jako odpowiednik funkcji `filter` omówionej w poprzednim rozdziale: pozwalają pominąć te elementy sekwencji, dla których warunek nie jest spełniony.

Zademonstrujmy obydwa podejścia na przykładzie wyboru liczb parzystych z sekwencji liczb od 0 do 4. Podobnie jak w powyższym przykładzie z funkcją `map`, wersja wykorzystująca funkcję `filter` do obsługi warunku używa prostej funkcji `lambda`. Dla kompletności przedstawiamy też odpowiednią wersję z pętlą `for`.

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> filter((lambda x: x % 2 == 0), range(5))
[0, 2, 4]

>>> res = []
>>> for x in range(5):
...     if x % 2 == 0:
...         res.append(x)
...
>>> res
[0, 2, 4]
```

Wszystkie wersje do wykrywania liczb parzystych wykorzystują operację modulo (reszta z dzielenia całkowitego, operator `%`): jeśli dzielenie przez 2 nie ma reszty, to dzielna jest liczbą parzystą. Wersja wykorzystująca funkcję `filter` jest niewiele dłuższa od wersji wykorzystującej listę składaną. Jednak w przypadku list składanych możemy wykorzystać skomplikowane wyrażenie oraz warunek wyboru danych z sekwencji, co w efekcie zadziała jak złożenie w jednym wyrażeniu funkcji `map` i `filter`.

```
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
```

Tym razem wynik ma zawierać kwadraty liczb parzystych z zakresu od 0 do 9 — wewnętrzna pętla `for` pomija liczby niespełniające warunku parzystości, a wyrażenie po lewej stronie `for` oblicza wynik. Analogiczne wywołanie funkcji `map` wymagałoby znacznie więcej pracy: musielibyśmy wykonać złożenie wywołań funkcji `map` z funkcją `filter`, co w efekcie da o wiele bardziej skomplikowane wyrażenie:

```
>>> list(map(lambda x: x**2), filter(lambda x: x % 2 == 0), range(10)) )
[0, 4, 16, 36, 64]
```

W rzeczywistości listy składane są jeszcze bardziej ogólnym narzędziem. Możemy bowiem wykorzystać dowolną liczbę zagnieżdżonych pętli `for`, każda z nich może zawierać warunek `if`. Ogólna struktura listy składanej jest następująca:

```
[wyrażenie for zmienna1 in sekwencja1 [if warunek1]
  for zmienna2 in sekwencja2 [if warunek2] ...
  for zmiennaN in sekwencjaN [if warunekN]]
```

Jeśli w wyrażeniu listy składanej zagnieżdziemy kolejne wyrażenie listy składanej, będzie ono działało analogicznie do zagnieżdżenia pętli `for`, na przykład:

```
>>> res = [x + y for x in [0, 1, 2] for y in [100, 200, 300]]
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Wyrażenie to daje taki sam wynik jak następujący (znacznie dłuższy) kod wykorzystujący pętle:

```
>>> res = []
>>> for x in [0, 1, 2]:
...     for y in [100, 200, 300]:
...         res.append(x + y)
...
>>> res
[100, 200, 300, 101, 201, 301, 102, 202, 302]
```

Choć listy składane zwracają listy, należy pamiętać, że mogą działać na dowolnej sekwencji lub innym obiekcie iterowanym. Poniższy przykład wykonuje złożenie listy na podstawie znaków z ciągów zamiast sekwencji liczb i zwraca kombinacje par znaków z każdego ciągu:

```
>>> [x + y for x in 'jajko' for y in 'JAJKO']
['j', 'JA', 'jA', 'jK', 'jO', 'aJ', 'aA', 'aJ', 'aK', 'aO', 'jJ', 'jA', 'jJ', 'jK',
 'jO', 'kJ', 'kA', 'kJ', 'kK', 'kO', 'oJ', 'oA', 'oJ', 'oK', 'oo']
```

I jeszcze jeden, bardziej skomplikowany przykład listy składanej, ilustrujący efekt użycia warunku `if` w zagnieżdżonym rozwinięciu `for`:

```
>>> [(x, y) for x in range(5) if x % 2 == 0 for y in range(5) if y % 2 == 1]
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Powyższe wyrażenie zwraca permutacje liczb parzystych z zakresu od 0 do 4 z liczbami nieparzystymi z tego zakresu. Warunki `if` odfiltrowują zbędne elementy w każdym z podwyrażeń. Wyrażenie to odpowiada następującemu kodowi z użyciem zagnieżdżeń pętli `for`:

```
>>> res = []
>>> for x in range(5):
...     if x % 2 == 0:
...         for y in range(5):
...             if y % 2 == 1:
...                 res.append((x, y))
...
>>> res
[(0, 1), (0, 3), (2, 1), (2, 3), (4, 1), (4, 3)]
```

Jak pamiętamy, w przypadku skomplikowanych wyrażeń list składanych możemy rozpisać je w kilku wierszach kodu z użyciem wcięć, co wizualnie da podobny efekt do powyższego. Kod będzie dłuższy, ale potencjalnie bardziej czytelny.

Wersja z użyciem funkcji `map` i `filter` będzie znacznie dłuższa i bardziej skomplikowana, z dużą liczbą zagnieżdżeń, zatem nie będę jej tu demonstrował. Pozostawiam to jako ćwiczenie dla mistrzów zen, byłych programistów Lispa i maniaków depresyjnych.

Listy składane i macierze

Oczywiście nie wszystkie zastosowania list składanych są tak akademickie. Przyjrzyjmy się zatem kilku zastosowaniom, które wymagają wyciągnięcia szarych komórek. Jednym ze sposobów implementacji macierzy (czyli tablic wielowymiarowych) w Pythonie jest struktura list zagnieżdżonych. Poniższy przykład implementuje macierz 3×3 w postaci listy zagnieżdżonych list:

```
>>> M = [[1, 2, 3],  
...       [4, 5, 6],  
...       [7, 8, 9]]  
  
>>> N = [[2, 2, 2],  
...       [3, 3, 3],  
...       [4, 4, 4]]
```

W tego typu strukturze możemy nawigować, podając indeks wiersza i indeks kolumny w wierszu:

```
>>> M[1]  
[4, 5, 6]  
  
>>> M[1][2]  
6
```

Listy składane są doskonałym narzędziem do przetwarzania struktur tego typu, ponieważ pozwalają przeszukiwać wiersze i kolumny w sposób automatyczny. Mimo że nasza struktura przechowuje macierz w postaci listy wierszy, aby wydobyć tylko drugą kolumnę, listy składane mogą przeiterować po wierszach, wydobywając kolumnę po indeksie, albo wręcz wykorzystać indeksy wierszy i indeks odpowiedniej kolumny:

```
>>> [row[1] for row in M]  
[2, 5, 8]  
  
>>> [M[row][1] for row in (0, 1, 2)]  
[2, 5, 8]
```

Dzięki indeksom możemy wykonywać bardziej skomplikowane operacje, jak wydobywanie przekątnej. Poniższe wyrażenie do wygenerowania sekwencji współrzędnych wykorzystuje funkcję range, a następnie z macierzy wydobywa wiersz i kolumnę o tej samej współrzędnej, na przykład $M[0][0]$, następnie $M[1][1]$ i tak dalej (zakładamy, że macierz jest kwadratowa).

```
>>> [M[i][i] for i in range(len(M))]  
[1, 5, 9]
```

Przy odrobinie kreatywności listy składane mogą posłużyć też do wykonywania operacji na macierzach. Poniższy przykład demonstruje utworzenie listy wyniku mnożenia odpowiadających komórek każdej z macierzy, kolejny przykład prezentuje sposób złożenia macierzy wynikowej (dzięki zagnieżdżeniu wyrażeń list składanych) zawierającej wynik mnożenia komórek macierzy wejściowych:

```
>>> [M[row][col] * N[row][col] for row in range(3) for col in range(3)]  
[2, 4, 6, 12, 15, 18, 28, 32, 36]  
  
>>> [[M[row][col] * N[row][col] for col in range(3)] for row in range(3)]  
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Ostatnie wyrażenie działa dzięki wykonaniu dwóch iteracji: zewnętrzna wydobywa wiersze i składa wynik, a wewnętrzna wykonuje operacje na odpowiednich komórkach. Poniższy kod prezentuje odpowiednik tego wyrażenia przy użyciu zagnieżdżonych pętli for:

```
>>> res = []
>>> for row in range(3):
...     tmp = []
...     for col in range(3):
...         tmp.append(M[row][col] * N[row][col])
...     res.append(tmp)
...
>>> res
[[2, 4, 6], [12, 15, 18], [28, 32, 36]]
```

Wersja z użyciem list składanych wymagała napisania jednego wiersza kodu, a w przypadku dużych macierzy będzie działała znacznie szybciej i istnieje duże ryzyko, że może doprowadzić głowę do eksplozji. Co prowadzi nas do następnego punktu.

Zrozumieć listy składane

Listy składane działają na tak wysokim poziomie uogólnienia, że szybko mogą stać się mało zrozumiałe, szczególnie przy użyciu zagnieżdżeń. Moja rada jest taka, aby poczynającą użytkownicy Pythona pisali tego typu kod w postaci pętli `for`, a funkcje `map` lub listy składane wykorzystywali wówczas, gdy są one łatwe w użyciu. Jak zawsze ma tu zastosowanie reguła „piękno tkwi w prostocie” — zwartość kodu jest o wiele mniej istotna od jego czytelności.

Jednak w tym przypadku dodatkowy poziom komplikacji pozwala uzyskać sporą korzyść w postaci wydajności: testy wykazują dwukrotnie większą wydajność wywołań `map` w porównaniu z pętlami realizującymi te same zadania, a listy składane są z reguły nieco wydajniejsze od funkcji `map`.¹ Tego typu różnice biorą się z tego, że funkcja `map` i listy składane są wykonywane w samym interpreterze z wydajnością języka C, natomiast pętle `for` są wykonywane przez maszynę wirtualną Pythona.

Pętle `for` pozwalają uzyskać większą czytelność kodu, przez co polecam ich stosowanie, jeśli chodzi o prostotę. Jednak warto znać zastosowania funkcji `map` i list składanych i nie obawiać się ich użycia, szczególnie w prostszych przypadkach lub w sytuacji, gdy kluczowe jest uzyskanie maksymalnej wydajności kodu. Ponadto z faktu, że wywołania `map` i listy składane są wyrażeniami, wynika, że można ich użyć w kontekstach, w których nie można zastosować pętli `for`, na przykład w ciele funkcji `lambda`, w literałach list i słowników itp. Należy jedynie pamiętać, aby wywołania funkcji `map` i list składanych były jak najprostsze, w przypadku skomplikowanych problemów lepiej użyć pętli.

¹ Należy pamiętać, że tego typu porównania są podatne na sposób wykonywania wywołań funkcji, jak również na optymalizacje dokonane w samym Pythonie. Najnowsze wydania Pythona zawierały na przykład optymalizacje właśnie w pętlach `for`. Jednak listy składane są z reguły znaczco wydajniejsze od pętli `for`, jak również od wywołań funkcji `map` (choć może być odwrotnie w przypadku użycia funkcji wbudowanych z funkcją `map`). Aby samodzielnie wykonać tego typu pomiary, warto zapoznać się z funkcjami `time.clock` i `time.time` z modułu wbudowanego `time`, z nowym — od wersji 2.4 — modelem `timeit` oraz z podrozdziałem „Pomiary wydajności implementacji iteratorów” w dalszej części niniejszego rozdziału.

Znaczenie list składanych oraz funkcji map

Oto bardziej realistyczne zastosowanie list składanych i funkcji `map` (problem ten został już rozwiązany z użyciem list składanych w rozdziale 14., ale wracamy do niego, aby uzupełnić go o zastosowanie funkcji `map`). Jak pamiętamy, metoda `readlines` obiektu plikowego zwraca wiersze pliku zakończone znakiem końca wiersza `\n`.

```
>>> open('myfile').readlines()
['aaa\n', 'bbb\n', 'ccc\n']
```

Jeśli chcemy pozbyć się znaku końca wiersza, możemy tego dokonać z użyciem list składanych lub funkcji `map` (funkcja `map` w 3.0 zwraca iterator, zatem musimy przekształcić jej wynik na listę w celu wyświetlenia na ekranie).

```
>>> [line.rstrip() for line in open('myfile').readlines()]
['aaa', 'bbb', 'ccc']

>>> [line.rstrip() for line in open('myfile')]
['aaa', 'bbb', 'ccc']

>>> list(map((lambda line: line.rstrip()), open('myfile')))
['aaa', 'bbb', 'ccc']
```

Ostatnie dwie wersje wykorzystują *iteratory plikowe* (co w praktyce oznacza, że do odczytu zawartości pliku nie musimy jawnie wywoływać jego metod). Wywołanie funkcji `map` wymaga wpisania nieco większej liczby znaków w porównaniu z listą składaną, ale nie musimy bezpośrednio składać wyniku w postaci listy.

Listy składane mogą być również użyte w charakterze operacji wydobywającej kolumny z bazy danych. Standardowe API Pythona do komunikacji z bazami danych SQL zwraca wyniki zapytań w postaci list krotek — lista jest odwzorowaniem tabeli, krotki są wierszami, a elementy krotek są wartościami w kolumnach:

```
listoftuple = [('Teodor', 35, 'dyrektor'), ('Teofil', 40, 'prezes')]
```

Dane z poszczególnych kolumn można odczytać za pomocą pętli `for`, ale `map` i listy składane pozwalają wykonać to w jednym wierszu kodu, a do tego szybciej.

```
>>> [age for (name, age, job) in listoftuple]
[35, 40]

>>> list(map((lambda row: row[1]), listoftuple))
[35, 40]
```

Pierwsze wyrażenie wykorzystuje przypisanie do *krotki zmiennych*, co pozwala rozpakować trzyelementową krotkę na trzy zmienne, drugie podejście wykorzystuje odczyt wartości po indeksie. W Pythonie 2.6 (ale nie w 3.0 — patrz uwaga dotycząca rozpakowywania listy argumentów funkcji w rozdziale 18.) funkcja `map` również może rozpakować listę swoich argumentów.

```
# tylko dla 2.6
>>> list(map((lambda (name, age, job): age), listoftuple))
[35, 40]
```

Informacje na temat API dostępu do baz danych dla Pythona można znaleźć w licznych książkach i innych źródłach dokumentacji Pythona.

Oprócz różnicy między uruchamianiem funkcji a wyrażeniami największa różnica między funkcją `map` a listami składanymi w 3.0 polega na tym, że `map` zwraca *iterator*, generujący wyniki na żądanie. Aby uzyskać tę samą skalę oszczędności pamięci, należałoby listę składaną zapisać jako wyrażenie generatora (co jest jednym z tematów niniejszego rozdziału).

Iteratorów ciąg dalszy — generatory

Python jest obecnie znacznie bardziej wstrzemięźliwy niż dawniej: oferuje mnóstwo narzędzi, które generują wyniki tylko wówczas, gdy te są potrzebne, a nie wszystkie naraz. Do tej kategorii narzędzi należy w szczególności zaliczyć dwie konstrukcje języka odraczące tworzenie wyników tak długo, jak to tylko możliwe:

- *funkcje generatorów*, które tworzy się tak samo jak zwykłe funkcje, ale zamiast instrukcji `return` do zwracania wyników częściowych wykorzystuje się instrukcję `yield`, która zawiesza wykonanie funkcji, zachowując jej stan, co pozwala wznowić ją w przypadku, gdy odbiorca znów poprosi o dane,
- *wyrażenia generatorów* mają składnię zbliżoną do list składanych, ale zwracają obiekt generujący wyniki częściowe na żądanie, zamiast tworzyć naraz całą listę wyników.

Żadna z wymienionych wyżej konstrukcji języka nie tworzy całego wyniku, zatem pozwala oszczędzać pamięć. Również obciążenie procesora związane z generowaniem danych jest rozłożone w czasie. Jak się przekonamy, obydwie te konstrukcje swoją magię implementują z użyciem protokołu iteratorów, który poznaliśmy w rozdziale 14.

Funkcje generatorów — `yield` kontra `return`

W tej części książki dowiedzieliśmy się, w jaki sposób kodować zwykłe funkcje otrzymujące parametry wejściowe i zwracające cały swój wynik w jednej instrukcji `return`. W Pythonie istnieje jednak możliwość napisania funkcji, która może zwracać wartość częściową, zapisując swój stan, aby po wznowieniu kontynuować od tego miejsca, w którym zatrzymała się poprzednio. Tego typu funkcje są znane jako *generatory*, ponieważ generują sekwencję wyników stopniowo, nie w jednej całości.

Funkcje generatorów są pod wieloma względami takie same jak zwykłe funkcje i ich kod definiuje się w instrukcji `def`. Jednak są pisane w taki sposób, aby obsługiwały protokół iteratorów, zatem mogą być używane we wszelkich kontekstach iteracyjnych. Iteratory omówiliśmy już w rozdziale 14., w tym rozdziale zrobimy małe powtórzenie, aby zbadać ich związek z generatorami.

Zawieszanie stanu

W przeciwieństwie do zwykłych funkcji zwracających wartość i kończących tym samym swoje działanie funkcje generatorów potrafią zawieszać swoje działanie i wznowiać je sekwencyjnie w miarę generowania kolejnych wartości. Dzięki temu są często użyteczną alternatywą dla tworzenia wyników w całości, na przykład dla klas, w których programista samodzielnie obsługuje zapisywanie i odtwarzanie stanu. Stan generatora zachowywany między kolejnymi iteracjami zawiera cały zakres nazw lokalnych, zatem również zmienne lokalne zachowują swoje wartości i są dostępne po wznowieniu działania funkcji.

Podstawowa różnica między generatorami a zwykłymi funkcjami polega na tym, że generator zwraca wynik za pomocą instrukcji `yield`, a nie `return`. Instrukcja `yield` zawiesza wykonanie funkcji i zwraca wynik do odbiorcy (kodu wywołującego), ale zachowuje stan wykonania, dzięki czemu funkcja może być wznowiona od tego samego miejsca. Po wznowieniu funkcja

kontynuuje działanie natychmiast po wykonanej poprzednio instrukcji `yield`. Z punktu widzenia funkcji mamy możliwość tworzenia szeregu wartości po jednej zamiast obliczania wszystkich za jednym zamachem i zwracania w całości, na przykład w postaci listy.

Integracja protokołu iteracji

Aby w pełni zrozumieć funkcje generatorów, należy poznać ich związek z protokołem iteracyjnym Pythona. Jak już wiemy, obiekty iterowane definiują metodę `__next__`, która zwraca kolejny element iteracji lub wywołuje wyjątek `StopIteration`. Iteratorem obiektu jest zwracany po wywołaniu funkcji wbudowanej `iter`.

Pętle `for` w Pythonie oraz inne konteksty iteracyjne wykorzystują protokół iteracyjny do odczytu kolejnych elementów sekwencji lub wartości zwracanych przez generator. Jeśli obiekt nie obsługuje protokołu iteracyjnego, iteracja przełącza się w tryb odczytu statycznej sekwencji w kolejności elementów.

W celu obsługi protokołu iteratorów instrukcje `yield` są komplikowane jako *generatory*. Po wywołaniu zwracają obiekt generatora obsługujący interfejs iteracji, który z kolei obsługuje automatycznie utworzoną metodę `__next__` wznowiącą wykonanie funkcji. Funkcje generatorów mogą również obsługiwać instrukcję `return`, która powoduje zakończenie działania funkcji, jak również sygnalizuje zakończenie iteracji. W takim przypadku po zwróceniu wyniku wywoływany jest wyjątek `StopIteration`. Z punktu widzenia kodu wywołującego metodę `__next__` generatora wznowia wykonanie funkcji aż do następnej instrukcji `yield` lub wywołania wyjątku `StopIteration`.

Najważniejszym spostrzeżeniem jest to, że funkcje generatorów kodowane w instrukcji `def` zawierają instrukcję `yield` i automatycznie obsługują protokół iteratorów, dzięki czemu mogą być używane w dowolnym kontekście iteracyjnym do generowania wyników na żądanie.



Jak wspomniałem w rozdziale 14., w Pythonie 2.6 i wcześniejszych obiekty iterowane definiują metody `next`. Dotyczy to również obiektów generatorów omówionych w tym rozdziale. W Pythonie 3.0 nazwa metody `next` została zmieniona na `__next__`. Dodatkowo powstała funkcja wbudowana `next(I)`, która w 3.0 wywołuje metodę `I.__next__()`, a w 2.6 `I.next()`. W wersjach wcześniejszych od 2.6 należy po prostu używać metody `I.next()`, zamiast ręcznie iterować po elementach.

Funkcje generatorów w działaniu

W celu zilustrowania podstaw działania generatorów przeanalizujmy przykładowy kod. Poniższy listing definiuje funkcję generatora obliczającą serię kwadratów kolejnych liczb całkowitych:

```
>>> def gensquares(N):
...     for i in range(N):
...         yield i ** 2          # Kolejne wywołanie wznowi wykonanie od tego miejsca
... 
```

Funkcja zwraca wartość za pomocą instrukcji `yield` i wstrzymuje swoje działanie w pętli `for`. Przy każdym wznowieniu funkcja kontynuuje swoje działanie od miejsca, w którym została wstrzymana (po instrukcji `yield`). Jeśli na przykład użyjemy jej w deklaracji pętli `for`, instrukcja `yield` z każdym przebiegiem pętli zwróci kolejną wartość.

```

>>> for i in gensquares(5):
...     print(i, end=' : ')
...
0 : 1 : 4 : 9 : 16 :
>>>

```

Aby zakończyć generowanie wartości, należy albo jawnie zakończyć funkcję instrukcją `return`, albo pozwolić jej „dobiec do końca” swojego kodu, co również zakończy jej działanie.

Aby sprawdzić, co się dzieje w pętli `for` generatora, wystarczy wywołać jego funkcję bezpośrednio.

```

>>> x = gensquares(4)
>>> x
<generator object at 0x0086C378>

```

Otrzymujemy obiekt generatora obsługujący protokół iteracyjny omówiony w rozdziale 14. Obiekt generatora posiada metodę `__next__`, która uruchamia funkcję lub wznawia jej działanie w miejscu poprzedniego wywołania instrukcji `yield`, a w przypadku zakończenia sekwencji wywołuje wyjątek `StopIteration`. Funkcja wbudowana `next(X)` wywołuje metodę `X.__next__()` obiektu generatora:

```

>>> next(x)                                # W 3.0 równoważne z x.__next__()
0
>>> next(x)                                # W 2.6 można użyć x.next() lub next(x)
1
>>> next(x)
4
>>> next(x)
9
>>> next(x)
Traceback (most recent call last):
...pominęta część wyniku...
StopIteration

```

Jak wiemy z rozdziału 14., pętle `for` (oraz inne konteksty iteracyjne) działają z generatorami właśnie w taki sposób: wywołują ich metodę `__next__` przy każdej iteracji, aż zostanie wywołany wyjątek. Jeśli obiekt poddany iteracji nie obsługuje tego protokołu, pętla `for` wykorzysta zastępco protokół indeksowania po elementach sekwencji.

W powyższym przykładzie listę wartości zwracanych przez `yield` moglibyśmy skonstruować w całości i zwrócić jednorazowo:

```

>>> def buildsquares(n):
...     res = []
...     for i in range(n): res.append(i ** 2)
...     return res
...
>>> for x in buildsquares(5): print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :

```

Do tego typu iteracji moglibyśmy również posłużyć się funkcją `map` lub listą składaną.

```

>>> for x in [n ** 2 for n in range(5)]:
...     print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :
...
>>> for x in map(lambda n: n ** 2, range(5)):
...     print(x, end=' : ')
...
0 : 1 : 4 : 9 : 16 :

```

Generatory mogą jednak okazać się znacznie efektywniejsze z punktu widzenia użycia pamięci i wydajności. Pozwalają funkcjom uniknąć wykonywania całej pracy za jednym zamachem, co jest szczególnie korzystne w przypadku, gdy wynik będzie w całości zajmował dużą ilość pamięci lub jego wygenerowanie zajmie dużo czasu. Generatory pozwalają równomiernie rozłożyć czas generowania danych, pozwalając w tym czasie innemu kodowi przetwarzać już wygenerowane dane cząstkowe.

Co więcej, zaawansowani programiści mogą użyć generatorów jako prostej alternatywy dla mechanizmów do zapisywania stanu między wywołaniami — w przypadku generatorów stan jest zapisywany i wznowiany w sposób automatyczny.² Iteratory implementowane w postaci klas omówimy w szerszym zakresie w części VI.

Rozszerzony protokół funkcji generatorów — send kontra next

W Pythonie 2.5 wprowadzono nową metodę `send` uzupełniającą możliwości protokołu generatorów. Metoda `send` powoduje przejście do kolejnego elementu sekwencji, podobnie jak `__next__`, ale dodatkowo pozwala kodowi wywołującemu skomunikować się z generatorem, aby wpływać na jego działanie.

Z technicznego punktu widzenia od tej pory `yield` staje się wyrażeniem zwracającym dane przekazane w metodzie `send`, a nie instrukcją, jak było dotychczas. Nadal jednak można `yield` wywołać starym sposobem, mamy więc `yield X`, ale też możemy zastosować `A = (yield X)`. Wyrażenie musi zostać ujęte w nawiasy, chyba że po prawej stronie instrukcji `yield` występuje tylko jeden argument. Na przykład `X = yield Y`, ale `X = (yield Y) + 42`.

W przypadku użycia tej dodatkowej składni dane są przesyłane do generatora `G` za pomocą wywołania `G.send(dane)`. Kod generatora jest wznowiany, a wyrażenie `yield` zwraca wartość przekazaną jako `dane`. Jeśli natomiast do wznowienia generatora zostanie użyta metoda `G.__next__()`, `yield` zwróci `None`.

```
>>> def gen():
...     for i in range(10):
...         X = yield i
...         print(X)
...
>>> G = gen()                                # Jako pierwsze wywołanie musi wystąpić next(), co uruchamia generator
0
>>> G.send(77)                               # Wznowienie generatora z przekazaniem wartości
77
1
>>> G.send(88)
88
```

² Co ciekawe, funkcje generatorów można wykorzystać również jako bardzo uproszczony mechanizm *współbieżności*: pozwalały na wykonanie kilku operacji naprzemiennie, z możliwością przełączania kontekstu między nimi w miejscu wywołania instrukcji `yield`. Generatory nie są jednak wątkami, program nadal jest wykonywany w jednym wątku fizycznym. Obsługa wątków jest mechanizmem bardziej ogólnym (procedury mogą działać naprawdę równolegle i zapisywać swoje wyniki w kolejce), ale generatorы pisze się znacznie prościej. Krótkie wprowadzenie do mechanizmów programowania wielowątkowego w Pythonie można znaleźć w drugim przypisie w rozdziale 17. Należy też pamiętać, że przełączanie kontekstu odbywa się w *stałych punktach*, to znaczy w miejscu wywołania instrukcji `yield` oraz w miejscu ponownego wywołania generatora, przez co generatory nie są mechanizmami w pełni współbieżnymi. Ich działanie jest natomiast zbliżone do *współprogramów* (ang. *coroutines*). Jest to formalny koncept programowania równoległego, którego tematyka wykracza jednak poza zakres tego rozdziału.

```
2  
>>> next(G)          # next() oraz X.__next__() przekazują None  
None  
3
```

Metoda `send` może być użyta na przykład do zaprogramowania generatora, który może zostać zatrzymany przez wysłanie specjalnego kodu lub przeprogramowany przez przekazanie nowej pozycji w sekwencji. W Pythonie 2.5 do generatorów dodano jeszcze metodę `throw(typ)`, która powoduje wywołanie w generatorze wyjątku klasy `typ` w miejscu ostatniego wywołania `yield`, oraz metodę `close`, która powoduje wywołanie wyjątku `GeneratorExit` kończącego działanie generatora. Są to zaawansowane możliwości, nad którymi nie będziemy się dłużej zatrzymywać. Więcej informacji na ten temat można znaleźć w standardowym podręczniku Pythona.

Jak już wspomniałem, Python 3.0 oferuje funkcję wbudowaną `next(G)`, będącą odpowiednikiem wywołania metody `G.__next__()`. Pozostałe metody generatorów, jak `send`, muszą być wywoływanie jako metody generatorów (na przykład `G.send(X)`). Ten pozorny brak spójności wyjaśnia się, gdy weźmiemy pod uwagę, że wspomniane dodatkowe metody funkcjonują wyłącznie w ramach obiektów generatorów, podczas gdy metoda `__next__()` jest elementem ogólnego interfejsu iteratorów (zarówno w przypadku wbudowanych typów, jak i klas zdefiniowanych przez użytkownika).

Wyrażenia generatorów — iteratory spotykają złożenia

We wszystkich współczesnych wersjach Pythona pojęcia iteratorów i list składanych zostały uzupełnione o nową cechę języka — *wyrażenia generatorów*. Składniowo wyrażenie generatora wygląda podobnie do zwykłej listy składanej, ale zamiast nawiasów kwadratowych stosuje się nawiasy okrągłe:

```
>>> [x ** 2 for x in range(4)]      # Lista składana: zwraca listę  
[0, 1, 4, 9]  
  
>>> (x ** 2 for x in range(4))      # Wyrażenie generatora: zwraca obiekt iterowany  
<generator object at 0x011DC648>
```

W rzeczywistości wyrażenie listy składanej da efektywnie ten sam wynik co opakowanie wyrażenia generatora w funkcję `list`, która przekształci iterator na listę jego elementów:

```
>>> list(x ** 2 for x in range(4))    # Odpowiednik listy składanej  
[0, 1, 4, 9]
```

Jednak z punktu widzenia mechanizmu wyrażenie generatora to coś zupełnie innego od listy składanej: zamiast budować cały wynik w pamięci, zwraca obiekt obsługujący protokół iteratorów, udostępniając pojedyncze wyniki sekwencji na żądanie:

```
>>> G = (x ** 2 for x in range(4))  
>>> next(G)  
0  
>>> next(G)  
1  
>>> next(G)  
4  
>>> next(G)  
9  
>>> next(G)
```

```
Traceback (most recent call last):  
...pominęty fragment wyniku...  
StopIteration
```

Jednak najczęściej nie mamy okazji ręcznie używać kolejnych wywoływań generatora, ponieważ w kontekście pętli for odbywa się to automatycznie:

```
>>> for num in (x ** 2 for x in range(4)):  
...     print('%s, %s' % (num, num / 2.0))  
...  
0, 0.0  
1, 0.5  
4, 2.0  
9, 4.5
```

Jak już mieliśmy okazję zaobserwować, każdy kontekst iteracyjny działa w taki sposób, w tym funkcje wbudowane sum, map i sorted, listy składane i inne konteksty iteracyjne, które omawialiśmy w rozdziale 14., jak funkcje wbudowane any, all czy list.

W przypadku gdy wyrażenie generatora jest ujęte w nawiasy (na przykład w parametrze wywołania funkcji), nie ma konieczności używania dodatkowej pary nawiasów. Jeśli jednak funkcja wymaga podania dwóch parametrów, nawiasy wokół wyrażenia generatora są konieczne:

```
>>> sum(x ** 2 for x in range(4))  
14  
  
>>> sorted(x ** 2 for x in range(4))  
[0, 1, 4, 9]  
  
>>> sorted((x ** 2 for x in range(4)), reverse=True)  
[9, 4, 1, 0]  
  
>>> import math  
>>> list(map(math.sqrt, (x ** 2 for x in range(4))))  
[0.0, 1.0, 2.0, 3.0]
```

Wyrażenia generatorów powstały głównie z myślą o optymalizacji użycia pamięci, nie wymagają bowiem tworzenia całego wyniku naraz, jak to ma miejsce w przypadku wyrażenia listy składanej. W praktyce mogą jednak działać wolniej, zatem najlepiej jest je stosować w przypadku bardzo dużych list wyników. Jednak dokładniejszą analizę wydajności na razie zostawimy i wróćmy do niej w dalszej części rozdziału.

Funkcje generatorów kontra wyrażenia generatorów

Co interesujące, często tę samą iterację można zapisać w postaci funkcji generatora i wyrażenia generatora. Poniższy przykład tworzy generator powtarzający czterokrotnie każdą literę podanego ciągu znaków:

```
>>> G = (c * 4 for c in 'JAJKO')          # Wyrażenie generatora  
>>> list(G)                            # Wymuszenie wygenerowania wyników w całości  
['JJJJ', 'AAAA', 'JJJJ', 'KKKK', 'OOOO']
```

Analogiczna funkcja iteratora wymaga wpisania nieco większej ilości kodu, ale dzięki temu można w niej zapisać więcej logiki, a w miarę potrzeby więcej danych stanu.

```
>>> def timesfour(S):                      # Funkcja generatora  
...     for c in S:  
...         yield c * 4  
...  
>>> G = timesfour('jajko')                # Iteracja automatyczna  
>>> list(G)  
['jjjj', 'aaaa', 'jjjj', 'kkkk', 'oooo']
```

Wyrażenia i funkcje generatorów obsługują automatyczną i ręczną iterację. W powyższym lisingu wywołanie funkcji list wykorzystuje iterację automatyczną, poniżej znajduje się przykład iteracji ręcznej.

```
>>> G = (c * 4 for c in 'JAJKO')
>>> I = iter(G)                                     # Ręczna iteracja
>>> next(I)
'JJJJ'
>>> next(I)
'AAAA'
>>> G = timesfour('jajko')
>>> I = iter(G)
>>> next(I)
'jjjj'
>>> next(I)
'aaaa'
```

Warto zauważyć, że do każdej iteracji tworzymy nowy generator. Jak się okaże w następnym punkcie, generatory są jednorazowymi iteratorami.

Generatory są jednorazowymi iteratorami

Funkcje i wyrażenia generatorów są jednocześnie iteratorami i obsługują tylko jedną iterację, w przeciwieństwie do wybranych typów wbudowanych, które pozwalają na wykonywanie wielu równoległych iteracji w różnych punktach sekwencji. Weźmy poprzedni przykład wyrażenia generatora. Iteratorem generatora jest sam generator (wywołanie funkcji iter na generatorze zwraca ten sam generator).

```
>>> G = (c * 4 for c in 'JAJKO')
>>> iter(G) is G                                # Iteratorem generatora jest tym samym generatorem: G obsługuje __next__
True
```

Jeśli spróbujemy utworzyć kilka iteratorów z jednego generatora, zauważymy, że wszystkie w danym momencie znajdują się na tej samej pozycji.

```
>>> G = (c * 4 for c in 'JAJKO')  # Utworzenie nowego generatora
>>> I1 = iter(G)                      # Ręczna iteracja
>>> next(I1)
'JJJJ'
>>> next(I1)
'AAAA'
>>> I2 = iter(G)                      # Drugi iterator znajduje się na tej samej pozycji!
>>> next(I2)
'JJJJ'
```

Co więcej, gdy jedna iteracja dojdzie do końca sekwencji, to samo staje się z pozostałymi. Aby zacząć od nowa, należy utworzyć nowy generator:

```
>>> list(I1)                                # Odczyt pozostałych elementów z I1
['KKKK', '0000']
>>> next(I2)                                # Pozostałe iteratory są również opróżnione
StopIteration

>>> I3 = iter(G)                            # Podobnie nowo tworzone iteratory
>>> next(I3)
StopIteration

>>> I3 = iter(c * 4 for c in 'JAJKO') # Nowy generator zaczyna od początku
>>> next(I3)
'JJJJ'
```

Taka sama sytuacja ma miejsce w przypadku funkcji generatorów — poniższy przykład przedstawia funkcję generatora, której iterotor wyczerpuje się po pierwszym przejściu:

```
>>> def timesfour(S):
...     for c in S:
...         yield c * 4
...
>>> G = timesfour('jakko')           # Funkcje generatorów działają w ten sam sposób
>>> iter(G) is G
True
>>> I1, I2 = iter(G), iter(G)
>>> next(I1)
'jjjj'
>>> next(I1)
'aaaa'
>>> next(I2)                      # I2 znajduje się na tej samej pozycji co I1
'jjjj'
```

To znacząca różnica w porównaniu z niektórymi typami wbudowanymi, które obsługują wieleokrotne iteracje i przejścia, a zmiany w ich wartości są odzwierciedlane w aktywnych iteratorach:

```
>>> L = [1, 2, 3, 4]
>>> I1, I2 = iter(L), iter(L)
>>> next(I1)
1
>>> next(I1)
2
>>> next(I2)                      # Listy obsługują wielokrotną iterację
1
>>> del L[2:]                     # Zmiany w liście są odzwierciedlane w iteratorach
>>> next(I1)
StopIteration
```

Gdy będziemy tworzyć własne iteratory w części VI, przekonamy się, że to od programisty zależy, ile równoległych iteracji chce obsłużyć, jeśli w ogóle.

Emulacja funkcji zip i map za pomocą narzędzi iteracyjnych

Aby zobaczyć możliwości narzędzi iteracyjnych, przyjrzyjmy się bardziej zaawansowanym przykładom użycia. Uzbrojeni w wiedzę o listach składanych, generatorach i innych narzędziach iteracyjnych możemy pokusić się o zaemulowanie niektórych rozszerzeń funkcyjnych Pythona. To zadanie okaże się o tyle proste co pouczające.

Mieliśmy już okazję zaobserwować, w jaki sposób funkcje `zip` i `map` przetwarzają obiekty iterowane z użyciem funkcji wykonywanych na ich elementach. W przypadku zastosowania kilku argumentów sekwencyjnych funkcja `map` wykonuje funkcje wieloargumentowe na kolejnych elementach każdej z list, natomiast `zip` łączy w nowe sekwencje odpowiadające sobie elementy każdej z list.

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>> list(zip(S1, S2))            # zip łączy w pary elementy iteratorów
[('a', 'x'), ('b', 'y'), ('c', 'z')]
# zip łączy elementy i przycina wynik do najkrótszej sekwencji
>>> list(zip([2, 1, 0, 1, 2]))      # Jedna sekwencja: zwraca krotki 1-elementowe
[(2,), (1,), (0,), (1,), (2,)]
```

```

>>> list(zip([1, 2, 3], [2, 3, 4, 5]))           # N sekwencji: zwraca krotki N-elementowe
[(1, 2), (2, 3), (3, 4)]
# map przekazuje połączone elementy do funkcji, przycina do najkrótszej sekwencji
>>> list(map(abs, [2, 1, 0, 1, 2]))            # Jedna sekwencja: wywołanie funkcji 1-argumentowej
[2, 1, 0, 1, 2]
>>> list(map(pow, [1, 2, 3], [2, 3, 4, 5]))    # N sekwencji: wywołanie funkcji N-argumentowej
[1, 8, 81]

```

Choć mechanizmy te powstały w nieco innym celu, to warto zwrócić uwagę na analogię, z jaką działają funkcje `map` i `zip` na większej liczbie sekwencji, ponieważ tę właściwość wykorzystamy w kolejnym przykładzie.

Tworzymy własną implementację funkcji `map`

Choć funkcje wbudowane `map` i `zip` są szybkie i wygodne w użyciu, zawsze istnieje możliwość zaemulowania ich na własną rękę. W poprzednim rozdziale na przykład mieliśmy okazję stworzyć funkcję emulującą funkcję wbudowaną `map`, obsługującą tylko jeden argument sekwencji. Uzupełnienie tej implementacji o obsługę większej liczby sekwencji nie stanowi jednak problemu.

```

# map(func, seqs...): emulacja z użyciem funkcji zip

def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        res.append(func(*args))
    return res

print(mymap(abs, [2, 1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))

```

Ta wersja intensywnie wykorzystuje specjalną składnię `*args` służącą do przekazywania argumentów między wywoływanymi funkcjami. Funkcja rozpakowuje swoje argumenty (będące sekwencjami, a właściwie obiektami iterowanymi) za pomocą funkcji `zip`, łącząc je w nowe sekwencje, po czym wywołuje funkcje przekazaną w pierwszym argumentem z użyciem tych nowych sekwencji jako argumentów. Wykorzystujemy zaobserwowany wyżej fakt, że operacja `zip` jest w zasadzie zagnieżdżoną operacją `map`. Test poniżej definicji funkcji sprawdza działanie naszego generatora dla dwóch sekwencji (wyniki są identyczne z tymi, które zwróciłaby wbudowana funkcja `map`).

```
[2, 1, 0, 1, 2]
[1, 8, 81]
```

W rzeczywistości ta wersja wykorzystuje klasyczny wzorzec znany z *list składanych*, budując listę wyników w pętli `for`. Naszą funkcję `map` możemy zatem zbudować w sposób bardziej zwarty, wykorzystując jednowierszowe wyrażenie listy składanej.

```

# Użycie listy składanej

def mymap(func, *seqs):
    return [func(*args) for args in zip(*seqs)]

print(mymap(abs, [2, 1, 0, 1, 2]))
print(mymap(pow, [1, 2, 3], [2, 3, 4, 5]))

```

Uruchomione testy zwracają taki sam wynik jak poprzednio, ale kod jest krótszy i prawdopodobnie będzie szybszy (więcej informacji na temat pomiarów wydajności przedstawię w punkcie „Pomiary wydajności implementacji iteratorów” w dalszej części rozdziału). Obie z przedsta-

wionych wersji funkcji `mymap` składają wynik w całości, co może mieć konsekwencje w postaci większego zapotrzebowania na pamięć. Dzięki możliwościom *funkcji i wyrażeń generatorów* możemy w prosty sposób przepisać te funkcje tak, aby zwracały wyniki na żądanie.

Użycie generatorów: `yield` i (...)

```
def mymap(func, *seqs):
    res = []
    for args in zip(*seqs):
        yield func(*args)

def mymap(func, *seqs):
    return (func(*args) for args in zip(*seqs))
```

Te wersje nie tworzą całych wyników naraz, ale zwracają generatory obsługujące protokół iteracyjny: pierwsza wersja zwraca wyniki za pomocą instrukcji `yield`, a druga jest jej funkcjonalnym odpowiednikiem dzięki wyrażeniu generatora. Obie wersje zwrócią takie same wyniki, jeśli przekształcimy je na listy za pomocą funkcji `list`.

```
print(list(mymap(abs, [2, 1, 0, 1, 2])))
print(list(mymap(pow, [1, 2, 3], [2, 3, 4, 5])))
```

Jednak w tym przypadku różnica jest taka, że generatory nie wykonują żadnej pracy do momentu, gdy zostaje na nich wywołana funkcja `list` aktywująca protokół iteracyjny. Generatory zwrócone z tych funkcji, jak również wersja dla Pythona 3.0 wykorzystująca funkcję `zip`, generują wyniki na żądanie.

Własna wersja funkcji `zip(...)` i `map(None, ...)`

Oczywiście spora część magii powyższych przykładów dzieje się dzięki zastosowaniu funkcji wbudowanej `zip` łączącej elementy z kilku sekwencji. Warto również zauważyć, że nasze implementacje funkcji `map` w rzeczywistości kopiąją zachowanie funkcji `map` z Pythona 3.0: przycinają wynik do długości najkrótszej sekwencji, zamiast uzupełniać brakujące elementy wartościami `None`, jak to się dzieje w Pythonie 2.X.

```
C:\misc> c:\python26\python
>>> map(None, [1, 2, 3], [2, 3, 4, 5])
[(1, 2), (2, 3), (3, 4), (None, 5)]
>>> map(None, 'abc', 'xyz123')
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
```

Wykorzystując narzędzia iteracyjne, możemy zaimplementować własne wersje funkcji `zip` przycinającej wynik do najkrótszej sekwencji (dla 2.6) lub funkcję `map` uzupełniającą brakujące elementy sekwencji wartościami `None` (dla 3.0). Jak się okazuje, funkcje te będą bardzo do siebie podobne:

Własne implementacje funkcji `zip(seqs...)` oraz `map(None, seqs...)` w wersji 2.6

```
def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    res = []
    while all(seqs):
        res.append(tuple(S.pop(0) for S in seqs))
    return res

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    res = []
    while any(seqs):
```

```

    res.append(tuple((S.pop(0) if S else pad) for S in seqs))
    return res

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))

```

Obie prezentowane funkcje współpracują z dowolnym obiektem iterowanym, ponieważ przekształcają swoje argumenty na listę, wymuszając wykonanie generatora (czyli jako argumenty mogą być podane pliki lub inne sekwencje, jak ciągi znaków). Należy zwrócić uwagę na użycie funkcji `all` i `any`: zwracają wartość `True`, odpowiednio, jeśli wszystkie lub dowolny element obiektu iterowanego ma wartość `True` (lub jest niepusty). Te funkcje służą do zatrzymania pętli, jeśli dowolny lub wszystkie elementy sekwencji zostaną usunięte.

Warto również zwrócić uwagę na argument *mogący być tylko słowem kluczowym* `pad`, dostępny wyłącznie w wersji 3.0. W przeciwieństwie do standardowej funkcji `zip` z 2.6 nasza wersja akceptuje dowolne obiekty uzupełniające sekwencje o mniejszej długości (jeśli ktoś chciałby użyć Pythona 2.6 z argumentem typu `**kargs`, szczegóły tej techniki znajdzie w rozdziale 18.). Po wywołaniu tych funkcji zostają wyświetlane następujące wyniki (`zip` i dwa wyniki `map` z uzupełnianiem):

```

[('a', 'x'), ('b', 'y'), ('c', 'z')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
[('a', 'x'), ('b', 'y'), ('c', 'z'), (99, '1'), (99, '2'), (99, '3')]

```

Omawiane funkcje nie mogą być w prosty sposób zaimplementowane w formie list składanych, ponieważ ich pętle `for` są zbyt specyficzne. Jednak, podobnie jak poprzednio, nasze wersje funkcji `zip` i `map`, które obecnie zwracają listę wyników w całości, mogą być w prosty sposób przekształcone w *generatory* (przy wykorzystaniu instrukcji `yield`), co spowoduje, że każdy element wyniku będzie generowany na żądanie. Efekt będzie zbliżony do poprzedniego, z tą różnicą, że w celu wyświetlenia wyników musimy przekształcić generator na listę:

Użycie generatorów: yield

```

def myzip(*seqs):
    seqs = [list(S) for S in seqs]
    while all(seqs):
        yield tuple(S.pop(0) for S in seqs)

def mymapPad(*seqs, pad=None):
    seqs = [list(S) for S in seqs]
    while any(seqs):
        yield tuple((S.pop(0) if S else pad) for S in seqs)

S1, S2 = 'abc', 'xyz123'
print(list(myzip(S1, S2)))
print(list(mymapPad(S1, S2)))
print(list(mymapPad(S1, S2, pad=99)))

```

Na koniec przedstawiamy alternatywną implementację naszych emulatorów funkcji `zip` i `map`: zamiast usuwać elementy z list za pomocą metody `pop`, funkcje te wykorzystują *długości list*. Dzięki tym wartościom łatwo jest zaprogramować osadzone listy składane budujące listy wyników z użyciem indeksów do kolejnych elementów list źródłowych.

Alternatywna implementacja wykorzystująca długości

```

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)

```

```

        return [tuple(S[i] for S in seqs) for i in range(minlen)]

def mymapPad(*seqs, pad=None):
    maxlen = max(len(S) for S in seqs)
    index  = range(maxlen)
    return [tuple((S[i] if len(S) > i else pad) for S in seqs) for i in index]

S1, S2 = 'abc', 'xyz123'
print(myzip(S1, S2))
print(mymapPad(S1, S2))
print(mymapPad(S1, S2, pad=99))

```

Z powodu wykorzystania długości i dostępu do elementów po indeksach funkcje te zakładają, że mają do czynienia z sekwencjami lub z obiektami o zbliżonych cechach, ale nie mogą to być zupełnie dowolne obiekty iterowane. Zewnętrzna lista składana iteruje po zakresie indeksów argumentów, wewnętrzna (przekształcona na krotkę) ponownie iteruje po przekazanych argumentach, wydobywając elementy na odpowiadających sobie indeksach. Wyniki tych funkcji są takie same jak poprzedniej implementacji.

W tym przykładzie najbardziej rzuca się w oczy powszechnie wykorzystanie generatorów i iteratorów. Argumenty przekazane do funkcji `min` i `max` są wyrażeniami generatorów, które są wykonywane do końca, zanim zostaną uruchomione zagnieżdżone listy składane. Co więcej, zagnieżdżone listy składane wykorzystują dwa poziomy opóźnionego wyliczenia: w Pythonie 3.0 funkcja wbudowana `range` jest obiektem iterowanym, jest nim również wyrażenie generatora przekształcone na krotkę.

W rzeczywistości w tych funkcjach nie dochodzi do jakiegokolwiek wyliczenia wartości aż do momentu, gdy nawiasy kwadratowe wyrażeń list składanych wymuszą wygenerowanie listy: efektywnie powodują pełne wywołanie sekwencji generatorów. Aby również to wyrażenie przekształcić na generator, zamiast budować listę w całości, wystarczy nawiasy kwadratowe zastąpić okrągłymi. Oto odpowiedni fragment naszej wersji funkcji `zip`:

```

# Użycie generatorów: (...)

def myzip(*seqs):
    minlen = min(len(S) for S in seqs)
    return (tuple(S[i] for S in seqs) for i in range(minlen))

print(list(myzip(S1, S2)))

```

W tym przypadku w celu wyświetlenia wyników na ekranie musimy posłużyć się przekształceniem wyniku na listę. Zachęcam do własnych eksperymentów z tymi funkcjami w celu pogłębiania wiedzy. Sugestie co do dalszych sposobów zmodyfikowania tych funkcji można znaleźć w ramce „Jednorazowe iteracje”.

Generowanie wyników we wbudowanych typach i klasach

Spędziliśmy nieco czasu na pisaniu własnych generatorów wartości, ale należy pamiętać, że wiele typów wbudowanych również zachowuje się w taki sposób. Jak widzieliśmy na przykład w rozdziale 14., słowniki posiadają iteratory generujące klucze przy każdej iteracji:

```

>>> D = {'a':1, 'b':2, 'c':3}
>>> x = iter(D)
>>> next(x)
'a'
>>> next(x)
'c'

```

Jednorazowe iteracje

W rozdziale 14. poznaliśmy funkcje wbudowane obsługujące tylko jedno przejście iteracyjne (na przykład `map`), po którym wynik jest pusty. Obiecałem, że w dalszej części książki zademonstruję przykład wyjaśniający, dlaczego to zjawisko jest subtelne i istotne w praktyce. W tym miejscu książki mamy już za sobą solidne podstawy zagadnień związanych z iteracjami, nadszedł więc dobry moment, by spełnić obietnicę. Weźmy na przykład następującą alternatywę dla przykładów emulacji funkcji `zip` przedstawionych w tym rozdziale (przykład zaczerpnięty z jednego z podręczników Pythona):

```
def myzip(*args):
    iters = map(ite, args)
    while iters:
        res = [next(i) for i in iters]
        yield tuple(res)
```

Dzięki temu, że ten kod wykorzystuje funkcję `next`, może współpracować z dowolnym obiektem iterowanym. Warto zwrócić uwagę, że nie ma sensu przechwytywanie wyjątku `StopIteration` wywoływanego przez funkcję `next`: jej wywołanie zostanie przekazane przez funkcję, co w efekcie spowoduje zakończenie pracy generatora — tak samo, jak byśmy zwróciли wartość za pomocą instrukcji `return`. Instrukcja `while iters:` wystarcza do realizacji pętli po elementach argumentów, pod warunkiem że do funkcji został przekazany co najmniej jeden. Zapobiega też efektowi nieskończonej pętli, jeśli nie został podany żaden argument (wyrażenie listy składanej zwróci pustą listę).

Powyższy kod bez problemu działa w Pythonie 2.6:

```
>>> list(myzip('abc', 'lmnop'))
[('a', 'l'), ('b', 'm'), ('c', 'n')]
```

Jednak w Pythonie 3.0 wpadamy w pętlę nieskończoną, ponieważ funkcja `map` w 3.0 zwraca generator jednorazowy zamiast listy, jak to ma miejsce w 2.6. W 3.0 po pierwszym przejściu iteratora przez obiekt `iters` ten ostatni będzie ciągle pusty (a `res` będzie miał wartość `[]`). Aby funkcja zadziałała w 3.0, musimy posłużyć się funkcją wbudowaną `list` do utworzenia obiektu obsługującego wielokrotną iterację:

```
def myzip(*args):
    iters = list(map(ite, args))
    ...dalszy ciąg taki sam...
```

Warto prześledzić ten kod, aby przekonać się, w jaki sposób działa. Morał z tej opowieści: w 3.0 przekształcenie wyników funkcji `map` na listę służy nie tylko do wyświetlania na ekranie!

Klucze słowników, podobnie jak generatory tworzone przez użytkownika, mogą być iterowane wielokrotnie, zarówno ręcznie, jak i w wielopoziomowych kontekstach iteracyjnych, jak pętle `for`, wywołania `map`, listy składane i wiele innych, które poznaliśmy w rozdziale 14.

```
>>> for key in D:
...     print(key, D[key])
...
a 1
c 3
b 2
```

Jak również mieliśmy okazję zaobserwować, w przypadku iteratorów plików Python po prostu wczytuje kolejne wiersze:

```
>>> for line in open('temp.txt'):
...     print(line, end='')
...
To tylko
rana powierzchnia.
```

Wbudowane iteratory typów są związane z konkretnym typem wyników, ale cała koncepcja jest zbliżona do tego, co zaprogramowaliśmy z użyciem wyrażeń i funkcji. Konteksty iteracyjne, jak pętle, akceptują dowolny obiekt iterowany, czy to zdefiniowany przez użytkownika, czy też wbudowany.

Możliwe jest też zaimplementowanie własnych obiektów generatorów, spełniających wymagania protokołu iteracyjnego, ale zagadnienie to wykracza poza zakres tematyki tego rozdziału. Tego typu klasy muszą definiować specjalną metodę `_iter_` wywoływaną przez funkcję wbudowaną `iter`, która musi zwracać obiekt implementujący metodę `_next_` wywoływaną przez funkcję wbudowaną `next` (jako mechanizm zastępczy protokołu iteracyjnego stosowana jest też metoda wydobywania elementów po indeksie `_getitem_`).

Obiekty instancji tworzone przez takie klasy są obiektami iterowanymi i mogą być wykorzystywane jako argumenty pętli `for` oraz w innych kontekstach iteracyjnych. Jednak dzięki klasom mamy dostęp do bogatszej grupy narzędzi, pozwalającej na definiowanie zaawansowanej logiki i struktur danych, nieoferowanych przez pozostałe konstrukcje języka związane z generatorami.

Tematyka iteratorów nie zostanie wyczerpana aż do zgłębienia ich powiązania z klasami. Na razie jednak jesteśmy zmuszeni odłożyć ten temat do rozdziału 29., w którym omówimy iteratory implementowane w postaci klas.

Podsumowanie obiektów składanych w 3.0

W tym rozdziale skupialiśmy się na listach składanych i generatorach, jednak należy pamiętać, że istnieją jeszcze dwie konstrukcje złożień: zbiory i słowniki składane, dostępne od Pythona 3.0. Mieliśmy okazję spotkać się z nimi w rozdziałach 5. i 8., ale teraz, wzbogaceni o większą wiedzę na temat obiektów składanych i generatorów, możemy w pełni docenić możliwości tych rozszerzeń Pythona 3.0.

- W przypadku *zbiorów* nowy literal `{1, 2, 3}` jest równoważny wyrażeniu `set([1, 2, 3])`, a nowa składnia zbiorów składanych `{f(x) for x in S if P(x)}` działa tak samo jak wyrażenie generatora `set(f(x) for x in S if P(x))`, gdzie `f(x)` jest dowolnym wyrażeniem.
- W przypadku *słowników* nowa składnia słowników składanych `{key: val for (key, val) in zip(keys, vals)}` działa tak samo jak wyrażenie `dict(zip(keys, vals))`, a `{x: f(x) for x in items}` działa jak wyrażenie generatora `dict((x, f(x)) for x in items)`.

Poniższy listing prezentuje podsumowanie wszystkich wyrażeń obiektów składanych w Pythonie 3.0. Ostatnie dwie konstrukcje są nowe i nie są dostępne w 2.6.

```
>>> [x * x for x in range(10)]      # Lista składana: zwraca listę
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81] # odpowiednik list (wyrażenie generatora)

>>> (x * x for x in range(10))      # Wyrażenie generatora: zwraca elementy na żądanie
<generator object at 0x009E7328>    # W niektórych kontekstach nawiasy są opcjonalne
```

```
>>> {x * x for x in range(10)}          # Zbiór składany, nowość w 3.0
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36} # {x, y} to literal zbioru w 3.0

>>> {x: x * x for x in range(10)}      # Słownik składany, nowość w 3.0
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Zrozumieć zbiory i słowniki składane

W pewnym sensie zbiory i słowniki składane stanowią jedynie składniowe uproszczenie istniejących konstrukcji, które można zbudować, na przykład przekształcając wyrażenie generatora na odpowiedni typ. Zarówno zbiory, jak i słowniki składane mogą pracować z dowolnym obiektem iterowanym, więc również generator sprawdzi się w tej roli:

```
>>> {x * x for x in range(10)}          # Zbiór składany
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> set(x * x for x in range(10))        # Generator przekształcony na zbiór
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> {x: x * x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
>>> dict((x, x * x) for x in range(10))
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Podobnie jak w przypadku listy składanej, wynik możemy wygenerować za pomocą odpowiedniej pętli. Oto rozwiniecia powyższych obiektów składanych w postaci zwykłej instrukcji pętli:

```
>>> res = set()
>>> for x in range(10):                  # Odpowiednik zbioru składanego
...     res.add(x * x)
...
>>> res
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> res = {}
>>> for x in range(10):                  # Odpowiednik słownika składanego
...     res[x] = x * x
...
>>> res
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Warto zwrócić uwagę, że mimo iż obydwie formy akceptują iteratory, nie zwracają ich jednak: obie konstrukcje zwracają w całości gotowy obiekt odpowiedniego typu. Jeśli ktoś potrzebuje tworzyć klucze i wartości na żądanie, lepiej nada się do tego wyrażenie generatora:

```
>>> G = ((x, x * x) for x in range(10))
>>> next(G)
(0, 0)
>>> next(G)
(1, 1)
```

Rozszerzona składnia zbiorów i słowników składanych

Zbiory i słowniki składane, podobnie jak listy składane i wyrażenia generatorów, obsługują wyrażenia warunkowe pozwalające odfiltrować z wyniku wybrane wartości sekwencji wejściowej. Poniżej znajduje się kod zwracający kwadraty parzystych elementów zakresu (czyli elementy niezwracające reszty z dzielenia przez dwa):

```

>>> [x * x for x in range(10) if x % 2 == 0]          # Listy zachowują kolejność elementów
[0, 4, 16, 36, 64]
>>> {x * x for x in range(10) if x % 2 == 0}        # Zbiory nie
{0, 16, 4, 64, 36}
>>> {x: x * x for x in range(10) if x % 2 == 0}      # Ani klucze słowników
{0: 0, 8: 64, 2: 4, 4: 16, 6: 36}

```

Również bez problemu działają zagnieździone pętle `for`, choć natura zbiorów i słowników (unikalność wartości zbioru i kluczy słownika) może powodować, że wynik będzie mniej czytelny:

```

>>> [x + y for x in [1, 2, 3] for y in [4, 5, 6]]    # Listy zachowują duplikaty
[5, 6, 7, 6, 7, 8, 7, 8, 9]
>>> {x + y for x in [1, 2, 3] for y in [4, 5, 6]}   # Zbiory nie
{8, 9, 5, 6, 7}
>>> {x: y for x in [1, 2, 3] for y in [4, 5, 6]}     # Ani klucze słowników
{1: 6, 2: 6, 3: 6}

```

Zbiory i słowniki składane, podobnie jak listy składane i wyrażenia generatorów, mogą wykorzystywać dowolne iteratory: listy, ciągi znaków, pliki, zakresy i dosłownie wszystko, co obsługuje protokół iteracyjny.

```

>>> {x + y for x in 'ab' for y in 'cd'}
{'bd', 'ac', 'ad', 'bc'}

>>> {x + y: (ord(x), ord(y)) for x in 'ab' for y in 'cd'}
{'bd': (98, 100), 'ac': (97, 99), 'ad': (97, 100), 'bc': (98, 99)}

>>> {k * 2 for k in ['mielonka', 'szynka', 'kiełbasa'] if k[1] == 'i'}
{'mielonkamielonka', 'kiełbasakiełbasa'}

>>> {k.upper(): k * 2 for k in ['mielonka', 'szynka', 'kiełbasa'] if k[1] == 'i'}
{'KIEŁBASA': 'kiełbasakiełbasa', 'MIELONKA': 'mielonkamielonka'}

```

Czytelników zainteresowanych dalszymi szczegółami zachęcam do własnych eksperymentów. Zbiory i słowniki składane mogą w niektórych przypadkach działać wydajniej w porównaniu z analogicznymi generatorami lub pętlami `for`, ale nie jest to zasadą. Aby zyskać pewność, należy po prostu zmierzyć wydajność każdej konstrukcji — co sprowadza nas do następnego tematu.

Pomiary wydajności implementacji iteratorów

W tej książce poznaliśmy sporo alternatywnych metod implementacji iteratorów. W ramach podsumowania przyjrzymy się nieco bardziej rozbudowanemu przykładowi, wykorzystującemu sporo z przyswojonej dotychczas wiedzy o iteratorach i funkcjach.

Kilka razy wspomniałem o tym, że listy składane są wydajniejsze od pętli `for` i że funkcja `map` w zależności od pewnych czynników może mieć lepszą lub gorszą wydajność od pętli `for`. Wyrażenia generatorów opisywane w poprzednim podrozdziale są nieco wolniejsze w działaniu od list składanych, ale za to pozwalają zminimalizować zużycie pamięci.

To są pewne uogólnienia, które nadal są prawdziwe, jednak szczegóły mogą się różnić w zależności od wersji Pythona, ponieważ język ten jest poddawany systematycznym pracom optymalizacyjnym. Jeśli chcemy mieć pewność, musimy napisać narzędzie pomiarowe i samodzielnie zmierzyć wydajność na własnym komputerze i na posiadanej wersji Pythona.

Moduł mytimer

Na szczęście Python ułatwia i to zadanie. Aby zmierzyć wydajność iteratorów, zacznijmy od napisania prostej, uogólnionej funkcji mierzącej czas wykonania. Funkcję zapiszemy w module *mytimer.py*, aby była dostępna w różnych programach, w których zechcemy jej użyć.

Plik mytimer.py

```
import time
reps = 1000
repslist = range(reps)

def timer(func, *pargs, **kargs):
    start = time.clock()
    for i in repslist:
        ret = func(*pargs, **kargs)
    elapsed = time.clock() - start
    return (elapsed, ret)
```

Funkcja `timer` mierzy czas wykonania dowolnej funkcji z dowolną liczbą argumentów pozycyjnych i argumentów ze słowami kluczowymi. Funkcja odczytuje czas uruchomienia, wywołuje funkcję, a po jej zakończeniu odejmuje czas zakończenia od czasu rozpoczęcia działania. Warto zwrócić uwagę na następujące szczegóły:

- moduł standardowy `time` udostępnia funkcję `time` zwracającą aktualny czas, ale jej precyza jest zależna od platformy systemowej; w systemie Windows to wywołanie zwraca wyniki z dokładnością mikrosekundową, zatem jest bardzo dokładne,
- wywołanie funkcji `range` jest wyłączone z pętli pomiarowej, dzięki czemu koszt wbudowania zakresu (Python 2.6) nie jest wliczany do pomiaru; w 3.0 funkcja `range` zwraca generator, więc koszt jest prawie zerowy, dzięki czemu ten zabieg nie jest konieczny (ale nie przeszkadza),
- licznik `reps` jest zmienną globalną, która może być zmodyfikowana przez kod importujący moduł `mytimer.reps = N`.

Po zakończeniu działania zwracana jest krotka zawierająca całkowity czas wszystkich wywołań wraz z wynikiem działania mierzonej funkcji, aby funkcja wywołująca mogła zweryfikować poprawność jej działania.

Dzięki temu, że funkcja mierząca wydajność została zapisana w module, staje się użytecznym narzędziem, które można wykorzystać w dowolnym programie. Więcej informacji na temat modułów i importowania przekażę w następnej części książki, w tym momencie wystarczy zaimportować moduł `mytimer` i wywołać funkcję `timer`, przekazując jako parametr funkcję, której wydajność chcemy zmierzyć (rozdział 3. można wykorzystać w charakterze powtórką z wiedzy o importowaniu nazw z modułów).

Skrypt mierzący wydajność

Aby zmierzyć wydajność narzędzia iteracyjnego, wystarczy uruchomić skrypt prezentowany na poniższym listingu. Skrypt wykorzystuje moduł `mytimer` napisany wcześniej i mierzy czasy wykonywania narzędzi konstruujących listy, które poznaliśmy dotychczas.

```

# Plik timeseqs.py

import sys, mytimer
reps = 10000
repslist = range(reps)

def forLoop():
    res = []
    for x in repslist:
        res.append(abs(x))
    return res

def listComp():
    return [abs(x) for x in repslist]

def mapCall():
    return list(map(abs, repslist)) # Zbudowanie listy powtórzeń w 2.6

def genExpr():
    return list(abs(x) for x in repslist) # Przekształcenie na listę na potrzeby 3.0

def genFunc():
    def gen():
        for x in repslist:
            yield abs(x)
    return list(gen())

print(sys.version)
for test in (forLoop, listComp, mapCall, genExpr, genFunc):
    elapsed, result = mytimer.timer(test)
    print ('-' * 33)
    print ('%-9s: %.5f => [%s...%s]' % (test.__name__, elapsed, result[0], result[-1]))

```

Skrypt testuje pięć alternatywnych implementacji budowania list i, jak widać w kodzie, w każdej z tych metod buduje dziesięć milionów elementów w seriach po tysiąc powtórzeń.

Należy zwrócić uwagę na to, że wyniki wyrażeń generatorów są przekształcane na listę za pomocą wbudowanej funkcji `list`. Gdybyśmy pominęli ten etap, generator nie wykonaliby swojej pracy, ponieważ elementy nie zostałyby wygenerowane. W Pythonie 3.0 (tylko) należy wykonać to samo dla wyników funkcji `map`, ponieważ w tej wersji Pythona funkcja `map` zwraca iterator, nie listę. Warto również zwrócić uwagę na to, że funkcja wypisująca wyniki pomiarów wykorzystuje atrybut `__name__` do wyświetlenia nazwy funkcji.³

Pomiary czasu

Gdy przedstawiony wyżej skrypt uruchomilem w Pythonie 3.0 na moim laptopie z systemem Windows Vista, okazało się, że funkcja `map` jest nieco szybsza od list składanych, obie te metody są szybsze od pętli `for`, a wyrażenia generatorów i funkcje plasują się w środku stawki.

```

C:\misc> c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
-----
forLoop : 2.64441 => [0...9999]
-----

```

³ Warto również zwrócić uwagę na sposób przekazania mierzonej funkcji do funkcji pomiarowej. W rozdziałach 38. i 39. dowiemy się o dekoratorach: jest to sposób, w jaki z reguły wywołuje się w Pythonie tego typu operacje na funkcjach.

```
listComp : 1.60110 => [0...9999]
-----
mapCall : 1.41977 => [0...9999]
-----
genExpr : 2.21758 => [0...9999]
-----
genFunc : 2.18696 => [0...9999]
```

Gdy przyjrzymy się tym wynikom i kodowi skryptu pomiarowego, zauważymy, że wyrażenia generatorów działają wolniej od list składanych. Choć przekształcenie wyrażenia generatorów w listę za pomocą funkcji `list` powoduje, że staje się ono efektywnym odpowiednikiem listy składanej, to wewnętrzna implementacja każdej z tych technologii jest na tyle odmienna, że występują różnice w wydajności działania (należy jednak pamiętać, że w przypadku testu generatora do wyniku jest również doliczane wywołanie funkcji `list`).

```
return [abs(x) for x in range(size)]      # 1,6 sekundy
return list(abs(x) for x in range(size))    # 2,2 sekundy: wewnętrzne różnice implementacji
```

Co interesujące, gdy ten sam kod uruchomiłem w Pythonie 2.5 w systemie Windows XP (pisząc poprzednie wydanie tej książki), wyniki były dość zbliżone: listy składane okazały się prawie dwukrotnie wydajniejsze od pętli `for`, zaś funkcja `map` była nieco szybsza od list składanych w przypadku użycia funkcji wbudowanych, jak `abs`. Nie mierzyłem wówczas wydajności funkcji generatorów, a format wyniku był nieco inny:

```
2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
forStatement      => 6.10899996758
listComprehension => 3.51499986649
mapFunction       => 2.73399996758
generatorExpression => 4.11600017548
```

Inne czasy wykonania testów w Pythonie 2.5 w porównaniu do tych wykonanych w Pythonie 3.0 wynikają z tego, że zostały wywołane na innym laptopie, a nie z tego, że Python 3.0 jest szybszy. W rzeczywistości wyniki tego samego skryptu dla Pythona 2.6 są nieco lepsze niż dla 3.0 na tej samej maszynie. Należy jedynie usunąć wywołanie funkcji `list` na wynikach funkcji `map`, ponieważ powoduje to dwukrotne zbudowanie listy (zachęcam do przeprowadzenia samodzielnnej próby porównawczej).

Zauważmy jednak, co się stanie, gdy w budowanych sekwencjach wywołamy jakieś rzeczywiste operacje, jak dodawanie, zamiast trywialnej funkcji wbudowanej `abs` (pominąłem fragmenty, w których skrypt nie różnił się od poprzedniego).

```
# File timeseqs.py
...
...
def forLoop():
    res = []
    for x in repslist:
        res.append(x + 10)
    return res

def listComp():
    return [x + 10 for x in repslist]

def mapCall():
    return list(map((lambda x: x + 10), repslist))           # tylko dla 3.0

def genExpr():
    return list(x + 10 for x in repslist)                      # 2.6 + 3.0

def genFunc():
```

```

def gen():
    for x in repslist:
        yield x + 10
    return list(gen())
...

```

W tym przypadku konieczność wywołania funkcji zdefiniowanej przez użytkownika w funkcji `map` powoduje, że ta metoda staje się wolniejsza od pętli `for`, mimo że pętla wymaga zapisania większej ilości kodu. W Pythonie 3.0 uzyskałem następujące wyniki:

```

C:\misc> c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
-----
forLoop   : 2.60754 => [10...10009]
-----
listComp   : 1.57585 => [10...10009]
-----
mapCall   : 3.10276 => [10...10009]
-----
genExpr   : 1.96482 => [10...10009]
-----
genFunc   : 1.95340 => [10...10009]

```

Wyniki dla Pythona 2.5 na słabszym komputerze były zbliżone (dwukrotnie mniejsza wydajność ponownie wynika z faktu użycia słabszej maszyny):

```

2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
forStatement      => 5.25699996948
listComprehension => 2.68400001526
mapFunction       => 5.96900010109
generatorExpression => 3.37400007248

```

Z powodu znaczących wewnętrznych optymalizacji interpretera tego typu porównania wydajności algorytmów są zadaniem dość niewdzięcznym. W zasadzie niemożliwe jest odgadnięcie, która metoda okaże się szybsza. Najlepszym rozwiązaniem jest zaimplementowanie wszystkich alternatyw i dokonanie pomiarów na komputerze, na którym kod będzie wykonywany, z taką wersją Pythona, z jaką będzie pracował na co dzień. I w takim przypadku jedyną rzeczą, jaką można stwierdzić na pewno, jest to, że na przykład „w wersji Pythona X funkcja `map` zwalnia o połowę w przypadku wykorzystania funkcji zdefiniowanej przez użytkownika, a listy składane są w tym teście najwydajniejszym rozwiązaniem”.

Jak wspomniałem wcześniej, wydajność kodu nie powinna być najistotniejszym czynnikiem w pracy z Pythonem. Pierwszą rzeczą, jaką powinniśmy zrobić, optymalizując kod w Pythonie, jest... nie optymalizować tego kodu! Kod piszemy z myślą o czytelności i prostocie, następnie go optymalizujemy, ale tylko wówczas, gdy wystąpi taka konieczność. Może się bowiem okazać, że wydajność każdej z pięciu alternatyw jest w zupełności wystarczająca dla danych, którymi będzie zajmował się program. W takiej sytuacji wybieramy sposób najkorzystniejszy z punktu widzenia czytelności programu.

Alternatywne moduły mierzące wydajność

Moduł mierzący wydajność zastosowany w poprzednim punkcie działa, ale jest dość prymitywny i to pod kilkoma względami:

- do pomiaru czasu zawsze wykorzystuje funkcję `time.clock`; w systemie Windows to dobry wybór, ale w przypadku niektórych Uniksów dokładniejsze wyniki daje funkcja `time.time`,

- zmiana liczby powtórzeń wymaga modyfikacji zmiennej globalnej modułu, co okaże się niefortunnym rozwiązaniem, jeśli ten sam moduł jest równolegle importowany przez wiele różnych modułów,
- pomiar czasu działają przez wywoływanie tej samej funkcji ogromną liczbę razy; biorąc pod uwagę możliwość wystąpienia chwilowych obciążzeń systemu, lepiej jest wybrać najszyszybszy czas wykonania zamiast czasu całkowitego.

Poniższy listing implementuje bardziej zaawansowaną wersję modułu mierzącego wydajność. W kodzie obsłużymy każdy z wymienionych wyżej słabych punktów poprzedniej implementacji: funkcję pomiaru czasu dobieramy na podstawie detekcji systemu operacyjnego, liczba powtórzeń jest przekazywana w argumencie ze słowem kluczowym `_reps` i wyświetla wynik najlepszego pomiaru.

Plik mytimer.py (dla 2.6 i 3.0)

```
"""
timer(mielonka, 1, 2, a=3, b=4, _reps=1000) wywołuje i mierzy czas mielonka(1, 2, a=3),
wywołując testy _reps razy, zwracany jest całkowity czas testu;

best(mielonka, 1, 2, a=3, b=4, _reps=50) wywołuje _reps powtórzeń i zwraca najlepszy czas wykonania,
co pozwala odfiltrować wariancję pomiaru czasu spowodowaną obciążeniem systemu
"""

import time, sys
if sys.platform[:3] == 'win':
    timefunc = time.clock
else:
    timefunc = time.time
# W Windows używamy time.clock
# Lepsza dokładność w niektórych Uniksach
# Lub: print args

def trace(*args):
    pass

def timer(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 1000)
    trace(func, pargs, kargs, _reps)
    repplist = range(_reps)
    start = timefunc()
    for i in repplist:
        ret = func(*pargs, **kargs)
    elapsed = timefunc() - start
    return (elapsed, ret)
# Przekazany parametr lub wartość domyślna reps
# Lista wartości z zakresu w 2.6

def best(func, *pargs, **kargs):
    _reps = kargs.pop('_reps', 50)
    best = 2 ** 32
    for i in range(_reps):
        (time, ret) = timer(func, *pargs, _reps=1, **kargs)
        if time < best: best = time
    return (best, ret)
```

Dokumentujący ciąg znaków modułu prezentuje sposób jego użycia. Wykorzystuje metodę `pop` słownika w celu usunięcia argumentu `_reps` z listy argumentów funkcji i pozostawienia standardowego zestawu parametrów. Funkcję `trace` można zmodyfikować w taki sposób, aby wyświetlała argumenty wywołania, w obecnej postaci nie robi ona zupełnie nic. Aby wykorzystać ten nowy moduł w Pythonie 3.0 lub 2.6, należy zmodyfikować skrypt mierzący wydajność (pominąty kod wykorzystuje funkcje testowe wykonujące dodawanie $x + 1$, jak w wersji z poprzedniego punktu).

```
# Plik timeseqs.py

import sys, mytimer
reps = 10000
repslist = range(reps)

def forLoop(): ...
def listComp(): ...
def mapCall(): ...
def genExpr(): ...
def genFunc(): ...

print(sys.version)
for tester in (mytimer.timer, mytimer.best):
    print('<%s>' % tester.__name__)
    for test in (forLoop, listComp, mapCall, genExpr, genFunc):
        elapsed, result = tester(test)
        print ('-' * 35)
        print ('%-9s: %.5f => [%s...%s]' %
              (test.__name__, elapsed, result[0], result[-1]))
```

Po uruchomieniu w Pythonie 3.0 wyniki pomiarów będą bardzo zbliżone do poprzednich — jak się okazuje, nie ma znaczących różnic między wyborem najlepszego pomiaru a całkowitym czasem wszystkich powtórzeń. Duża liczba powtórzeń testu jest dobrym sposobem odfiltrowania odchyлеń od normy spowodowanych chwilowymi skokami obciążenia systemu, jednak metoda wyboru najlepszego pomiaru może okazać się precyzyjniejsza w przypadku operacji trwających dłuższy czas. Na mojej maszynie otrzymałem następujące wyniki:

```
C:\misc> c:\python30\python timeseqs.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
<timer>
-----
forLoop : 2.35371 => [10...10009]
-----
listComp : 1.29640 => [10...10009]
-----
mapCall : 3.16556 => [10...10009]
-----
genExpr : 1.97440 => [10...10009]
-----
genFunc : 1.95072 => [10...10009]
<best>
-----
forLoop : 0.00193 => [10...10009]
-----
listComp : 0.00124 => [10...10009]
-----
mapCall : 0.00268 => [10...10009]
-----
genExpr : 0.00164 => [10...10009]
-----
genFunc : 0.00165 => [10...10009]
```

Czasы uzyskane metodą wyboru najlepszego pomiaru są oczywiście niewielkie, ale mogą być dużo dłuższe w przypadku pomiarów innych operacji, na przykład wykonywanych na bardzo dużych zbiorach danych. Jak widać, przynajmniej w kwestii porównania alternatywnych implementacji, najwydajniejsze okazują się listy składane. Funkcja `map` jest niewiele wydajniejsza, ale wyłącznie w przypadku użycia funkcji wbudowanych.

Użycie argumentów mogących być tylko słowami kluczowymi w 3.0

W celu uproszczenia kodu w module `mytimer` możemy wykorzystać nową cechę funkcji w Pythonie 3.0: *argumenty mogące być tylko słowami kluczowymi*. Jak się dowiedzieliśmy w rozdziale 19., argumenty takie są doskonałym rozwiązaniem w sytuacji, gdy potrzebujemy dodatkowego argumentu typu `_reps`. Argumenty te muszą być zadeklarowane w nagłówku funkcji po argumentach pozycyjnych `*`, a przed argumentami ze słowami kluczowymi `**`, zaś w wywoaniu funkcji muszą być przekazane przez klucz i muszą występować przed argumentem `**`, o ile jest użyty. Poniższy listing prezentuje kod będący modyfikacją poprzedniego i wykorzystującą argument mogący być tylko słowem kluczowym. Kod ten jest prostszy, ale wymaga zastosowania Pythona 3.0 lub nowszego, nie będzie działał w 2.6.

```
# Plik mytimer.py (wyłącznie 3.X)

"""
Wykorzystana składnia argumentów mogących być tylko słowami kluczowymi, specyficzna dla 3.0, zamiast argumentów **
i wykonywania metody pop na słownikach.
Nie ma konieczności wydobywania listy z wyniku funkcji range(): w 3.0 to jest generator, nie lista
"""

import time, sys
trace = lambda *args: None # or print
timefunc = time.clock if sys.platform == 'win32' else time.time

def timer(func, *pargs, _reps=1000, **kargs):
    trace(func, pargs, kargs, _reps)
    start = timefunc()
    for i in range(_reps):
        ret = func(*pargs, **kargs)
    elapsed = timefunc() - start
    return (elapsed, ret)

def best(func, *pargs, _reps=50, **kargs):
    best = 2 ** 32
    for i in range(_reps):
        (time, ret) = timer(func, *pargs, _reps=1, **kargs)
        if time < best: best = time
    return (best, ret)
```

Tej wersji używa się dokładnie tak samo jak poprzedniej i tworzy ona identyczne wyniki (nie licząc oczywiście pomijalnych różnic pomiaru czasu):

```
C:\misc> c:\python30\python timeseqs.py
...takie same wyniki...
```

Tym razem dla odmiany przetestujemy tę wersję modułu w trybie interaktywnym, zupełnie niezależnie od skryptu mierzącego wydajność konstrukcji zwracających sekwencje, którym zajmowaliśmy się wcześniej. Jak wspomniałem, moduł pomiaru wydajności jest narzędziem ogólnego zastosowania.

```
C:\misc> c:\python30\python
>>> from mytimer import timer, best
>>>
>>> def power(X, Y): return X ** Y          # Funkcja testowana
...
>>> timer(power, 2, 32)                      # Czas całkowity
(0.002625403507987747, 4294967296)
>>> timer(power, 2, 32, _reps=1000000)       # Wymuszenie liczby powtórzeń
(1.1822605247314932, 4294967296)
>>> timer(power, 2, 100000)[0]                # 2 ** 100000, całkowity czas dla 1000 powtórzeń
2.2496919999608878
```

```
>>> best(power, 2, 32)          # Najlepszy czas, ostatni wynik
(5.58730229727189e-06, 4294967296)
>>> best(power, 2, 100000)[0]    # 2 ** 100,000 best time
0.0019937589833460834
>>> best(power, 2, 100000, _reps=500)[0]  # Wymuszamy liczbę powtórzeń
0.0019845399345541637
```

W przypadku tak trywialnego kodu jak zastosowany w tej sesji interaktywnej koszt czasowy wykonania kodu pomiaru czasu może być tak samo istotny jak koszt czasowy mierzonej funkcji, nie należy zatem uznawać zmierzonych czasów za wartości bezwzględne (mierzmy znacznie więcej niż czas wykonania wyrażenia $x^{**} Y$). Wyniki pomiaru wydajności pozwalają ocenić względne prędkości alternatywnych implementacji, a w przypadku operacji zajmującej sporo czasu pomiary te można już uznać za dość precyzyjne. Za przykład weźmy następny listing — obliczenie 2 do potęgi milionowej zajmuje o rząd wielkości więcej czasu niż 2 do potęgi sto tysięcy.

```
>>> timer(power, 2, 1000000, _reps=1)[0]      # 2 ** 1000000: czas całkowity
0.088112804839710179
>>> timer(power, 2, 1000000, _reps=10)[0]
0.40922470593329763

>>> best(power, 2, 1000000, _reps=1)[0]        # 2 ** 1000000: najlepszy czas
0.086550036387279761
>>> best(power, 2, 1000000, _reps=10)[0]       # 10 powtórzeń czasem daje tak samo dobre wyniki jak 50
0.029616752967200455
>>> best(power, 2, 1000000, _reps=50)[0]        # Najlepszy czas dla 50 powtórzeń
0.029486918030102061
```

W tym przypadku ponownie zmierzone czasy są bardzo krótkie, ale różnice mogą być znaczące w programach często obliczających potęgi.

Więcej informacji na temat argumentów mogących być tylko słowami kluczowymi, dodanych do Pythona w wersji 3.0 można znaleźć w rozdziale 19. Ta nowa cecha języka może znacząco uprościć tworzenie konfigurowalnych narzędzi, jak prezentowane powyżej funkcje. Niestety, nie jest ona zgodna z Pythonem 2.X. Jeśli chcemy porównywać wydajność kodu w Pythonie 2.X oraz 3.X albo gdy nasz kod ma być używany w starszych wersjach Pythona, poprzednia wersja jest lepszym wyborem. W przypadku użycia Pythona 2.6 można przeprowadzić testową sesję interaktywną prezentowaną wyżej, należy jedynie użyć poprzedniej wersji modułu mierzącego wydajność.

Inne sugestie

Dalsze eksperymenty w dziedzinie pomiaru czasu wykonania kodu mogą polegać na próbach modyfikowania liczby powtórzeń wykorzystywanych przez moduły. Warto też przetestować moduł standardowy Pythona `timeit`. Moduł ten ma za zadanie uproszczenie pisania kodu mierzącego czas wykonania, obsługuje tryby wywołania z wiersza poleceń i zawiera mechanizmy ułatwiające przenoszenie kodu między platformami systemowymi. Dokumentację jego użycia można znaleźć w standardowej dokumentacji Pythona.

Kolejnym narzędziem wartym sprawdzenia jest standardowy moduł `profile`, definiujący mechanizmy profilowania kodu. Więcej na jego temat dowiemy się w rozdziale 35. w kontekście tworzenia narzędzi programistycznych na potrzeby wielkich projektów. Kod powinien być profilowany przed etapem tworzenia alternatyw i wykonywania testów porównawczych, które prezentowaliśmy w tym podrozdziale.

Zamiast wykorzystywać wyrażenie podstawiania % (które może zostać wycofane w przyszłych wersjach Pythona), warto poeksperymentować z nową metodą formatowania ciągów znaków wprowadzoną w Pythonie 2.6 i 3.0. W przypadku skryptu mierzącego wydajność należy zmodyfikować wywołania funkcji print w sposób następujący:

```
print('<%s>' % tester.__name__)           # Wyrażenie  
print('<{0}>'.format(tester.__name__))      # Metoda formatująca  
  
print ('%-9s: %.5f => [%s...%s]' %  
      (test.__name__, elapsed, result[0], result[-1]))  
  
print('{0:<9}: {1:.5f} => [{2}...{3}]'.format(  
      test.__name__, elapsed, result[0], result[-1]))
```

Różnice w czytelności kodu pozostawiam ocenie Czytelnika.

W ramach dalszych eksperymentów można zmodyfikować kod funkcji mierzącej wydajność w taki sposób, aby zmierzyć wydajność zbiorów i słowników składanych dostępnych w Pythonie 3.0. Wykorzystanie tych nowych konstrukcji jest mniej powszechnie w Pythonie, zatem ich przetestowanie pozostawię jako sugerowane ćwiczenie dla Czytelnika (i bez zgadywania, bardzo proszę).

Kod modułu testującego wydajność przyda się w dalszej części książki — wykorzystamy go do pomiaru wydajności alternatywnych operacji obliczania pierwiastka kwadratowego w ćwiczeniach na końcu rozdziału. Będziemy również eksperymentować z technikami pomiaru czasu budowania słowników składanych w porównaniu z pętlami for.⁴

Pułapki związane z funkcjami

Zakończyliśmy rozdział poświęcony funkcjom, nadszedł więc czas na uświadomienie sobie kilku powszechnych pułapek związanych z tym zagadnieniem. Funkcje mają kilka zdradliwych cech, których mało kto się spodziewa. Wszystkie one są ukryte, a część z nich jest pomału usuwana z języka w najnowszych wydaniach, jednak stanowią poważny problem dla nowych programistów Pythona.

Lokalne nazwy są wykrywane w sposób statyczny

Jak wiemy, Python domyślnie klasyfikuje nazwy przypisywane w ciele funkcji jako *lokalne*, to znaczy zmiany w nich wprowadzane istnieją tylko w czasie działania funkcji. Jednak niewiele osób wie o tym, że Python wykrywa zmienne lokalne w sposób statyczny, na etapie komplikacji definicji funkcji, nie w trakcie wykonania. To prowadzi do szeregu dziwacznych zachowań, chętnie wytykanych przez początkujących programistów Pythona na listach dyskusyjnych.

⁴ Ciekawym ćwiczeniem może być zastosowanie prostej funkcji zdefiniowanej przez użytkownika, na przykład `def f(I): return(I)`, we wszystkich pięciu technikach, których wydajność mierzmy. Wyniki będą podobne jak w przypadku zastosowania funkcji wbudowanej, na przykład `abs` — spośród pięciu technik najszybszą okazuje się wersja `map`, jeśli wszystkie wywołują funkcję (wbudowaną lub własną), ale jest też naj wolniejsza, jeśli funkcja nie jest wywoływana. Innymi słowy, `map` okazuje się techniką wolniejszą tylko dla tego, że wymaga wywołania funkcji, a takie wywołania są relatywnie kosztowne czasowo. Ponieważ w przypadku funkcji `map` nie ma możliwości uniknięcia takiego wywołania, automatycznie traci ona w porównaniu z konkurencją.

Nazwa, która nie została przypisana w funkcji, jest odczytywana z otaczającego ją modułu:

```
>>> X = 99
>>> def selector():
...     print(X)
... selector()
99
```

W powyższym przypadku nazwa X jest odczytywana z modułu. Zaobserwujmy jednak, co się stanie, jeśli zastosujemy przypisanie do zmiennej X po jej odczytaniu w funkcji:

```
>>> def selector():
...     print(X)          # Z jeszcze nie istnieje!
...     X = 88           # X jest zaklasyfikowana jako zmienna lokalna
...                   # Analogiczna sytuacja może wystąpić dla "import X", "def X"...
>>> selector()
...pominięta treść błędu...
UnboundLocalError: local variable 'X' referenced before assignment
```

Otrzymujemy błąd użycia zmiennej przed przypisaniem, ale przyczyna jest trudna do uchwycenia. Python, komplując kod funkcji, napotyka przypisanie do nazwy `X`, więc decyduje, że jest to zmienna lokalna funkcji. Jednak na etapie wykonania funkcja `print` jest wywoływana przed przypisaniem, przez co zmienna lokalna `X` nie jest jeszcze zdefiniowana, co powoduje błąd. Zgodnie z regułami stosowania nazw w Pythonie, błąd informuje nas o tym, że zmienna lokalna została użyta, zanim została przypisana jej wartość. Każde przypisanie wartości do zmiennej w funkcji powoduje, że zmienna ta jest traktowana jako lokalna. Importy, przypisywanie, zagnieżdżone klasy i zagnieżdżone definicje funkcji są podatne na ten efekt uboczny.

Problem wynika z tego, że nazwy, do których przypisywane są wartości w ciele funkcji, są traktowane jako lokalne zmienne tej funkcji w całym jej zakresie, nie tylko od miejsca przypisania. Przedstawiony przykład zawiera zatem niejasność: czy intencją programisty było wypisanie wartości zmiennej `X` zdefiniowanej globalnie w module, a następnie utworzenie zmiennej lokalnej `X`? A może rzeczywiście mamy do czynienia z błędem programisty? Python traktuje nazwę `X` jako lokalną w całym zakresie funkcji, a taka sytuacja jest traktowana jako błąd. Jeśli chcemy wypisać wartość zmiennej globalnej `X`, należy zadeklarować ją w funkcji jako globalną:

```
>>> def selector():
...     global X
...     print(X)
...     X = 88
...
>>> selector()
88
```

Należy jednak pamiętać, że to spowoduje, że przypisania do zmiennej X zmodyfikują wartość globalną, nie lokalną. W ramach funkcji nie możemy użyć globalnej zmiennej X , a następnie utworzyć lokalnej zmiennej o takiej samej nazwie. Aby móc użyć zmiennych w taki sposób, należy zastosować kwalifikację (odwołanie do nazwy w jej przestrzeni nazw), w tym przypadku importując w funkcji moduł, w którym jest ona zdefiniowana, i odwołując się do zmiennej globalnej X przez kwalifikację w ramach modułu:

```
>>> X = 99
>>> def selector():
...     import __main__
...     print(__main__.X)      # Zimportowanie własnego modułu
...     X = 88                # Kwalifikowane odwołanie w celu uzyskania globalnej wersji nazwy
...     print(X)              # Niekwalifikowane odwołanie użycie wersji lokalnej
...                         # Wypisanie lokalnej wersji
```

```
>>> selector()
99
88
```

Kwalifikacja (odwołanie typu `.X`) pobiera wartość z przestrzeni nazw. W przypadku sesji interaktywnej przestrzeń nazw nosi nazwę `_main__`, zatem `_main__.X` odwołuje się do globalnej nazwy `X`. Szczegóły wyjaśniające tę zasadę można znaleźć w rozdziale 17.

W najnowszych wersjach Python został nieco naprawiony w tym zakresie i wypisuje trochę czytelniejsze komunikaty o błędach (dawniej wywoływał ogólny błąd nazwy). Jednak sam mechanizm działa nadal w ten sam sposób (pułapka nadal istnieje).

Wartości domyślne i obiekty mutowalne

Domyślne wartości argumentów są wyliczane i zapisywane statycznie podczas wywołania instrukcji `def`, nie przy wywołaniu samej funkcji. Wewnętrznie Python zapisuje w ramach samej funkcji po jednym obiekcie dla każdego parametru słownikowego (ze zdefiniowaną wartością domyślną).

Z reguły właśnie o to nam chodzi: ponieważ wartości domyślne są wyliczane na etapie budowania funkcji, mogą one być wyliczone z użyciem zmiennych zakresu nazw, w którym znajduje się funkcja, a następnie zapisane w tej funkcji. Jednak ponieważ wartość domyślna wykorzystuje ten sam obiekt przy każdym wywołaniu funkcji, należy zachować ostrożność w przypadku użycia zmiennych mutowalnych w charakterze wartości domyślnych. Na przykład poniższy kod wykorzystuje pustą listę jako wartość domyślną, a następnie zmienia tę wartość przy każdym wywołaniu funkcji.

```
>>> def saver(x=[]):
...     x.append(1)
...     print(x)
...
>>> saver([2])
[2, 1]                                         # Wartość domyślna nie jest użyta
>>> saver()
[1]                                         # Wartość domyślna jest użyta
>>> saver()
[1, 1]                                         # Rośnie przy każdym wywołaniu!
>>> saver()
[1, 1, 1]
```

Niektórzy programiści wykorzystują tę cechę języka: skoro mutowalne argumenty domyślne zachowują swój stan między wywołaniami, można ich używać w podobny sposób jak *staticznych* lokalnych zmiennych funkcji w języku C. W pewnym sensie mechanizm ten działa jak zmienne globalne, ale ich nazwy są lokalne w zakresie funkcji, zatem nie kolidują z takimi samymi nazwami zdefiniowanymi gdzieś indziej.

Dla większości programistów ta cecha Pythona wydaje się błędem, szczególnie gdy napotkają ją po raz pierwszy. Istnieją lepsze sposoby zachowania stanu między wywołaniami, jak choćby użycie klas, co omówimy w części VII.

Co więcej, mutowalne wartości domyślne są trudne do zapamiętania (i zrozumienia). Ich wartość jest zależna od kontekstu budowania obiektu funkcji. W poprzednim przykładzie w każdym wywołaniu funkcji wykorzystywany jest jeden obiekt listy: ten, który został utworzony na

etapie wykonania instrukcji `def`. Oznacza to, że w celu obsłużenia domyślnej wartości funkcji nie będzie za każdym razem tworzona nowa lista, przez co wykonanie metody `append` powoduje zwiększenie tej wartości domyślnej.

Jeśli nie o takie zachowanie funkcji nam chodzi, wystarczy na początku funkcji wykonać kopię wartości domyślnej i pracować na tej kopi. Inny sposób polega na przeniesieniu tworzenia domyślnej wartości do samej funkcji. Jeśli wartość jest tworzona przy każdym uruchomieniu funkcji, za każdym razem dostaniemy zupełnie nowy obiekt:

```
>>> def saver(x=None):
...     if x is None:                      # Nie został przekazany argument?
...         x = []                          # Tworzymy nową lokalną listę
...     x.append(1)                        # Modyfikujemy lokalną listę
...     print(x)
...
>>> saver([2])
[2, 1]
>>> saver()                           # Teraz nie rośnie
[1]
>>> saver()
[1]
```

Warunek `if` w powyższym przykładzie można zastąpić wyrażeniem `x = x or []`, które wykorzystuje fakt, że operator `or` zwraca wartość jednego ze swoich operandów: jeśli `x` nie ma zdefiniowanej wartości (domyślna wartość `None`), operator `or` zwróci pustą listę (drugi operand).

Jednak to nie będzie dokładnie to samo. Jeśli w argumencie zostanie przekazana pusta lista, wyrażenie `or` spowoduje, że rozszerzona i zwrócona zostanie nowa lista, nie ta, która została przekazana w argumencie funkcji, jak to się dzieje w przypadku użycia warunku `if` (wyrażenie zostaje rozwinięte do `x = [] or []`, co powoduje, że zwrócona zostaje wartość po prawej stronie operatora `or`; szczegóły można znaleźć w punkcie „Testy prawdziwości”). Decyzja o zastosowaniu jednej z tych wersji powinna być uzależniona od rzeczywistych wymagań w stosunku do programu.

W Pythonie istnieje jeszcze inny sposób uzyskania efektu mutowalnych wartości domyślnych — w znacznie bardziej zrozumiałym sposobie, a mianowicie z użyciem *atrybutów funkcji*, co omawiamy w rozdziale 19.

```
>>> def saver():
...     saver.x.append(1)
...     print(saver.x)
...
>>> saver.x = []
>>> saver()
[1]
>>> saver()
[1, 1]
>>> saver()
[1, 1, 1]
```

Nazwa funkcji jest nazwą globalną dla tej funkcji, ale nie musimy jej deklarować jako takiej, ponieważ nie modyfikujemy jej w danej funkcji. Mechanizm atrybutów działa zupełnie inaczej od domyślnych wartości argumentów, jednak ten sposób przywiązania wartości do funkcji jest znacznie czytelniejszy dla programisty (i bez wątpienia mniej magiczny).

Funkcje niezwracające wyników

W Pythonie funkcje nie muszą zwracać wyników: instrukcja `return` (lub `yield`) jest opcjonalna. Jeśli funkcja nie zwraca wartości wprost za pomocą instrukcji `return`, działanie funkcji kończy się wraz z końcem kodu funkcji. Z technicznego punktu widzenia wszystkie funkcje zwracają wartość. Jeśli nie zostanie ona określona w instrukcji `return`, wówczas funkcja automatycznie zwraca wartość `None`.

```
>>> def proc(x):
...     print(x)                                     # Brak instrukcji return powoduje zwrócenie wartości None
...
>>> x = proc('testing 123...')
testing 123...
>>> print(x)
None
```

Funkcje niezwracające wartości, takie jak powyższa, są w niektórych językach nazywane „procedurami”. Procedury są wywoływanie jak instrukcje, a zwracana wartość `None` jest ignorowana, ponieważ ich celem nie jest zwracanie użytecznego wyniku.

Warto zdawać sobie sprawę z tej specyfiki języka, ponieważ Python nie informuje o tym, że próbujemy użyć wyniku funkcji, która nie zwraca żadnej wartości. Na przykład przypisanie wyniku metody `append` nie wywoła błędu, ale otrzymamy wartość `None`, a nie (jak można by się spodziewać) zmodyfikowaną wartość listy:

```
>>> list = [1, 2, 3]
>>> list = list.append(4)                      # append jest "procedurą"
>>> print(list)                                # append modyfikuje listy w miejscu
None
```

Jak wspomniałem w punkcie „Często spotykane problemy programistyczne” w rozdziale 15., funkcje niezwracające wyniku wykonują swoją pracę jako efekt uboczny i powinny być stosowane w charakterze instrukcji, nie wyrażenia.

Funkcje zagnieżdżone a zmienne modyfikowane w pętli

Tę pułapkę opisałem w rozdziale 17. przy okazji dyskusji o zakresach zmiennych kontrolnych pętli w kontekście funkcji zagnieżdżonych, ale w ramach przypomnienia: w przypadku zagnieżdżenia funkcji w pętli należy zachować ostrożność z wykorzystaniem w argumentach tych funkcji wartości zmiennych modyfikowanych w pętli — zmienne te przy każdym wywołaniu będą miały *ostatnią* wartość przypisaną w pętli. Aby użyć właściwych wartości zmiennych z pętli, należy posłużyć się wartościami domyślnymi funkcji (więcej szczegółów można znaleźć w rozdziale 17.).

Podsumowanie rozdziału

Niniejszy rozdział zawiera podsumowanie zdobytej dotychczas wiedzy na temat wbudowanych narzędzi obiektów składanych oraz iteratorów. Omówiliśmy listy składane w kontekście narzędzi funkcyjnych, a jako nowe narzędzia iteracyjne zaprezentowaliśmy funkcje i wyrażenia generatorów. Następnie zajęliśmy się tematyką pomiarów wydajności różnych implementacji narzędzi iteracyjnych, a na końcu rozdziału omówiliśmy typowe, czyhające na programistów pułapki związane z użyciem narzędzi funkcyjnych.

W tym miejscu kończy się część książki poświęcona narzędziom funkcyjnym. W następnej części zajmiemy się *modułami*, które są najwyższym poziomem struktury programów w Pythonie, struktura, w której funkcjonują tworzone przez nas funkcje. Następnie zajmiemy się klasami, czyli narzędziami będącymi połączeniem funkcji i danych. Jak zobaczymy, klasy definiowane przez użytkownika mogą implementować obiekty realizujące protokół iteratorów, podobnie jak poznane w niniejszym rozdziale generatory i inne konstrukcje iterowane. Wiedza nabыта w tym rozdziale przyda się później, w kontekście metod klas.

Jednak zanim przejdziemy do następnego zagadnienia, popracujmy chwilę nad quizem do rozdziału oraz ćwiczeniami dla tej części książki, aby sprawdzić w praktyce nabytą właśnie wiedzę na temat funkcji.

Sprawdź swoją wiedzę — quiz

1. Jaka jest różnica między nawiasami kwadratowymi a okrągłymi w wyrażeniu listy składanej?
2. Jaki jest związek między generatorami a iteratorami?
3. W jaki sposób rozpoznać, czy funkcja jest generatorem?
4. Co robi instrukcja `yield`?
5. Jaki jest związek między funkcją `map` a listami składanymi? Podaj podobieństwa i różnice.

Sprawdź swoją wiedzę — odpowiedzi

1. Wyrażenie listy składanej w nawiasach kwadratowych zwraca listę wygenerowaną w całości. To samo wyrażenie ujęte w nawiasy okrągłe jest w rzeczywistości wyrażeniem generatora — efekt działania jest zbliżony, ale wynik nie jest tworzony w całości. Wyrażenie generatora zwraca obiekt generatora, który zwraca kolejne wyniki na żądanie, na przykład w kontekście iteracyjnym.
2. Generatory są obiektami obsługującymi protokół iteracyjny: posiadają metodę `__next__` zwracającą kolejny element sekwencji lub wywołującą wyjątek sygnalizujący osiągnięcie końca sekwencji. W Pythonie istnieje kilka sposobów tworzenia generatorów: funkcja generatora (zwracająca wyniki za pomocą instrukcji `yield`), wyrażenie generatora (odpowiednik listy składanej ujęty w nawiasy okrągłe) oraz obiekt generatora utworzony z klasy definiującej metodę `__iter__` (omówioną w dalszej części książki).
3. Funkcja generatora wykorzystuje w swoim kodzie instrukcję `yield`. Funkcje generatorów są pod względem składniowym identyczne ze zwykłymi funkcjami, ale są kompliwane przez interpretera Pythona w taki sposób, aby przy wywołaniu zwracały obiekt iterowany.
4. Gdy instrukcja `yield` występuje w kodzie funkcji, interpreter Pythona kompiluje tę funkcję w specjalny sposób, aby przy wywołaniu zwracała generator. Obiekt generatora zwracany przez funkcję tego typu obsługuje protokół iteracyjny. Gdy zostaje wywołana instrukcja `yield`, funkcja zwraca wynik do wyrażenia wywołującego i zawiesza swoje działanie. Funkcja może zostać wznowiona od tego miejsca, w którym wywołała instrukcję `yield` —

służy do tego funkcja wbudowana `next` lub metoda `__next__` generatora. Funkcje generatorów mogą również zwrócić wartość za pomocą instrukcji `return`, co kończy działanie generatora.

5. Funkcja `map` ma działanie podobne do listy składanej: obie konstrukcje budują nową listę, zbierając wyniki operacji wykonanych kolejno na elementach na sekwencji lub innego obiektu iterowanego. Główna różnica polega na tym, że funkcja `map` wymaga wywołania wskazanej funkcji na każdym elemencie sekwencji, natomiast lista składana pozwala wywołać dowolne wyrażenie. Dzięki temu listy składane mają bardziej ogólne zastosowanie; można ich użyć do wywołania funkcji na każdym elemencie (jak ma to miejsce w przypadku `map`), ale do wywołania innych wyrażeń za pomocą `map` użytkownik musi napisać do tego funkcję. Listy składane obsługują ponadto składnię rozszerzoną, pozwalając na zastosowanie zagnieżdżonych pętli `for` i warunku `if` pozwalającego na odfiltrowanie elementów z wyniku (co jest odpowiednikiem funkcji wbudowanej `filter`).

Sprawdź swoją wiedzę — ćwiczenia do części czwartej

W poniższych ćwiczeniach oczekuję od Czytelnika stworzenia nieco bardziej zaawansowanych programów. Własne rozwiązania należy skonsultować z podrozdziałem „Część IV Funkcje” w dodatku B. Kod powinien być zapisany w plikach modułów, dzięki czemu w przypadku błędu w konsoli interaktywnej nie będzie konieczności przepisywania wszystkiego od początku.

1. *Podstawy.* W konsoli interaktywnej Pythona napisać funkcję przyjmującą pojedynczy argument, wypisującą na ekranie jego wartość, a następnie wywołać ją ręcznie, przekazując argumenty różnych typów: ciąg znaków, liczbę całkowitą, listę, słownik. Następnie wywołać tę funkcję, nie podając argumentu. Co się wówczas stanie? A w przypadku podania dwóch argumentów?
2. *Argumenty.* Napisać funkcję `adder` w pliku modułu pythonowego. Funkcja powinna przyjmować dwa argumenty i zwracać ich sumę (lub połączenie). Następnie napisać kod testujący na końcu modułu, wywołujący tę funkcję z argumentami różnych typów (dwa ciągi znaków, dwie listy, dwie liczby zmiennoprzecinkowe) i wywołać moduł z wiersza poleceń jako skrypt. Czy istnieje konieczność wywoływania funkcji `print` w celu wyświetlenia na ekranie wyników tych wywołań?
3. *Zmienna liczba argumentów.* Uogólnić funkcję `adder` napisaną w poprzednim punkcie w taki sposób, aby obliczała sumę dowolnej liczby argumentów i zmienić wywołania testowe w taki sposób, aby wykorzystywały więcej niż jeden argument. Jakiego typu jest zwracana suma? (Wskazówka: wycinek `S[:0]` zwraca pustą sekwencję tego samego typu co `S`, a funkcja wbudowana `type` może służyć do sprawdzania typów. Prostsze rozwiązanie można znaleźć w rozdziale 18. w przykładzie własnej implementacji funkcji `min`). Co się stanie, gdy zostaną przekazane argumenty różnych typów? Co, jeśli zostaną przekazane słowniki?
4. *Słowa kluczowe.* Zmodyfikować funkcję `adder` napisaną w punkcie 2. w taki sposób, aby akceptowała trzy argumenty i wykonywała na nich operację sumowania lub łączenia: `def adder(good, bad, ugly)`. Zdefiniować wartości domyślne każdego z tych argumentów i poeksperymentować z funkcją w trybie interaktywnym. Spróbować wywołań z jednym, dwoma, trzema i czterema argumentami. Następnie przekazać argumenty ze słowami kluczowymi. Czy zadziała wywołanie `adder(ugly=1, good=2)`? Dlaczego tak się dzieje? Uogólnić funkcję `adder` w taki sposób, aby akceptowała dowolną liczbę argumentów ze

- słowami kluczowymi. Zadanie jest podobne do punktu 3., z tą różnicą, że należy iterować po słowniku, nie po krotce argumentów. (Wskazówka: metoda `dict.keys` zwraca listę, po której można przemieszczać się w pętli `for` lub `while`, ale w Pythonie 3.0 wynik tej metody należy przekształcić na listę, zanim odwołamy się do jej elementów po indeksie!).
5. Napisać funkcję `copyDict(dict)` kopującą argumenty słownikowe. Funkcja powinna zwrócić kopię słownika wszystkich swoich argumentów. Użyć metody `keys` słownika do przemieszczania się po kluczach słownika (od Pythona 2.2 można iterować po kluczach słownika bez wywołania metody `keys`). Kopiowanie sekwencji jest proste (`X[:]` wykonuje tzw. płytką kopię). Czy taka metoda działa również dla słowników?
 6. Napisać funkcję `addDict(dict1, dict2)` generującą złączenie dwóch słowników. Funkcja powinna zwracać nowy słownik zawierający wszystkie wartości z obydwu argumentów (przyjmuje się, że są to słowniki). Jeśli w każdym z argumentów występuje ten sam klucz, w wyniku może znaleźć się dowolna z wartości. Przetestować funkcję za pomocą kodu testującego w zawierającym ją module, kod powinienny być uruchamiany w przypadku wywołania funkcji w formie skryptu. Co się stanie, jeśli zamiast słowników zostaną przekazane listy? W jaki sposób można uogólnić funkcję, aby obsługiwała również ten przypadek? (Wskazówka: przydatna może okazać się funkcja `type` wspomniana wcześniej. Czy kolejność przekazanych argumentów ma znaczenie?).
 7. Więcej przykładów dopasowywania argumentów. Zdefiniować sześć poniższych funkcji (w trybie interaktywnym lub pliku modułu):

```

def f1(a, b): print(a, b)                      # Normalne argumenty
def f2(a, *b): print(a, b)                      # Zmienna liczba argumentów pozycyjnych

def f3(a, **b): print(a, b)                     # Zmienna liczba słów kluczowych

def f4(a, *b, **c): print(a, b, c)              # Tryby mieszane

def f5(a, b=2, c=3): print(a, b, c)            # Wartości domyślne

def f6(a, b=2, *c): print(a, b, c)              # Zmienna liczba argumentów pozycyjnych i wartości domyślnych

```

Przetestować te funkcje w trybie interaktywnym i wyjaśnić wynik. W niektórych przypadkach przydatny może okazać się algorytm wydobywania argumentów funkcji opisany w rozdziale 18. Czy mieszanie trybów dopasowania argumentów jest dobrym pomysłem? Czy można sobie wyobrazić przypadki, gdy taka technika będzie użyteczna?

```

>>> f1(1, 2)
>>> f1(b=2, a=1)

>>> f2(1, 2, 3)
>>> f3(1, x=2, y=3)
>>> f4(1, 2, 3, x=2, y=3)
>>> f5(1)
>>> f5(1, 4)

>>> f6(1)
>>> f6(1, 3, 4)

```

8. Powrót do liczb pierwszych. Poniższy kod jest powtórką z rozdziału 13. Jest to uproszczony algorytm weryfikujący, czy podana liczba jest liczbą pierwszą:

```

x = y // 2                                     # Dla y > 1
while x > 1:
    if y % x == 0:                            # Reszta z dzielenia

```

```

        print(y, 'dzieli się przez', x)
        break                                # Pomijamy resztę
    x -= 1
else:                                     # Normalne wyjście
    print(y, 'jest liczbą pierwszą')

```

Zapisać ten kod jako funkcję w module do ponownego użycia (y powinien być przekazywanym argumentem), a w module umieścić kod testujący w przypadku uruchomienia modułu jako skryptu. Poeksperymentować, zastępując operator dzielenia całkowitego `//` z pierwszego wiersza kodu, aby przekonać się, w jaki sposób w 3.0 zmieniła się semantyka zwykłego dzielenia `(/)`, co może powodować efekty uboczne w kodzie napisanym dla poprzednich wersji Pythona (dalsze informacje na ten temat można znaleźć w rozdziale 5.). Co można zrobić z liczbami ujemnymi oraz z 0 i 1? W jaki sposób przyspieszyć ten kod? Wyniki wywołania funkcji powinny wyglądać następująco:

```

13 jest liczbą pierwszą
13.0 jest liczbą pierwszą
15 dzieli się przez 5
15.0 dzieli się przez 5.0

```

9. *Listy składane.* Napisać kod budujący nową listę zawierającą pierwiastki kwadratowe wszystkich liczb z listy [2, 4, 9, 16, 25]. Kod zapisać w postaci pętli `for`, funkcji `map`, a następnie listy składanej. Wykorzystać funkcję `sqrt` ze standardowego modułu `math` (zaimportować moduł `math`, a następnie `math.sqrt(X)`). Która z tych wersji wygląda najlepiej?
10. *Narzędzia do pomiaru czasu.* W rozdziale 5. widzieliśmy trzy sposoby obliczenia pierwiastków kwadratowych: `math.sqrt(X)`, `X ** .5` oraz `pow(X, .5)`. Jeśli w pisany programie jesteśmy zmuszeni do wykonywania dużej ilości tego typu obliczeń, różnice w wydajności tych alternatyw mogą mieć znaczący wpływ na wydajność całego programu. Należy zmodyfikować moduł `timeseqs.py` prezentowany w niniejszym rozdziale (można użyć wersji z argumentem mogącym być tylko słowem kluczowym specyficznej dla 3.0 lub uniwersalnej implementacji dla 2.6/3.0) w taki sposób, aby zmierzyć wydajność każdej z tych trzech alternatyw obliczania pierwiastka kwadratowego. Można również rozbudować moduł testujący w taki sposób, aby do funkcji uruchamiającej testy przekazywać krotkę testowanych funkcji (można przyjąć metodę `kopiuj-wklej-zmodyfikuj`). Która z podanych alternatyw obliczania pierwiastków kwadratowych jest najwydajniejsza? Dodatkowo w trybie interaktywnym porównać wydajność słowników składanych z pętlami `for`.

CZĘŚĆ V

Moduły

Moduły — wprowadzenie

Niniejszy rozdział rozpoczyna nasze pogłębione omówienie *modułu* w Pythonie — jednostki najwyższego poziomu organizacji programu, która pakuje razem kod programu oraz dane w celu późniejszego, ponownego ich użycia. Moduły zazwyczaj odpowiadają plikom programów Pythona (lub rozszerzeniom napisanym w językach zewnętrznych, takich jak C, Java czy C#). Każdy plik jest modelem, a moduły importują inne moduły w celu skorzystania z zawartych w nich zmiennych. Moduły przetwarzane są za pomocą dwóch instrukcji oraz jednej ważnej funkcji:

```
import
    Pozwala klientowi (plikowi importującemu) pobrać moduł jako całość.

from
    Pozwala klientom pobierać określone zmienne modułu.

imp.reload
    Umożliwia przeładowanie kodu modułu bez zatrzymywania Pythona.
```

W rozdziale 3. przedstawiono podstawy modułów i od tego czasu regularnie z modułami korzystamy. Niniejsza część książki rozpoczyna się od rozszerzenia podstawowych koncepcji dotyczących modułów, a następnie przechodzi do bardziej zaawansowanego ich użycia. Pierwszy rozdział umożliwia ogólne spojrzenie na rolę modułów w całej strukturze programu. W kolejnym rozdziale oraz kilku następnych zagłębimy się w szczegóły programistyczne stojące za teorią.

Po drodze zapoznamy się z pominiętymi dotychczas szczegółami dotyczącymi modułów. Dowiemy się więcej na temat między innymi przeładowywania modułów, atrybutów `_name_` oraz `_all_`, importowania pakietów czy składeń importowania względnego. Ponieważ moduły oraz klasy są jedynie wyróżnionymi przestrzeniami nazw, powrócimy również do koncepcji związanych z samymi przestrzeniami nazw.

Po co używa się modułów?

W skrócie, moduły udostępniają łatwy sposób organizowania komponentów w systemy, służąc jako samodzielne pakiety zmiennych znane jako *przestrzenie nazw*. Wszystkie zmienne zdefiniowane na najwyższym poziomie pliku modułu stają się atrybutami zaimportowanego obiektu modułu. Jak widzieliśmy w poprzedniej części książki, importowanie daje dostęp do

nazw z zakresu globalnego modułu. Oznacza to, że zakres globalny pliku modułu staje się po zaimportowaniu przestrzenią nazw atrybutów obiektu modułu. Moduły Pythona pozwalają na łączenie pojedynczych plików w większe systemy programów.

Co więcej, z ogólnego punktu widzenia moduły pełnią przynajmniej trzy role.

Ponowne wykorzystanie kodu

Jak wspomniano w rozdziale 3., moduły pozwalają na stałe zachować kod w plikach. W przeciwnieństwie do kodu wpisanego w interaktywnym wierszu poleceń Pythona, który znika w momencie zakończenia Pythona, kod z modułów pozostaje — można go przeładować i wykonać ponownie tyle razy, ile potrzebujemy. Co ważniejsze, moduły są miejscem definiowania nazw, znanych jako *atrybuty*, do których może się odnosić wiele klientów zewnętrznych.

Dzielenie przestrzeni nazw systemu

Moduły są również w Pythonie jednostką organizacyjną najwyższego poziomu. Przede wszystkim są one pakietami zmiennych. Moduły łączą zmienne w samodzielne pakiety, co pomaga zapobiegać konfliktom — dopóki w jawnym sposób nie zaimportujemy pliku, nie zobaczymy jego zmiennych. Tak naprawdę wszystko, co „żyje” w module — wykonywany kod i tworzone obiekty — jest zawsze w niewidzialny sposób w nim zamknięte. Z tego powodu moduły są naturalnymi narzędziami grupowania komponentów systemu.

Implementowanie współdzielonych usług oraz danych

Z operacyjnego punktu widzenia moduły przydają się również do implementowania komponentów, które są współdzielone w całym systemie i tym samym wymagają tylko jednej kopii. Jeśli na przykład chcemy udostępnić obiekt globalny, który będzie wykorzystywany przez większą liczbę funkcji lub plików, możemy umieścić go w module, który następnie zostanie zaimportowany przez wielu klientów.

By w pełni zrozumieć rolę modułów w systemie Pythona, musimy poczynić pewną dygresję i zająć się na chwilę ogólną strukturą programu napisanego w Pythonie.

Architektura programu w Pythonie

Dotychczas w opisach programów Pythona w książce upraszczałem ich stopień skomplikowania. W praktyce programy zazwyczaj obejmują więcej niż jeden plik. Większość programów, poza może najprostszymi skryptami, przybiera formę systemów składających się z wielu plików. I nawet jeśli uda nam się zmieścić własny program w jednym pliku, z pewnością kiedyś zaczniemy używać plików zewnętrznych napisanych przez inne osoby.

W niniejszym podrozdziale omówimy ogólną architekturę programów Pythona — sposób dzielenia programu na zbiór plików źródłowych (inaczej modułów) oraz łączenie poszczególnych części w całość. Po drodze zapoznamy się również z najważniejszymi zagadnieniami dotyczącymi modułów Pythona, importowania oraz atrybutów obiektów.

Struktura programu

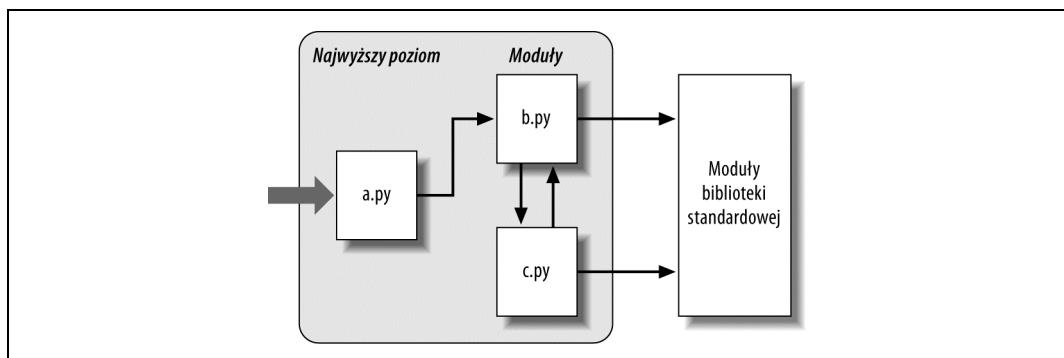
Program napisany w Pythonie składa się najczęściej z kilku plików tekstowych zawierających instrukcje Pythona. Program ustrukturyzowany jest jako jeden główny plik *najwyższego poziomu* wraz z zero lub większą liczbą plików dodatkowych znanych w Pythonie jako *moduły*.

W Pythonie plik najwyższego poziomu (nazywany także skryptem) zawiera podstawowy przebieg sterowania programu — jest to plik, który wykonuje się w celu uruchomienia aplikacji. Pliki modułów są bibliotekami narzędzi wykorzystywanymi do zebrania komponentów używanych przez plik najwyższego poziomu (i być może również gdzie indziej). Pliki najwyższego poziomu wykorzystują narzędzia zdefiniowane w plikach modułów, a moduły korzystają z narzędzi zdefiniowanych w innych modułach.

Pliki modułów na ogół nie robią nic po bezpośrednim ich wykonaniu. Zamiast tego definiują narzędzia, które mogą być wykorzystane w innych plikach. W Pythonie plik *importuje* moduł w celu uzyskania dostępu do narzędzi w nim zdefiniowanych, znanych jako *atrybuty* (czyli nazwy zmiennych dołączone do obiektów, takie jak funkcje). Importujemy moduły i uzyskujemy dostęp do ich atrybutów w celu wykorzystania ich narzędzi.

Importowanie i atrybuty

Spróbujmy to trochę skonkretyzować. Na rysunku 21.1 przedstawiono strukturę napisanego w Pythonie programu składającego się z trzech plików — *a.py*, *b.py* oraz *c.py*. Plik *a.py* został wybrany jako plik najwyższego poziomu. Będzie prostym plikiem tekstowym z instrukcjami, po uruchomieniu wykonywanym od góry do dołu. Pliki *b.py* oraz *c.py* są modułami. Są również plikami tekstowymi z instrukcjami, jednak zazwyczaj nie są uruchamiane w sposób bezpośredni. Zamiast tego, zgodnie z powyższym opisem, moduły są zazwyczaj importowane przez inne pliki, które chcą korzystać ze zdefiniowanych przez nie narzędzi.



Rysunek 21.1. Architektura programu w Pythonie. Program jest systemem modułów. Zawiera jeden plik skryptu najwyższego poziomu (uruchamiany w celu wykonania programu) i większą liczbę plików modułów (importowanych bibliotek narzędzi). Skrypty oraz moduły są plikami tekstowymi zawierającymi instrukcje Pythona, choć instrukcje z modułów zazwyczaj po prostu tworzą obiekty, które mają być użyte później. Biblioteka standardowa Pythona udostępnia zbiór gotowych modułów

Załóżmy na przykład, że plik *b.py* z rysunku 21.1 definiuje funkcję o nazwie *spam*, która może być wykorzystana przez pliki zewnętrzne. Jak wiemy z omówienia funkcji w czwartej części książki, *b.py* zawiera generującą funkcję instrukcję Pythona `def`, którą można później wykonać, przekazując do niej zero lub większą liczbę wartości w nawiasach po nazwie funkcji.

```
def spam(text):
    print(text, 'mielonka')
```

Załóżmy teraz, że skrypt *a.py* chciałby skorzystać z funkcji *spam*. Mógłby zatem zawierać instrukcję podobne do poniższych.

```
import b  
b.spam('gumby')
```

Pierwsza z tych instrukcji (`import`) daje plikowi `a.py` dostęp do wszystkich zmiennych zdefiniowanych przez kod najwyższego poziomu w pliku `b.py`. Oznacza w przybliżeniu: „Załaduj plik `b.py` (o ile nie został on jeszcze załadowany) i daj mi dostęp do wszystkich jego atrybutów przez nazwę `b`”. Instrukcje `import` (a także — jak zobaczymy później — instrukcje `from`) wykonyują i ładują inne pliki w czasie wykonywania.

W Pythonie połączenie między plikami modułów nie jest gotowe, dopóki instrukcja `import` nie zostanie wykonana. W rezultacie nazwy modułów (jako zwykłe zmienne) zostają załadowane do obiektów modułów. Tak naprawdę nazwa modułu użyta w instrukcji `import` spełnia dwie role: identyfikuje ona zewnętrzny plik, jaki ma zostać załadowany, ale również staje się zmienną przypisaną do załadowanego modułu. Obiekty zdefiniowane w module są również tworzone w czasie wykonywania, kiedy wykonywana jest instrukcja `import`. Instrukcja ta tak naprawdę wykonuje instrukcje z pliku docelowego jedna za drugą — w celu utworzenia jego zawartości.

Druga instrukcja z pliku `a.py` wywołuje funkcję `spam` zdefiniowaną w module `b`, używając do tego zapisu z atrybutem obiektu. Kod `b.spam` oznacza, że należy pobrać wartość zmiennej `spam` znajdującej się w obiekcie `b`. Zmienna ta okazuje się funkcją wywoływalną, do której w nawiasach przekazujemy łańcuch znaków (`'gumby'`). Gdybyśmy naprawdę utworzyli te pliki, zapisali je i wykonali `a.py`, wyświetcone zostałyby słowa „`gumby mielonka`”.

Zapis `obiekt.atrybut` można spotkać we wszystkich skryptach napisanych w Pythonie — większość obiektów ma przydatne atrybuty pobierane za pomocą operatora `..`. Niektóre z nich można wywoływać (jak funkcje), inne to proste wartości danych nadające właściwości obiektom (na przykład imię osoby).

Pojęcie importowania jest również w Pythonie całkowicie uniwersalne. Dowolny plik może importować narzędzia z każdego innego pliku. Plik `a.py` może na przykład importować `b.py` w celu wywołania jego funkcji, jednak `b.py` może również zaimportować `c.py`, by skorzystać z różnych narzędzi zdefiniowanych w tym module. Łańcuchy importowania mogą sięgać tak głęboko, jak chcemy — w przykładzie wyżej moduł `a` może importować moduł `b`, który importuje moduł `c`, importujący z kolei moduł `b`.

Poza służeniem jako struktura organizacyjna najwyższego poziomu moduły (i pakiety modułów opisane w rozdziale 23.) są również najwyższym poziomem *ponownego wykorzystania kodu* w Pythonie. Umieszczanie komponentów w plikach modułów sprawia, że są one przydatne w naszym aktualnym programie, ale również w każdym innym programie, jaki możemy kiedyś napisać. Po utworzeniu programu z rysunku 21.1 możemy na przykład odkryć, że funkcja `b.spam` jest uniwersalnym narzędziem, które można wykorzystać w zupełnie innym programie. Wystarczy tylko ponownie zaimportować plik `b.py` z pliku innego programu.

Moduły biblioteki standardowej

Warto zwrócić uwagę na element rysunku 21.1 znajdujący się najbardziej na prawo. Niektóre moduły importowane przez nasz program pochodzą z samego Pythona, a nie z tworzonych przez nas plików.

Python automatycznie zawiera wielki zbiór modułów narzędzi znany pod nazwą *biblioteki standardowej*. Zbiór ten, składający się z około dwustu modułów, zawiera niezależną od platformy obsługę często wykonywanych zadań programistycznych — interfejsów systemu operacyjnego, trwałości obiektów, dopasowywania wzorców tekstowych, skryptów sieciowych i internetowych, tworzenia graficznego interfejsu użytkownika i wiele, wiele innych. Żadne z tych narzędzi nie jest częścią samego języka Python, jednak można z nich korzystać, importując odpowiednie moduły w dowolnej standardowej instalacji Pythona. Ponieważ są to moduły biblioteki standardowej, można być stosunkowo pewnym, że będą one dostępne i działające na większości platform, na których uruchamia się Pythona.

W przykładach z książki można zobaczyć działanie niektórych modułów z biblioteki standardowej w praktyce. By uzyskać pełny ich obraz, należy zajrzeć do dokumentacji biblioteki standardowej Pythona dostępnej albo wraz z naszą instalacją Pythona (za pomocą IDLE lub w odpowiednim wpisie opcji *Python* w menu *Start* systemu Windows), albo w Internecie pod adresem <http://www.python.org>.

Ponieważ istnieje tak wiele modułów, jest to tak naprawdę jedyny sposób przekonania się, jakie narzędzia są dostępne. Omówienie narzędzi biblioteki Pythona można również znaleźć w publikowanych książkach dotyczących programowania aplikacji, takich jak *Programming Python* wydawnictwa O'Reilly, jednak dokumentacja jest darmowa, można ją oglądać w dowolnej przeglądarce internetowej (publikowana jest w formacie HTML) i jest uaktualniana z każdym kolejnym wydaniem Pythona.

Jak działa importowanie

W poprzednim podrozdziale omawialiśmy importowanie modułów bez objaśnienia, co tak naprawdę się dzieje, kiedy wykonujemy tę operację. Ponieważ importowanie jest w Pythonie sercem struktury programu, w niniejszym podrozdziale przyjrzymy się detalom operacji importowania, by proces ten stał się dla nas mniej abstrakcyjny.

Niektórzy programiści języka C lubią porównywać importowanie modułów w Pythonie do `#include` z języka C, jednak tak naprawdę nie powinni tego robić — w Pythonie importowanie nie jest tylko tekstowym wstawieniem jednego pliku do drugiego. Tak naprawdę są to operacje w czasie wykonywania, które przy pierwszym imporcie danego pliku przez program składają się z trzech osobnych kroków.

1. *Odnalezienie* pliku modułu.
2. *Skompilowanie* go do kodu bajtowego (jeśli jest to konieczne).
3. *Wykonanie* kodu modułu w celu utworzenia definiowanych przez niego obiektów.

By lepiej zrozumieć importowanie modułów, omówimy poszczególne kroki po kolei. Należy pamiętać, że wszystkie trzy kroki są wykonywane jedynie przy *pierwszym* importowaniu modułu w czasie wykonywania programu. Późniejsze importy tego samego modułu pomijają wszystkie te kroki i po prostu pobierają załadowany obiekt modułu z pamięci. Python wykonyuje to za pomocą przechowywania modułów w tabeli o nazwie `sys.modules` i sprawdzania tej tabeli na początku operacji importowania. Jeśli moduł jest tam nieobecny, rozpoczyna się poniższy proces składający się z trzech kroków.

1. Odnalezienie modułu

Przede wszystkim Python musi zlokalizować plik modułu, do którego odnosi się instrukcja `import`. Warto zauważyć, że instrukcja `import` z przykładu z poprzedniego podrozdziału podaje nazwę pliku bez przyrostka `.py` i bez ścieżki do katalogu. W kodzie podano jedynie `import b`, a nie coś w stylu `import c:\katalog1\b.py`. Tak naprawdę podaje się jedynie prostą nazwę; ścieżka do katalogu oraz przyrostek pomijane są celowo, gdyż Python do odszukania pliku modułu wymienionego w instrukcji `import` wykorzystuje standardową ścieżkę *wyszukiwania modułów*.¹ Ponieważ to najważniejsza część operacji importowania, jaką muszą znać programiści, powrócimy do tego zagadnienia za moment.

2. (Ewentualne) Kompilowanie

Po odnalezieniu (dzięki przejściu ścieżki wyszukiwania modułów) pliku z kodem źródłowym, odpowiadającego instrukcji `import`, Python kompluluje następnie ten plik do kodu bajtowego — jeśli jest to konieczne. Kod bajtowy omawialiśmy w rozdziale 2.

Python sprawdza daty plików i jeśli plik z kodem bajtowym jest starszy od pliku źródłowego (w którym coś zmieniliśmy), automatycznie ponownie generuje kod bajtowy przy wykonywaniu programu. Jeśli jednak plik z kodem bajtowym `.pyc` okaże się nie starszy od odpowiadającego mu pliku z kodem źródłowym, pomija krok komplikacji kodu źródłowego na kod bajtowy. Dodatkowo jeśli Python odnajduje w ścieżce wyszukiwania jedynie plik z kodem bajtowym, a nie widzi źródeł, po prostu ładuje kod bajtowy bezpośrednio (co oznacza, że możemy publikować program w postaci samych plików z kodem bajtowym i nie udostępniać plików źródłowych). Innymi słowy, krok komplikacji jest pomijany, jeśli jest to możliwe, w celu przyspieszenia uruchamiania programu.

Warto zauważyć, że komplikacja ma miejsce, kiedy plik jest importowany. Z tego powodu zazwyczaj nie widzimy kodu pliku bajtowego `.pyc` dla pliku najwyższego poziomu programu, o ile nie zostanie on zimportowany przez inny plik — jedynie pliki importowane pozostawiają na naszym komputerze pliki `.pyc`. Kod bajtowy plików najwyższego poziomu jest zazwyczaj wykorzystywany wewnętrznie i usuwany. Kod bajtowy importowanych plików jest zapisywany w plikach, by móc w przyszłości poprawić szybkość wykonywania.

Pliki najwyższego poziomu są często zaprojektowane do bezpośredniego wykonywania, a nie importowania. Później zobaczymy, że można projektować pliki służące zarówno jako kod najwyższego poziomu programu, jak i moduł narzędzi do zimportowania. Takie pliki mogą być wykonywane oraz importowane i tym samym generują pliki `.pyc`. By zobaczyć, jak to działa, wystarczy prześledzić omówienie specjalnego atrybutu `_name_` oraz `_main_` w rozdziale 24.

¹ Tak naprawdę podanie ścieżki do katalogu oraz przyrostka pliku w instrukcji `import` jest niepoprawne składowo. *Importowanie pakietów*, omówione w rozdziale 23., pozwala na umieszczenie w instrukcji `import` części ścieżki do katalogu prowadzącej do pliku w postaci zbioru nazw rozdzielonych kropkami. Importowanie pakietów nadal lokalizuje najbardziej lewy katalog ze ścieżki pakietu za pomocą normalnej ścieżki wyszukiwania modułów (jest zatem względne w stosunku do katalogu ze ścieżki wyszukiwania). W instrukcjach `import` nie można również używać żadnej składni specyficznej dla jakiejś platformy — taka składnia działa jedynie w ścieżce wyszukiwania. Należy również pamiętać, że problemy ze ścieżką wyszukiwania modułów nie są tak istotne, kiedy używa się *zamrożonych plików binarnych* (omówionych w rozdziale 2). Zazwyczaj osadzają one kod bajtowy w obrazie binarnym.

3. Wykonanie

Ostatnim krokiem w operacji importowania jest wykonanie kodu bajtowego modułu. Wszystkie instrukcje pliku są wykonywane po kolej, od góry do dołu, a przypisania do nazw wykonane na tym etapie generują atrybuty wynikowego obiektu modułu. Krok wykonywania generuje zatem wszystkie narzędzia definiowane przez jądro modułu. Przykładowo instrukcje `def` pliku są wykonywane w czasie importowania w celu utworzenia funkcji i przypisania atrybutów wewnętrz modułu do tych funkcji. Funkcje mogą następnie być wywoływane później w programie przez kod importujący plik.

Ponieważ ostatni krok operacji importowania faktycznie wykonuje kod pliku, jeśli jakiś kod najwyższego poziomu w pliku modułu naprawdę coś robi, zobaczymy jego wynik w czasie importowania. Przykładowo instrukcje `print` na najwyższym poziomie modułu wyświetla swoje wyniki po zimportowaniu pliku. Instrukcje `def` funkcji z kolei po prostu definiują obiekty, które mogą zostać wykorzystane później.

Jak widać, operacje importowania obejmują całkiem sporo pracy — wyszukują pliki, być może uruchamiają kompilator i wykonują kod Pythona. Z tego powodu każdy moduł importowany jest domyślnie tylko raz na proces. Przyszłe operacje importowania pomijają wszystkie trzy ważne etapy i korzystają z modułu załadowanego już do pamięci.² Jeśli potrzebujemy ponownie zainportować plik po tym, jak został już załadowany (na przykład w celu obsługi dostosowania kodu do potrzeb użytkownika), musimy to wymusić za pomocą wywołania funkcji `imp.reload` — narzędzia, z którym spotkamy się w kolejnym rozdziale.

Ścieżka wyszukiwania modułów

Jak wspomniano wcześniej, najistotniejszą dla programistów częścią procedury importowania jest zazwyczaj część pierwsza — lokalizacja pliku, który ma zostać zainportowany (część „odnalezienie”). Ponieważ możemy być zmuszeni do poinformowania Pythona o tym, gdzie ma szukać importowanych plików, musimy wiedzieć, w jaki sposób możemy dostać się do jego ścieżki wyszukiwania w celu rozszerzenia jej.

W wielu przypadkach możemy polegać na automatycznej naturze ścieżki wyszukiwania importowanych modułów i nie musimy jej wcale konfigurować. Jeśli jednak chcemy importować pliki ponad granicami katalogów, musimy wiedzieć, w jaki sposób działa ścieżka wyszukiwania, by móc ją dostosować do własnych potrzeb. Ścieżka wyszukiwania modułów Pythona składa się z zestawienia poniższych komponentów podstawowych. Niektóre z nich są ustawione za nas, inne musimy dostosować do własnych potrzeb, by przekazać Pythonowi, gdzie ma szukać.

1. Katalog główny programu.
2. Katalogi PYTHONPATH (jeśli są ustawione).
3. Katalogi biblioteki standardowej.
4. Zawartość wszystkich plików `.pth` (jeśli są one obecne).

² Jak wspomniano wcześniej, Python przechowuje zainportowane już moduły we wbudowanym słowniku `sys.modules`, dzięki czemu może śledzić, które z nich zostały już załadowane. Jeśli chcemy sprawdzić, które moduły zostały już załadowane, należy zainportować moduł `sys` i wyświetlić `list(sys.modules.keys())`. Więcej informacji na temat tej wewnętrznej tabeli znajduje się rozdziale 24.

Zestawienie tych czterech komponentów staje się `sys.path` — listą łańcuchów znaków nazw katalogów, o której powiemy więcej nieco później w tym podrozdziale. Pierwszy i trzeci element ścieżki wyszukiwania definiowane są automatycznie. Ponieważ jednak Python przeszukuje zestawienie tych komponentów od góry do dołu, drugi i czwarty element można wykorzystać do rozszerzenia tej ścieżki w taki sposób, by obejmowała ona nasze własne katalogi z kodem źródłowym. Poniżej przedstawiono, w jaki sposób Python wykorzystuje każdy z tych komponentów ścieżki.

Katalog główny

Python najpierw szuka importowanych plików w katalogu głównym. Znaczenie „katalogu głównego” będzie różne w zależności od sposobu uruchamiania kodu. Jeśli wykonujemy program, będzie to katalog zawierający plik skryptowy najwyższego poziomu tego programu. Jeśli pracujemy interaktywnie, będzie to katalog, w którym pracujemy (czyli aktualny katalog roboczy).

Ponieważ ten katalog zawsze przeszukiwany jest jako pierwszy, jeśli program jest w całości umieszczony w jednym katalogu, wszystkie operacje importowania będą działać automatycznie, bez konieczności konfigurowania ścieżki. Z drugiej strony, ponieważ katalog ten przeszukiwany jest jako pierwszy, jego pliki będą także przeciągały moduły o tej samej nazwie znajdujące się w katalogach podanych w innych miejscach ścieżki. Należy uważać, by w ten sposób nie ukryć przypadkowo modułów biblioteki, jeśli będą nam one potrzebne w programie.

Katalogi PYTHONPATH

Następnie Python przeszukuje wszystkie katalogi podane w zmiennej środowiskowej `PYTHONPATH`, od lewej do prawej strony (zakładając, że ją w ogóle ustawiliśmy). Mówiąc w skrócie, w `PYTHONPATH` podaje się listę zdefiniowanych przez użytkownika i specyficznych dla platformy nazw katalogów zawierających pliki z kodem Pythona. Możemy tam dodać wszystkie katalogi, z których chcemy coś importować, a Python rozszerzy ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona wszystkie katalogi podane w `PYTHONPATH`.

Ponieważ Python najpierw przeszukuje katalog główny, to ustawienie ma znaczenie jedynie wtedy, gdy importuje się pliki pomiędzy różnymi katalogami — to znaczy, kiedy musimy zaimportować plik przechowywany w katalogu innym niż katalog pliku go importującego. Najprawdopodobniej będziemy chcieli ustawić zmienną `PYTHONPATH`, kiedy zaczniemy pisać większe programy, jednak na początku, dopóki zapisujemy wszystkie pliki modułów w katalogu, w którym pracujemy (czyli w opisany wcześniej katalogu głównym), operacje importowania powinny działać bez troszczenia się o to ustawienie.

Katalogi biblioteki standardowej

Następnie Python automatycznie przeszukuje katalogi, w których na naszym komputerze zainstalowane są moduły biblioteki standardowej. Ponieważ są one przeszukiwane zawsze, zazwyczaj nie trzeba się martwić o dodanie ich do zmiennej środowiskowej `PYTHONPATH` lub do omówionych niżej plików ścieżek.

Katalogi ścieżek plików .pth

Wreszcie stosunkowo rzadziej wykorzystywana opcja, pozwalająca użytkownikom dodawać katalogi do ścieżki wyszukiwania modułów za pomocą wymienienia ich, po jednym w wierszu, w pliku tekstowym kończącym się rozszerzeniem `.pth` (od ang. `path` — ścieżka). Te pliki konfiguracyjne są raczej zaawansowaną opcją instalacyjną, dlatego nie

będziemy ich tutaj w pełni omawiać. Stanowią one alternatywę dla ustawień zmiennej środowiskowej PYTHONPATH.

W skrócie, pliki tekstowe z nazwami katalogów wstawione do odpowiedniego katalogu mogą pełnić tę samą rolę co zmienna środowiskowa PYTHONPATH. W celu rozszerzenia ścieżki wyszukiwania modułów plik o nazwie *myconfig.pth* można na przykład w systemie Windows, przy korzystaniu z Pythona 3.0, umieścić na najwyższym poziomie katalogu instalacyjnego Pythona (C:\Python30) lub w podkatalogu *site-packages* biblioteki standardowej (C:\Python30\Lib\site-packages). W systemach uniksowych plik ten można natomiast umieścić na przykład w */usr/local/lib/python3.0/site-packages* lub */usr/local/lib/site-python*.

Kiedy plik taki jest obecny, Python doda katalogi wymienione w wierszach pliku, od pierwszego do ostatniego, na końcu listy ścieżki wyszukiwania modułów. Tak naprawdę Python zbierze nazwy katalogów ze wszystkich plików ze ścieżkami, jakie znajdzie, a następnie odfiltruje wszystkie powtarzające się lub nieistniejące katalogi. Ponieważ są to pliki, a nie ustawienia powłoki, pliki ze ścieżkami mogą mieć zastosowanie dla wszystkich użytkowników danej instalacji, a nie tylko dla jednego użytkownika lub powłoki. Co więcej, dla niektórych użytkowników utworzenie plików tekstowych może być łatwiejsze od użycia zmiennych środowiskowych.

Opcja ta jest nieco bardziej zaawansowana, niż wskazywałaby na to powyższy opis. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona (w szczególności dotyczącej modułu *site* — moduł ten pozwala na konfigurację lokalizacji bibliotek Pythona oraz plików ścieżek, a w jego dokumentacji opisano także oczekiwane lokalizacje plików ścieżek). Osobom poczynającym polecam korzystanie z PYTHONPATH lub być może pojedynczego pliku *.pth*, i to tylko wtedy, gdy muszą one importować moduły pomiędzy różnymi katalogami. Pliki ze ścieżkami są częściej wykorzystywane przez biblioteki zewnętrzne, które najczęściej instalują plik tego typu w katalogu *site-packages* Pythona, tak by użytkownik nie musiał wprowadzać żadnych ustawień (system instalacyjny *distutils* Pythona, opisany w ramce poniżej, poddaje automatyzacji wiele kroków instalacyjnych).

Konfiguracja ścieżki wyszukiwania

W rezultacie zarówno PYTHONPATH, jak i komponenty plików ze ścieżkami dla ścieżki wyszukiwania pozwalają dostosować miejsca wyszukiwania importowanych plików do własnych potrzeb. Sposób ustawiania zmiennych środowiskowych i miejsce przechowywania plików ścieżek różnią się dla poszczególnych platform. W systemie Windows można na przykład wykorzystać ikonę *System* Panelu sterowania do ustawienia PYTHONPATH na listę katalogów rozdzielonych średnikami, jak poniżej:

```
c:\pycode\utilities;d:\pycode\package1
```

Można również zamiast tego utworzyć plik tekstowy o nazwie C:\Python30\pydirs.pth i następującej postaci:

```
c:\pycode\utilities
d:\pycode\package1
```

Ustawienia te na innych platformach wyglądają analogicznie, jednak szczegóły mogą się od siebie różnić w zbyt dużym stopniu, byśmy opisywali je tutaj w całości. Przykłady często spotykanych sposobów rozszerzania ścieżki wyszukiwania modułów za pomocą PYTHONPATH oraz plików *.pth* na różnych platformach można znaleźć w dodatku A.

Wariacje ścieżki wyszukiwania modułów

Powyższy opis ścieżki wyszukiwania modułów jest poprawny, jednak bardzo ogólny. Dokładna konfiguracja ścieżki wyszukiwania może na różnych platformach i dla różnych wersji Pythona się różnić. W zależności od platformy dodatkowe katalogi mogą również być dodawane do ścieżki wyszukiwania modułów automatycznie.

Python może na przykład dodać wpis dla *bieżącego katalogu roboczego* — katalogu, z którego uruchomiliśmy program — do ścieżki wyszukiwania po katalogach PYTHONPATH, a przed wpisami biblioteki standardowej. Po uruchomieniu z wiersza poleceń bieżący katalog roboczy może nie być tym samym co katalog roboczy pliku najwyższego poziomu (czyli katalog, w którym znajduje się plik programu). Ponieważ bieżący katalog roboczy może zmieniać się za każdym wykonaniem programu, zazwyczaj nie powinniśmy polegać na jego wartości na potrzeby importowania. Więcej informacji na temat uruchamiania programów z wiersza poleceń znajduje się w rozdziale 3.³

By sprawdzić, jak Python konfiguruje ścieżkę wyszukiwania modułów dla określonej platformy, zawsze można zbadać listę `sys.path` — zagadnienie omówione poniżej.

Lista `sys.path`

Jeśli chcemy zobaczyć, jak na naszym komputerze skonfigurowana jest ścieżka wyszukiwania modułów, zawsze możemy to zobaczyć w takiej samej postaci, jak widzi to Python, wyświetlając wbudowaną listę `sys.path` (czyli atrybut `path` modułu `sys` z biblioteki standardowej). Ta lista łańcuchów znaków nazw katalogów jest prawdziwą ścieżką wyszukiwania w Pythonie. W przypadku operacji importowania Python przeszukuje każdy katalog listy od lewej do prawej strony.

Tak naprawdę `sys.path` jest ścieżką wyszukiwania modułów. Python konfiguruje ją w momencie uruchomienia programu, automatycznie łącząc w listę katalog główny pliku najwyższego poziomu (lub pusty łańcuch znaków oznaczający bieżący katalog roboczy), wszystkie katalogi z PYTHONPATH, zawartość wszelkich utworzonych plików `.pth`, a także katalogi biblioteki standardowej. W rezultacie otrzymujemy listę łańcuchów znaków nazw katalogów, którą Python przeszukuje przy każdej operacji importowania nowego pliku.

Python udostępnia tę listę z dwóch ważnych powodów. Po pierwsze, jest to sposób pozwalający na zweryfikowanie wprowadzonych ustawień ścieżki — jeśli naszych ustawień na tej liście nie będzie, będziemy musieli sprawdzić swoją pracę. Przykładowo poniżej widać, jak wygląda moja ścieżka wyszukiwania modułów w systemie Windows, dla Pythona 3.0, ze zmienną środowiskową PYTHONPATH ustawioną na `C:\users` oraz plikiem ścieżki `C:\Python30\mypath.py` zawierającym katalog `C:\users\mark`. Pusty łańcuch znaków na początku oznacza, że połączony został katalog bieżący oraz moje dwa ustawienia (reszta to katalogi i pliki biblioteki standardowej):

³ Więcej informacji na ten temat znajduje się w omówieniu nowej *składni importowania względnego* z Pythona 3.0 w rozdziale 23. Modyfikuje ona ścieżkę wyszukiwania dla instrukcji `from`, kiedy użyte zostaną znaki kropki (na przykład `from . import string`). Domyślnie w Pythonie 3.0 własny katalog pakietu nie jest przeszukiwany, o ile importowanie względne nie jest wykorzystywane przez pliki w samym pakiecie.

```
>>> import sys
>>> sys.path
[ '', 'C:\\\\users', 'C:\\Windows\\\\system32\\\\python30.zip', 'c:\\\\Python30\\\\DLLs',
  'c:\\\\Python30\\\\lib', 'c:\\\\Python30\\\\lib\\\\plat-win', 'c:\\\\Python30',
  'C:\\\\Users\\\\Mark', 'c:\\\\Python30\\\\lib\\\\site-packages' ]
```

Po drugie, jeśli wiemy, co robimy, lista ta umożliwia również skryptom ręczne dostosowywanie ścieżek wyszukiwania. Jak zobaczymy później w dalszej części książki, modyfikując `sys.path`, możemy również zmodyfikować ścieżkę wyszukiwania dla wszystkich przyszłych importów. Takie zmiany trwają jednak tylko do końca czasu trwania skryptu; `PYTHONPATH` i pliki `.pth` oferują bardziej trwałe sposoby zmodyfikowania tej ścieżki.⁴

Wybór pliku modułu

Należy pamiętać, że rozszerzenia plików (na przykład `.py`) są w instrukcjach `import` celowo pomijane. Python wybiera pierwszy plik odpowiadający importowanej nazwie, jaki może znaleźć na ścieżce wyszukiwania. Instrukcja `import` w postaci `import b` może załadować:

- Plik z kodem źródłowym o nazwie `b.py`.
- Plik z kodem bajtowym nazwie `b.pyc`.
- Katalog o nazwie `b` w przypadku importowania pakietów (opisanego w rozdziale 23.).
- Skompilowany moduł rozszerzenia, zazwyczaj napisany w językach C lub C++ i dynamicznie dołączany w momencie importowania (na przykład `b.so` w systemie Linux, `b.dll` lub `b.pyd` dla środowisk Cygwin i Windows).
- Skompilowany moduł wbudowany napisany w języku C i statycznie dołączony do Pythona.
- Komponent pliku ZIP rozpakowywany automatycznie po zimportowaniu.
- Obraz z pamięci w przypadku zamrożonych plików wykonywalnych.
- Klasę języka Java w wersji Pythona o nazwie Jython.
- Komponent .NET w wersji Pythona o nazwie IronPython.

Rozszerzenia języka C, Jython oraz importowanie pakietów rozszerzają operacje importowania poza proste pliki. Dla kodu importującego różnice w typach załadowanych plików są jednak całkowicie niewidoczne, zarówno przy importowaniu, jak i przy pobieraniu atrybutów modułów. Instrukcja `import b` pobiera dowolny moduł `b`, czymkolwiek by on nie był, zgodnie ze ścieżką wyszukiwania modułów, natomiast `b.atrybut` pobiera element z tego modułu — obojętnie, czy jest to zmienna Pythona, czy dołączona funkcja języka C. Niektóre moduły biblioteki standardowej, z których będziemy korzystać w książce, są tak naprawdę napisane w języku C, a nie w Pythonie. Ponieważ jest to niewidoczne dla kodu importującego, nie ma to żadnego znaczenia.

Jeśli mamy zarówno plik `b.py`, jak i `b.so` w różnych katalogach, Python zawsze załadowuje ten znaleziony w pierwszym (znajdującym się bardziej po lewej stronie) katalogu ścieżki wyszukiwania

⁴ Niektóre programy naprawdę muszą jednak modyfikować `sys.path`. Przykładowo skrypty działające na serwerach WWW zazwyczaj działają jako użytkownik „nobody” w celu ograniczenia dostępu do komputera. Ponieważ takie skrypty nie mogą zwykle polegać na ustawianiu `PYTHONPATH` w określony sposób przez użytkownika „nobody”, często muszą ręcznie ustawać `sys.path` w celu dodania wymaganych katalogów źródłowych przed wykonaniem jakichkolwiek instrukcji `import`. Często wystarczy użyć `sys.path.append` (`nazwa_katalogu`).

modułów w czasie przeszukiwania listy `sys.path` od lewej do prawej strony. Co się jednak dzieje, jeśli Python znajdzie zarówno `b.py`, jak i `b.so` w tym samym katalogu? W takim przypadku Python postępuje zgodnie ze standardową kolejnością wybierania, choć nie ma żadnej gwarancji, że kolejność ta nie zmieni się w przyszłości. Nie powinniśmy zatem polegać na tym, który plik o tej samej nazwie w danym katalogu wybierze Python. Lepiej jest nadawać plikom inne nazwy lub skonfigurować ścieżkę wyszukiwania modułów w taki sposób, by nasze preferencje w zakresie wybierania modułów były bardziej oczywiste.

Zaawansowane zagadnienia związane z wyborem modułów

Normalnie importowanie działa tak, jak opisano to w niniejszym podrozdziale. Python odszukuje i ładuje pliki na nasz komputer. Można jednak redefiniować część tego, co w Pythonie robi operacja importowania, za pomocą czegoś, co znane jest pod nazwą *punktów zaczepienia operacji importowania* (ang. *import hook*). Te punkty zaczepienia można wykorzystać do importowania przydatnych elementów, na przykład ładowania plików z archiwów czy odszyfrowywania.

Tak naprawdę sam Python wykorzystuje punkty zaczepienia w celu umożliwienia bezpośredniego importu plików z archiwów ZIP — zarchiwizowane pliki są automatycznie pobierane w czasie importowania, kiedy plik `.zip` zostanie wybrany w ścieżce wyszukiwania importowanych modułów. Jeden z katalogów biblioteki standardowej w powyższym kodzie wyświetlaczącym zawartość `sys.path` był na przykład plikiem `.zip`. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona, w opisie wbudowanej funkcji `__import__` — narzędzia, które można dostosowywać do własnych potrzeb, a które tak naprawdę wykonywane jest przez instrukcję `import`.

Python obsługuje również pojęcie zoptymalizowanych plików z kodem bajtowym (`.pyo`), dworzonych i wykonywanych z opcją wiersza poleceń Pythona `-O`. Ponieważ działają one tylko odrobinę szybciej od normalnych plików `.pyc` (zazwyczaj pięć procent szybciej), są rzadko używane. System Psyco (opisany w rozdziale 2.) umożliwia o wiele większe przyspieszenie wykonywania kodu.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy podstawy modułów, atrybutów oraz importowania, a także działanie instrukcji `import`. Zobaczyliśmy, że operacje importowania odnajdują plik docelowy w ścieżce wyszukiwania modułów, komplikują go do kodu bajtowego i wykonują wszystkie jego instrukcje w celu wygenerowania zawartości. Dowiedzieliśmy się również, w jaki sposób — przede wszystkim za pomocą ustawień zmiennej środowiskowej `PYTHONPATH` — konfiguruje się ścieżkę wyszukiwania, tak by móc importować z innych katalogów poza katalogiem głównym oraz katalogami biblioteki standardowej.

Jak pokazał niniejszy rozdział, operacja importowania i moduły są sercem architektury programów Pythona. Większe programy podzielone są na większą liczbę plików łączonych razem przez operacje importowania w czasie wykonywania. Importowanie wykorzystuje z kolei ścieżkę wyszukiwania modułów do lokalizacji plików, a moduły definiują atrybuty do wykorzystania przez pliki zewnętrzne.

Dodatkowe oprogramowanie — distutils

Opis ustawień ścieżki wyszukiwania modułów jest przeznaczony przede wszystkim dla pisanej przez nas kodu źródłowego zdefiniowanego przez użytkownika. Rozszerzenia zewnętrzne do Pythona zazwyczaj wykorzystują narzędzia `distutils` z biblioteki standardowej do automatycznej instalacji, dzięki czemu nie jest wymagane dodatkowe konfigurowanie ścieżki, by móc korzystać z ich kodu.

Systemy wykorzystujące `distutils` najczęściej zawierają skrypt `setup.py` wykonywany w celu ich zainstalowania. Skrypt ten importuje i wykorzystuje moduły `distutils` w celu umieszczenia systemu w katalogu stojącym się automatycznie częścią ścieżki wyszukiwania modułów (zazwyczaj w podkatalogu `Lib\site-packages` drzewa instalacyjnego Pythona, gdziekolwiek się on nie znajduje na komputerze docelowym).

Więcej informacji dotyczących dystrybucji i instalacji za pomocą `distutils` można znaleźć w dokumentacji biblioteki standardowej Pythona. Wykorzystywanie tych narzędzi pozostaje poza zakresem niniejszej książki (pozwalają one na przykład automatycznie kompilować rozszerzenia napisane w języku C na komputerze docelowym). Można również zapoznać się z systemem zewnętrznym na licencji open source o nazwie `eggs`, który dodaje do zainstalowanego oprogramowania napisanego w Pythonie możliwość sprawdzania zależności.

Oczywiście cały cel importowania oraz modułów polega na nadaniu programom struktury dzielącej ich logikę na samodzielne komponenty oprogramowania. Kod z jednego modułu jest izolowany od kodu z pozostałych modułów. Tak naprawdę żaden plik nie może zobaczyć nazw zdefiniowanych w innym, dopóki w jawnym sposób nie wykona się instrukcji `import`. Z tego powodu moduły minimalizują konflikty pomiędzy różnymi częściami programu w zakresie nazw zmiennych.

W kolejnym rozdziale przekonamy się, co to wszystko znaczy w praktyce, na przykładzie prawdziwego kodu. Zanim jednak przejdziemy dalej, zajmijmy się quizem dotyczącym niniejszego rozdziału.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób plik z kodem źródłowym modułu staje się obiektem modułu?
2. Po co ustawia się zmienną środowiskową `PYTHONPATH`?
3. Należy wymienić cztery główne komponenty ścieżki wyszukiwania importowanych modułów.
4. Należy podać cztery typy plików, jakie Python może załadować w odpowiedzi na operację importowania.
5. Czym jest przestrzeń nazw i co zawiera przestrzeń nazw modułu?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik z kodem źródłowym automatycznie staje się obiektem modułu po zainportowaniu tego modułu. Z technicznego punktu widzenia kod źródłowy modułu jest wykonywany w czasie operacji importowania, po jednej instrukcji na raz, a wszystkie nazwy przypisane w tym czasie stają się atrybutami obiektu modułu.
2. Zmienną `PYTHONPATH` ustawia się tylko po to, by móc importować z katalogów innych niż katalog roboczy (czyli katalog bieżący w czasie pracy w sesji interaktywnej lub katalog zawierający plik najwyższego poziomu).
3. Cztery podstawowe komponenty ścieżki wyszukiwania importowanych modułów to katalog główny skryptu najwyższego poziomu (katalog zawierający ten skrypt), wszystkie katalogi wymienione w zmiennej środowiskowej `PYTHONPATH`, katalogi biblioteki standardej oraz wszystkie katalogi wymienione w plikach ścieżek `.pth` umieszczonych w standardowych miejscach. Z tego wszystkiego programiści mogą dostosowywać do własnych potrzeb zmienną `PYTHONPATH` oraz pliki `.pth`.
4. Python może załadować plik z kodem źródłowym (`.py`), kodem bajtowym (`.pyc`), moduł rozszerzenia w języku C (na przykład plik `.so` w systemie Linux lub `.dll` albo `.pyd` w systemie Windows) lub katalog o danej nazwie w przypadku importowania pakietów. Operacje importowania mogą również ładować bardziej egzotyczne komponenty, takie jak części archiwów ZIP, klasy języka Java w wersji Jython, komponenty platformy .NET w wersji IronPython, a także statycznie dołączone rozszerzenia w języku C, które nie mają żadnych plików. Dzięki punktom zaczepienia operacji importowania można ładować prawie wszystko.
5. Przestrzeń nazw to samodzielny pakiet zmiennych znanych jako *atrybuty* obiektu przestrzeni nazw. Przestrzeń nazw modułu zawiera wszystkie nazwy przypisane na najwyższym poziomie pliku modułu (czyli niezagnieżdżone w instrukcjach `def` lub `class`). Z technicznego punktu widzenia zakres globalny modułu staje się przestrzenią nazw atrybutów obiektu modułu. Przestrzeń nazw modułu można również modyfikować za pomocą operacji przypisania w plikach innych od importujących moduł, co wielu osobom się nie podoba (więcej informacji na ten temat znajduje się w rozdziale 17.).

Podstawy tworzenia modułów

Skoro już przyjrzeliśmy się podstawowym koncepcjom stojącym za modułami, czas przejść do prostych przykładów modułów w działaniu. Moduły Pythona łatwo jest *utworzyć* — są to po prostu pliki z kodem w języku Python utworzone za pomocą edytora tekstowego. Nie musimy pisać żadnej specjalnej składni, by przekazać Pythonowi, że tworzymy moduł — wystarczy dowolny plik tekstowy. Ponieważ Python sam radzi sobie ze szczegółami odnajdywania i ładowania modułów, są one również bardzo proste w *użyciu*. Klient po prostu importuje moduł lub poszczególne zmienne przez niego definiowane i wykorzystuje obiekty, do których zmienne te się odnoszą.

Tworzenie modułów

By zdefiniować moduł, wystarczy wykorzystać posiadany edytor tekstu do wpisania kodu Pythona do pliku tekstowego i zapisać ten plik z rozszerzeniem *.py*. Każdy taki plik jest automatycznie uznawany za moduł Pythona. Wszystkie zmienne przypisane na najwyższym poziomie modułu stają się jego *atrybutami* (zmiennymi powiązanymi z obiektem modułu) i są eksportowane do klientów w celu użycia.

Jeśli na przykład wpiszemy poniższą instrukcję `def` do pliku o nazwie *module1.py*, a następnie zaimportujemy ten plik, utworzymy obiekt modułu z jednym atrybutem — zmienną `printer`, która okazuje się referencją do obiektu funkcji.

```
def printer(x):                                # Atrybut modułu
    print(x)
```

Zanim przedziemy dalej, powinieneś powiedzieć kilka słów na temat nazw plików modułów. Moduły możemyazywać w dowolny sposób, jednak jeśli planujemy je w przyszłości importować, ich pliki powinny mieć rozszerzenie *.py*. Rozszerzenie to jest opcjonalne dla plików najwyższego poziomu, które będziemy wykonywać, ale nie importować — dodanie *.py* w każdym przypadku sprawia, że typ pliku będzie bardziej oczywisty, i w przyszłości pozwoli zaimportować wszystkie pliki.

Ponieważ nazwy modułów (bez rozszerzenia *.py*) stają się wewnętrz programu Pythona nazwami zmiennych, powinny być zgodne z normalnymi regułami dotyczącymi nazw zmiennych, przedstawionymi w rozdziale 11. Możemy na przykład utworzyć moduł o nazwie *if.py*, jednak nie możemy go zaimportować, ponieważ `if` jest słowem zarezerwowanym — kiedy spróbujemy wykonać instrukcję `import if`, otrzymamy błąd składni. Tak naprawdę zarówno nazwy plików

modułów, jak i nazwy katalogów wykorzystywanych w importowaniu pakietów (omówionych w kolejnym rozdziale) muszą być zgodne z regułami dotyczącymi nazw zmiennych zaprezentowanymi w rozdziale 11. Mogą one w związku z tym zawierać wyłącznie litery, cyfry oraz znaki `_`. Nazwy katalogów pakietów nie mogą również zawierać żadnej składni specyficznej dla określonej platformy, w tym na przykład spacji.

Kiedy moduł jest importowany, Python odwzorowuje wewnętrzną nazwę modułu na zewnętrzną nazwę pliku, dodając ścieżkę do katalogu ze ścieżki wyszukiwania modułów na początku, a rozszerzenie `.py` (lub inne) na końcu. Moduł o nazwie `M` może na przykład zostać odwzorowany na jakiś plik zewnętrzny `<katalog>\M.<rozszerzenie>` zawierający kod modułu.

Jak wspomniano w poprzednim rozdziale, można również utworzyć moduł Pythona, pisząc kod w języku takim, jak C czy C++ (lub w przypadku implementacji Jython — również w Javie). Takie moduły znane są jako *moduły rozszerzeń* i są najczęściej używane do opakowania zewnętrznych bibliotek do użycia w skryptach Pythona. Po zaimportowaniu przez kod Pythona moduły rozszerzeń wyglądają i działają tak samo jak moduły utworzone jako pliki z kodem źródłowym Pythona — dostęp do nich odbywa się za pomocą instrukcji `import` i udostępniają one funkcje oraz obiekty jako atrybuty modułu. Moduły rozszerzeń wykraczają poza zakres niniejszej książki; więcej informacji na ich temat można znaleźć w dokumentacji Pythona lub tekstu zaawansowanych, takich jak *Programming Python*.

Użycie modułów

Klient może wykorzystywać napisane przez nas proste pliki modułów, wykorzystując do tego instrukcje `import` lub `from`. Obie instrukcje odnajdują, komplują oraz wykonują kod pliku modułu, jeśli nie został on jeszcze załadowany. Podstawowa różnica polega na tym, że instrukcja `import` pobiera moduł jako całość, zatem by pobrać poszczególne nazwy, musimy skorzystać ze składni kwalifikującej (z kropką). W przeciwnieństwie do tego instrukcja `from` pobiera (lub kopiuje) określone zmienne z modułu.

Zobaczmy, co to wszystko oznacza dla kodu. Wszystkie poniższe przykłady w rezultacie wywołują funkcję `printer` zdefiniowaną w przedstawionym wyżej pliku modułu `module1.py`, jednak robią to w różny sposób.

Instrukcja `import`

W pierwszym przykładzie nazwa `module1` spełnia dwa różne cele — identyfikuje plik zewnętrzny, który ma zostać załadowany, i staje się zmienną w skrypcie, będącą po załadowaniu pliku odniesieniem do obiektu modułu.

```
>>> import module1                                # Pobranie modułu jako całości
>>> module1.printer('Witaj, świecie!')            # Zapis z kropką w celu otrzymania zmiennej
Witaj, świecie!
```

Ponieważ instrukcja `import` daje nazwę, która odnosi się do całego obiektu modułu, w celu pobrania atrybutów modułu musimy uwzględnić jego nazwę (na przykład `module1.printer`).

Instrukcja from

W przeciwnieństwie do `import`, instrukcja `from` kopiuje zmienne z jednego pliku do innego zakresu, dlatego pozwala na bezpośrednie używanie nazw zmiennych w skrypcie, bez dodawania do nich nazwy modułu (wystarczy zatem użyć samego `printer`).

```
>>> from module1 import printer          # Skopiowanie jednej zmiennej  
>>> printer('Witaj, świecie!')           # Nie ma konieczności użycia nazwy modułu  
Witaj, świecie!
```

Ma to taki sam efekt jak poprzedni przykład, jednak ponieważ importowana zmienna jest kopiwana do zakresu, w którym pojawia się instrukcja `from`, użycie tej zmiennej w skrypcie nie wymaga tyle pisania — nazwy zmiennej możemy użyć bezpośrednio, zamiast podawać jednocześnie nazwę zawierającego ją modułu.

Jak zobaczymy w szczegółach później, instrukcja `from` jest jedynie nieznacznym rozszerzeniem instrukcji `import` — importuje ona plik modułu tak jak zawsze, jednak dodaje dodatkowy krok kopiący jedną zmienną (lub większą ich liczbę) z tego pliku.

Instrukcja from *

Kolejny przykład wykorzystuje specjalną formę instrukcji `from`. Kiedy użyjemy znaku `*`, otrzymujemy kopie *wszystkich* zmiennych przypisanych na najwyższym poziomie modułu. Tutaj możemy znowu wykorzystać nazwę `printer` w skrypcie bez dodawania do niej nazwy modułu.

```
>>> from module1 import *                  # Skopiowanie wszystkich zmiennych  
>>> printer('Witaj, świecie!')  
Witaj, świecie!
```

Z technicznego punktu widzenia instrukcje `import` oraz `from` wywołują tę samą operację importowania. Forma `from *` po prostu dodaje jeszcze jeden krok kopiący wszystkie zmienne z modułu do zakresu importującego. W rezultacie przestrzeń nazw jednego modułu zostaje tak naprawdę włączona do przestrzeni nazw drugiego. Oznacza to dla nas mniej pisania.

I to tyle — moduły są naprawdę proste w użyciu. By jednak lepiej zrozumieć, co tak naprawdę dzieje się, kiedy definiujemy i wykorzystujemy moduły, przyjrzyjmy się bardziej szczegółowo niektórym ich właściwościom.



W Pythonie 3.0 opisana tutaj postać `from ... *` może być zastosowana *jedynie* na najwyższym poziomie pliku modułu, a nie wewnętrz funkcji. Python 2.6 pozwala na użycie jej wewnętrz funkcji, jednak wyświetla ostrzeżenie. W praktyce użycie tej instrukcji wewnętrz funkcji spotyka się wyjątkowo rzadko. Kiedy tak się dzieje, Python nie jest w stanie wykrywać zmiennych statycznie, przed wykonaniem funkcji.

Operacja importowania odbywa się tylko raz

Jednym z pytań najczęściej zadawanych przez osoby początkujące wykorzystujące moduły jest na ogół: „Dlaczego moje importy już nie działają?”. Często zgłaszą one, że pierwsza operacja importowania działa dobrze, natomiast późniejsze w czasie sesji interaktywnej (lub wykonywania programu) wydają się nie przynosić efektu. Tak naprawdę wcale nie powinny przynosić i zaraz wyjaśnimy, dlaczego tak jest.

Moduły są ładowane i wykonywane tylko przy pierwszej instrukcji `import` lub `from`. Jest to celowe — ponieważ importowanie jest kosztowną operacją, Python domyślnie wykonuje ją tylko raz na plik, raz na proces. Późniejsze operacje importowania po prostu pobierają już załadowany obiekt modułu.

W konsekwencji, ponieważ kod najwyższego poziomu w pliku modułu jest zazwyczaj wykonywany tylko raz, możemy go wykorzystać do inicjalizacji zmiennych. Rozważmy na przykład plik `simple.py`.

```
print('witam')  
spam = 1  
# Inicjalizacja zmiennej
```

W powyższym przykładzie instrukcje `print` oraz `=` wykonywane są za pierwszym razem, gdy moduł jest importowany, natomiast zmienna `spam` inicjalizowana jest w czasie importowania.

```
% python  
>>> import simple  
witam  
>>> simple.spam  
1  
# Pierwszy import: ładuje i wykonuje kod pliku  
# Przypisanie tworzy atrybut
```

Druga operacja importowania i kolejne nie wykonują kodu modułu ponownie, a po prostu pobierają już utworzony obiekt modułu z wewnętrznej tabeli modułów Pythona. Z tego powodu zmienna `spam` nie zostaje zainicjalizowana ponownie.

```
>>> simple.spam = 2  
>>> import simple  
>>> simple.spam  
2  
# Modyfikacja atrybutu w module  
# Pobiera już załadowany moduł  
# Kod nie został wykonany ponownie — atrybut nie zmienia się
```

Oczywiście czasami naprawdę chcemy, by kod modułu był wykonany ponownie w kolejnej operacji importowania. W dalszej części rozdziału zobaczymy, jak można to uzyskać za pomocą funkcji Pythona `reload`.

Instrukcje `import` oraz `from` są przypisaniami

Tak jak `def`, instrukcje `import` oraz `from` są instrukcjami wykonywalnymi, a nie deklaracjami w czasie komplikacji. Mogą one być zagnieżdżone w testach `if` czy pojawić się w instrukcjach `def` funkcji i nie są wykonywane, dopóki Python nie dojdzie do nich w czasie wykonywania programu. Innymi słowy, importowane moduły oraz zmienne nie są dostępne, dopóki nie zostaną wykonane powiązane z nimi instrukcje `import` lub `from`. Podobnie jak `def`, `import` oraz `from` są niejawnymi przypisaniami.

- Instrukcja `import` przypisuje cały obiekt modułu do jednej nazwy.
- Instrukcja `from` przypisuje jedną lub więcej zmiennych do obiektów o tych samych nazwach w innym module.

Wszystkie omówione dotychczas kwestie dotyczące przypisania mają zastosowanie również do dostępu do modułu. Zmienna skopiowana za pomocą instrukcji `from` staje się na przykład referencją do współdzielonego obiektu. Tak jak w przypadku argumentów funkcji, ponowne przypisanie zmiennej nie ma wpływu na moduł, z którego została ona skopiowana, jednak modyfikacja pobranego obiektu *zmennego* może spowodować jego zmianę w module, z którego został on zimportowany. By to zilustrować, rozważmy poniższy plik `small.py`.

```

x = 1
y = [1, 2]

% python
>>> from small import x, y           # Skopiowanie dwóch zmiennych z modułu
>>> x = 42                          # Modyfikacja jedynie lokalnej zmiennej x
>>> y[0] = 42                        # Modyfikacja współdzielonego obiektu zmiennego w miejscu

```

W powyższym kodzie `x` nie jest współdzielonym obiektem zmiennym, natomiast `y` nim jest. Zmienna `y` w kodzie importującym i w kodzie importowanym odnoszą się do tego samego obiektu listy, dlatego jego modyfikacja w jednym miejscu powoduje zmianę w innym.

```

>>> import small
>>> small.x                         # Pobranie nazwy modułu (from tego nie robi)
1                                         # x z modułu small nie jest moim x
>>> small.y                         # Współdzielimy jednak zmienny obiekt
[42, 2]

```

By lepiej sobie wyobrazić, co przypisania `from` robią z referencjami, warto wrócić do rysunku 18.1 przedstawiającego przekazywanie argumentów funkcji i zastąpić w nim „wywołującego” oraz „funkcję”, „importowanym” i „importującym”. Rezultat będzie taki sam, tyle że tutaj mamy do czynienia ze zmiennymi w modułach, a nie w funkcjach. Przypisanie działa w Pythonie tak samo w każdych okolicznościach.

Modyfikacja zmiennych pomiędzy plikami

W poprzednim przykładzie widać było, że przypisanie do `x` w sesji interaktywnej modyfikowało zmienną `x` tylko w tym zakresie, a nie miało wpływu na zmienną `x` z pliku. Nie istnieje żadne łącze pomiędzy zmienną skopowaną za pomocą instrukcji `from` a plikiem, z którego zmienna ta pochodzi. By naprawdę zmodyfikować zmienną globalną z innego pliku, musimy użyć instrukcji `import`.

```

% python
>>> from small import x, y           # Skopiowanie dwóch zmiennych z modułu
>>> x = 42                          # Modyfikacja jedynie lokalnej zmiennej x

>>> import small
>>> small.x = 42                    # Pobranie nazwy modułu
                                         # Modyfikuje zmienną x w innym module

```

To zjawisko zostało pokazane w rozdziale 17. Ponieważ modyfikacja zmiennych w innych modułach jest znanym źródłem nieporozumień (a często również złym wyborem projektowym), powrócimy do tej techniki w dalszej części książki. Warto zauważyć, że zmiana `y[0]` w poprzedniej sesji jest inna — modyfikuje ona obiekt, a nie nazwę zmiennej.

Równoważność instrukcji `import` oraz `from`

Warto zwrócić uwagę na to, że w poprzednim przykładzie musieliśmy wykonać instrukcję `import` po instrukcji `from` w celu uzyskania dostępu do nazwy modułu `small`. Instrukcja `from` kopiuje jedynie zmienne z jednego modułu do drugiego i nie przypisuje samej nazwy modułu. Przynajmniej z koncepcjonalnego punktu widzenia poniższa instrukcja `from`:

```

from module import name1, name2          # Skopiowanie tylko tych dwóch zmiennych z modułu
                                         # Pobranie obiektu modułu
jest odpowiednikiem poniższej sekwencji instrukcji:                                # Skopiowanie zmiennej przez przypisanie

import module
name1 = module.name1

```

```
name2 = module.name2  
del module  
# Pozbycie się nazwy modułu
```

Jak wszystkie przypisania, instrukcje `from` tworzą w kodzie importującym nowe zmienne, które początkowo odnoszą się do obiektów o tych samych nazwach w pliku importowanym. Kopowane są jednak jedynie zmienne, a nie sam moduł. Kiedy skorzystamy z formy `from *` (`from module import *`), równoważność instrukcji pozostaje bez zmian, jednak w ten sposób do zakresu importującego kopowane są wszystkie zmienne najwyższego poziomu.

Warto zwrócić uwagę na to, że pierwszy krok instrukcji `from` wykonuje normalną operację `import`. Z tego powodu `from` zawsze importuje cały moduł do pamięci (jeśli jeszcze nie został on zimportowany), bez względu na to, ile zmiennych kopujemy z pliku. Nie da się załadować jedynie części pliku modułu (na przykład jednej funkcji), jednak ponieważ moduły są w Pythonie kodem bajtowym, a nie maszynowym, wpływ takiego rozwiązania na wydajność jest pomijany.

Potencjalne pułapki związane z użyciem instrukcji `from`

Ponieważ instrukcja `from` sprawia, że lokalizacja zmiennych staje się mniej jawną i bardziej niejasna (zmienna `name` jest mniej zrozumiała dla osoby czytającej kod od zmiennej `module.name`), niektórzy użytkownicy Pythona polecają zamiast `from` używać najczęściej instrukcji `import`. Nie jestem jednak pewien, czy taka rada jest uzasadniona — instrukcja `from` jest powszechnie używana bez jakichś poważniejszych negatywnych konsekwencji. W praktyce, w prawdziwych programach, często wygodniej jest nie musieć wpisywać pełnej nazwy modułu za każdym razem, kiedy chcemy skorzystać z jego narzędzi. Jest to szczególnie prawdziwe w przypadku dużych modułów udostępniających wiele atrybutów — na przykład modułu graficznego interfejsu użytkownika `tkinter` z biblioteki standardowej.

To prawda, że instrukcja `from` może potencjalnie zniszczyć przestrzenie nazw, przynajmniej teoretycznie — jeśli będziemy ją wykorzystywać do importowania zmiennych, które mają takie same nazwy jak istniejące zmienne z naszego zakresu, zmienne te zostaną po cichu nadpisane. Ten problem nie ma miejsca w przypadku prostych instrukcji `import`, ponieważ w ich przypadku zawsze trzeba przejść nazwę modułu w celu dotarcia do jego zawartości (`module.` → `attr` nie będzie w konflikcie ze zmienną o nazwie `attr` w naszym zakresie). Dopóki rozumiemy ten sposób działania instrukcji `from` i jesteśmy na niego przygotowani, nie jest to w praktyce wielkim problemem, w szczególności kiedy w jasny sposób wymienimy importowane zmienne (na przykład `from module import x, y, z`).

Z drugiej strony, instrukcja `from` powoduje poważniejsze problemy, kiedy używa się jej w połączeniu z wywołaniem funkcji `reload`, gdyż importowane zmienne mogą odnosić się do wcześniejszych wersji obiektów. Co więcej, forma `from module import *` naprawdę może zniszczyć przestrzenie nazw i sprawić, że zmienne będą trudne do zrozumienia — w szczególności kiedy zastosuje się ją do większej liczby plików. W takim przypadku nie da się powiedzieć, z którego modułu pochodzi zmienna — poza, oczywiście, przeszukaniem zewnętrznych plików źródłowych. W rezultacie forma ze znakiem `*` włącza jedną przestrzeń nazw do drugiej i tym samym niszczy cechę modułów, jaką jest podział przestrzeni nazw. Kwestie te omówimy bardziej szczegółowo w podrozdziale „Pułapki związane z modułami” na końcu tej części książki (w rozdziale 24.).

Chyba najlepszą radą, jakiej można udzielić w tym temacie, jest wybieranie `import` w przypadku prostych modułów, jawne wymienianie importowanych zmiennych w większości instrukcji `from` i ograniczenie formy `from *` do jednej operacji importowania na plik. W ten sposób możemy zakładać, że wszystkie niezdefiniowane zmienne pochodzą z modułu, do którego odnosimy się za pomocą formy `from *`. Wykorzystywanie instrukcji `from` wymaga nieco uwagi, jednak mając odrobinę wiedzy, większość programistów uważa, że jest to wygodny sposób dostępu do modułów.

Kiedy wymagane jest stosowanie instrukcji `import`

Instrukcji `import` w miejsce `from` trzeba używać jedynie wtedy, gdy musimy wykorzystać zmienną o tej samej nazwie, zdefiniowaną w dwóch różnych modułach. Jeśli na przykład dwa pliki inaczej definiują tę samą zmienną:

```
# Plik M.py
def func():
    ...coś robi...
```

```
# Plik N.py
def func():
    ...robi coś innego...
```

i musimy w programie wykorzystać obie wersje tej zmiennej, instrukcja `from` nie przyda się nam do niczego — w jednym zakresie możemy mieć tylko jedno przypisanie do zmiennej.

```
# Plik O.py
from M import func
from N import func
func()                                # Nadpisuje zmienną otrzymaną z modułu M
                                         # Wywołuje tylko N.func
```

W tym przypadku zadziała jednak instrukcja `import`, ponieważ uwzględnienie nazwy modułu sprawia, że zmienne stają się unikalne.

```
# Plik O.py
import M, N
M.func()                                # Pobranie całych modułów, nie ich zmiennych
N.func()                                # Teraz możemy wywołać obie zmienne
                                         # Dzięki nazwom modułów są one unikalne
```

Taka sytuacja zdarza się na tyle rzadko, że w praktyce mało prawdopodobne jest, iż się z nią spotkamy. Jeśli tak się jednak stanie, użycie instrukcji `import` pozwala zapobiec konfliktowi między nazwami zmiennych.

Przestrzeń nazw modułów

Moduły najlepiej jest sobie chyba wyobrazić jako pakiety zmiennych, czyli miejsca, w których definiujemy zmienne, jakie chcemy pokazać reszcie systemu. Z technicznego punktu widzenia moduły zazwyczaj odpowiadają plikom, a Python tworzy obiekt modułu zawierający wszystkie zmienne przypisane w pliku modułu. W uproszczeniu moduły są jedynie przestrzeniami nazw (miejscami, w których tworzone są zmienne), a zmienne istniejące w module nazywane są jego *atrybutami*. W niniejszym podrozdziale wyjaśnimy, jak to wszystko działa.

Pliki generują przestrzenie nazw

W jaki sposób pliki stają się przestrzeniami nazw? Mówiąc w uproszczeniu, każda zmienna, do której na najwyższym poziomie pliku (czyli poza zagnieżdżeniem w ciele funkcji lub klasy) przypisywana jest wartość, staje się atrybutem modułu.

Kiedy mamy instrukcję przypisania `X = 1` na najwyższym poziomie pliku modułu `M.py`, zmienna `X` staje się atrybutem modułu `M`, do którego możemy się spoza modułu odnosić przez `M.X`. Zmienna `X` staje się również zmienną globalną dla pozostałego kodu wewnątrz modułu `M.py`, jednak żeby to zrozumieć, musimy wyjaśnić nieco bardziej szczegółowo pojęcie ładowania modułów oraz zakresów.

- **Instrukcje modułów wykonywane są przy pierwszej operacji importowania.** Za pierwszym razem, gdy moduł jest importowany w dowolnym miejscu systemu, Python tworzy pusty obiekt modułu i wykonuje instrukcje z pliku modułu jedna po drugiej, od góry do dołu.
- **Przypisania na najwyższym poziomie pliku tworzą atrybuty modułów.** W czasie operacji importowania instrukcje przypisujące zmienne na najwyższym poziomie pliku (niezagnieżdżone w funkcji lub klasie), takie jak `def` czy `=`, tworzą atrybuty obiektu modułu. Przypisane nazwy przechowywane są w przestrzeni nazw modułu.
- **Dostęp do przestrzeni nazw modułu odbywa się za pomocą atrybutu `_dict_` lub `dir(M)`.** Przestrzeń nazw modułów utworzone przez operacje importowania są słownikami. Dostęp do nich można uzyskać za pomocą wbudowanego atrybutu `_dict_` powiązanego z obiektami modułów, a można go sprawdzić za pomocą funkcji `dir`. Funkcja `dir` jest przybliżonym odpowiednikiem listy posortowanych kluczy atrybutu `_dict_` obiektu, jednak zawiera odziedziczone zmienne klas, może nie być kompletna i może się zmieniać pomiędzy wydaniami Pythona.
- **Moduły są jednym zakresem (lokalne jest w nich globalny).** Jak widzieliśmy w rozdziale 17., zmienne znajdujące się na najwyższym poziomie pliku modułu poddają się tym samym regułom przypisania i referencji co zmienne funkcji, jednak zakresy lokalny i globalny są tym samym (z formalnego punktu widzenia są zgodne z regułą zakresów LEGB z rozdziału 17., jednak bez warstw wyszukiwania `L` oraz `E`). W modułach `zakres` modułu staje się słownikiem atrybutów *obiektu* modułu po załadowaniu tego modułu. W przeciwieństwie do funkcji (gdzie zakres lokalny istnieje tylko na czas wykonywania funkcji) zakres pliku modułu staje się przestrzenią nazw atrybutów obiektu modułu i istnieje także po operacji importowania.

Poniżej znajduje się demonstracja tych koncepcji. Założmy, że utworzymy następujący plik modułu w edytorze tekstu i nazwiemy go `module2.py`.

```
print('rozpoczęto ładowanie...')  
import sys  
name = 42  
  
def func(): pass  
  
class klass: pass  
  
print('zakończono ładowanie.')
```

Przy pierwszym importie modułu (lub wykonaniu go jako programu) Python wykonuje jego instrukcje od góry do dołu. Niektóre instrukcje tworzą zmienne w przestrzeni nazw modułu, inne mogą wykonywać jakieś zadania w czasie importowania modułu. Przykładowo dwie instrukcje `print` w tym pliku wykonywane są w czasie importowania.

```
>>> import module2
rozpoczęto ładowanie...
zakończono ładowanie.
```

Po załadowaniu modułu jego zakres staje się przestrzenią nazw atrybutów w obiekcie modułów otrzymywanym z instrukcją `import`. Dostęp do tych atrybutów uzyskuje się, poprzedzając ich nazwy nazwą zawierającej je modułu.

```
>>> module2.sys
<module 'sys' (built-in)>

>>> module2.name
42

>>> module2.func
<function func at 0x026D3BB8>

>>> module2.klass
<class 'module2(klass)'>
```

W powyższym kodzie zmienne `sys`, `name`, `func` oraz `klass` zostały przypisane w czasie wykonywania instrukcji modułu, dlatego po zakończeniu importowania są atrybutami. Klasy omówione zostaną w szóstej części książki, jednak już teraz możemy zwrócić uwagę na atrybut `sys`. Instrukcja `import` naprawdę przypisuje obiekty modułów do zmiennych, a każdy rodzaj przypisania na najwyższym poziomie pliku generuje atrybut modułu.

Wewnętrznie przestrzenie nazw modułu przechowywane są jako obiekty słowników. Są to normalne obiekty słowników ze zwykłymi metodami słowników. Dostęp do słownika przestrzeni nazw modułu możemy uzyskać za pomocą atrybutu `__dict__` (należy pamiętać o opanowaniu go w Pythonie 3.0 wywołaniem funkcji `list` — jest to obiekt widoku).

```
>>> list(module2.__dict__.keys())
['name', '__builtins__', '__file__', '__package__', 'sys', 'klass', 'func',
 '__name__', '__doc__']
```

Zmienne przypisane w pliku modułu wewnętrznie stają się kluczami słownika, dlatego większość nazw z tej listy odzwierciedla przypisania na najwyższym poziomie pliku. Python dodaje jednak również pewne zmienne do przestrzeni modułów za nas — `__file__` podaje na przykład nazwę pliku, z którego załadowany został moduł, a `__name__` jego nazwę widoczną dla kodu importującego (bez rozszerzenia `.py` oraz ścieżki do katalogu).

Kwalifikowanie nazw atrybutów

Skoro już zapoznaliśmy się bliżej z modułami, powinniśmy bliżej przyjrzeć się pojęciu *kwalifikacji* zmiennych. W Pythonie możemy uzyskać dostęp do atrybutów dowolnego obiektu za pomocą składni kwalifikującej (z kropką, pobierającej atrybuty) w postaci `obiekt.atrybut`.

Kwalifikacja jest tak naprawdę wyrażeniem zwracającym wartość przypisaną do nazwy atrybutu powiązanego z obiektem. Przykładowo wyrażenie `module2.sys` z poprzedniego przykładu pobiera wartość przypisaną do zmiennej `sys` z modułu `module2`. W podobny sposób, jeśli mamy wbudowany obiekt listy `L`, wyrażenie `L.append` zwraca metodę `append` obiektu powiązanego z tą listą.

Co zatem kwalifikacja atrybutów robi z regułami zakresu omówionymi w rozdziale 17.? Tak naprawdę nic — są to niezależne koncepcje. Kiedy kwalifikację wykorzystamy w celu użycia dostępu do atrybutów, podajemy Pythonowi obiekt, z którego ma pobrać określone

zmienne. Reguła LEGB ma zastosowanie jedynie do „gołych” zmiennych bez kwalifikacji. Poniżej znajdują się odpowiednie reguły.

Proste zmienne

`X` oznacza wyszukanie zmiennej `X` w zakresie bieżącym (zgodnie z regułą LEGB).

Kwalifikacja

`X.Y` oznacza wyszukiwanie `X` w zakresach bieżących, a następnie odnalezienie atrybutu `Y` w obiekcie `X` (a nie w zakresach).

Ścieżki kwalifikacji

`X.Y.Z` oznacza wyszukiwanie zmiennej `Y` w obiekcie `X`, a następnie odnalezienie zmiennej `Z` w obiekcie `X.Y`.

Universalność

Kwalifikacja działa na wszystkich obiektach z atrybutami — modułach, klasach czy typach rozszerzeń języka C.

W szóstej części książki zobaczymy, że kwalifikacja oznacza nieco więcej w przypadku klas (jest również miejscem, w którym zachodzi *dziedziczenie*), jednak powyższe reguły odnoszą się do wszystkich zmiennych Pythona.

Importowanie a zakresy

Jak już wiemy, nie da się uzyskać dostępu do zmiennych zdefiniowanych w innym pliku modułu bez uprzedniego zaimportowania tego pliku. Oznacza to, że nigdy automatycznie nie zobaczymy zmiennych z innego pliku bez względu na strukturę importów lub wywołań funkcji w naszym programie. Znaczenie zmiennej jest zawsze uzależnione od lokalizacji przypisań w naszym kodzie źródłowym, a atrybutów obiektu zawsze żąda się w sposób jawnny.

Przykładowo rozważmy dwa poniższe proste moduły. Pierwszy (*moda.py*) definiuje zmienną `X` globalną dla kodu jego pliku wraz z funkcją modyfikującą tę zmienną globalną w pliku.

```
X = 88  
def f():  
    global X  
    X = 99  
  
# X — zmienna globalna dla tego pliku  
# Modyfikuje zmienną X tego pliku  
# Nie widzi zmiennych z innych modułów
```

Drugi moduł (*modb.py*) definiuje własną zmienną globalną `X`, a potem importuje i wywołuje funkcję z pierwszego modułu.

```
X = 11  
  
import moda  
moda.f()  
print(X, moda.X)  
  
# X — zmienna globalna dla tego pliku  
# Dostęp do zmiennych z moda  
# Ustawia moda.X, a nie X z tego pliku
```

Po wykonaniu funkcja `moda.f` modyfikuje zmienną `X` z `moda`, a nie zmienną `X` z `modb`. Globalnym zakresem `moda.f` jest zawsze plik zawierający tę funkcję, bez względu na to, z którego modułu funkcja ta zostanie wywołana.

```
% python modb.py  
11 99
```

Innymi słowy, operacje importowania nigdy nie przyznają kodowi z importowanego pliku widoczności „w góre” — importowany plik nie widzi zmiennych z pliku importującego. Bardziej formalnie:

- Funkcje nigdy nie widzą zmiennych innych funkcji, o ile ich nie są przez nie fizycznie zawierane.
- Kod modułu nigdy nie widzi zmiennych z innych modułów, o ile nie zostaną one zainportowane w sposób jawnym.

Takie zachowanie jest częścią koncepcji *zakresów leksykalnych*. W Pythonie zakresy otaczające fragment kodu są całkowicie uzależnione od fizycznej pozycji kodu w pliku. Wywołania funkcji czy operacje importowania modułów nigdy nie mają wpływu na zakresy.¹

Zagnieżdżanie przestrzeni nazw

W pewnym sensie, choć importowanie nie zagnieżdża przestrzeni nazw w góre, zagnieżdża je w dół. Wykorzystując ścieżki kwalifikacji atrybutów, możemy zejść na poziom dowolnie zagnieżdżonych modułów i uzyskać dostęp do ich atrybutów. Rozważmy na przykład kolejne trzy pliki. Plik *mod3.py* definiuje zmienną globalną oraz atrybut przez przypisanie.

```
X = 3
```

Plik *mod2.py* definiuje z kolei własną zmienną *X*, a następnie importuje *mod3* i wykorzystuje składnię kwalifikującą w celu uzyskania dostępu do atrybutu zimportowanego modułu.

```
X = 2
import mod3

print(X, end=' ')
# Globalna zmienna X pliku
print(mod3.X)
# Zmienna X z modułu mod3
```

Plik *mod1.py* również definiuje własną zmienną *X*, a następnie importuje *mod2* i pobiera atrybuty obu plików — *mod3.py* oraz *mod2.py*.

```
X = 1
import mod2

print(X, end=' ')
# Globalna zmienna X pliku
print(mod2.X, end=' ')
# Zmienna X z modułu mod2
print(mod2.mod3.X)
# Zmienna X z zagnieżdżonego modułu mod3
```

Tak naprawdę, kiedy *mod1* importuje *mod2*, ustawia dwupoziomowe zagnieżdżenie. Wykorzystując ścieżkę nazw *mod2.mod3.X*, może zejść aż do modułu *mod3* zagnieżdżonego w zainportowanym module *mod2*. Rezultat będzie taki, że *mod1* może widzieć zmienne *X* ze wszystkich trzech plików i tym samym ma dostęp do wszystkich trzech zakresów globalnych.

```
% python mod1.py
2 3
1 2 3
```

Odwrotna zależność nie jest jednak prawidłowa — *mod3* nie widzi zmiennych z *mod2*, a *mod2* nie widzi tych z *mod1*. Przykład ten łatwiej jest zrozumieć, kiedy nie będziemy myśleć w kategoriach przestrzeni nazw i zakresów, a zamiast tego skupimy się na obiektach. Wewnątrz modułu *mod1*, *mod2* jest po prostu zmienną odnoszącą się do obiektu z atrybutami, z których niektóre mogą się odnosić do innych obiektów z atrybutami (instrukcja *import* jest przypisaniem). W przypadku ścieżek takich, jak *mod2.mod3.X* Python po prostu oblicza je od lewej strony do prawej, po drodze pobierając atrybuty z obiektów.

¹ Niektóre języki programowania działają inaczej i udostępniają *zakresy dynamiczne*, gdzie zakresy naprawdę mogą być uzależnione od wywołań w czasie wykonywania. Sprawia to jednak, że kod staje się bardziej podchwytyliwy, ponieważ znaczenie zmiennej może się z czasem zmienić.

Warto zauważyć, że kod modułu `mod1` może zawierać instrukcję `import mod2`, a następnie wyrażenie `mod2.mod3.X`, jednak nie można użyć instrukcji `import mod2.mod3` — taka składnia wywołuje tak zwane importowanie pakietów (katalogów) opisane w kolejnym rozdziale. Importowanie pakietów tworzy również zagnieżdżenie przestrzeni nazw modułu, jednak takie instrukcje `import` służą do odzwierciedlania drzew katalogów, a nie prostych łańcuchów importu.

Przeładowywianie modułów

Jak widzieliśmy, kod modułu wykonywany jest domyślnie tylko raz na proces. By wymusić przeładowanie kodu modułu i ponowne jego wykonanie, trzeba zażądać tego od Pythona w sposób jawnym, wywołując jego wbudowaną funkcję `reload`. W niniejszym podrozdziale sprawdzimy, w jaki sposób można wykorzystać przeładowywianie modułów i tym samym sprawić, że nasze systemy będą bardziej dynamiczne. W skrócie:

- Operacje importowania (wykonywane za pośrednictwem instrukcji `import` oraz `from`) ładują i wykonują kod modułu jedynie za pierwszym razem, gdy moduł jest importowany w danym procesie.
- Późniejsze operacje importowania wykorzystują załadowany już obiekt modułu bez przeładowywania go lub ponownego wykonywania kodu pliku.
- Funkcja `reload` wymusza ponowne załadowanie i wykonanie kodu załadowanego wcześniej modułu. Przypisania w nowym kodzie pliku modyfikują istniejący obiekt modułu w miejscu.

Skąd to całe zamieszanie związane z przeładowywaniem modułów? Funkcja `reload` pozwala na modyfikację części programu bez konieczności zatrzymywania całego programu. Dzięki `reload` efekty zmian komponentów mogą być zauważone natychmiast. Przeładowywianie modułów nie pomaga w każdej sytuacji, ale tam, gdzie tak jest, bardzo skraca cykl programowania. Wyobraźmy sobie na przykład program bazodanowy, który na początku musi się połączyć z serwerem. Ponieważ modyfikacje programu czy jego dostosowanie do potrzeb użytkownika mogą być testowane natychmiast po przeładowaniu, wystarczy połączyć się tylko raz w czasie debugowania. Również długie działające serwery mogą się w ten sposób uaktualniać.

Ponieważ Python jest językiem (mniej więcej) interpretowanym, pozbywa się kroków komplikowania i łączenia, które musimy przejść, by zmusić do działania program napisany w języku C. Moduły są ładowane dynamicznie, kiedy importowane są przez działający program. Przeładowywianie oferuje dalszą przewagę w zakresie wydajności, pozwalając nam modyfikować części działających programów bez zatrzymywania ich. Warto zauważyć, że `reload` działa obecnie jedynie na modułach napisanych w Pythonie. Skompilowane moduły rozszerzeń napisane w językach takich, jak C można dynamicznie ładować w czasie wykonywania, jednak nie można ich przeładowywać.



Uwaga na temat wersji: W Pythonie 2.6 `reload` dostępne jest w postaci funkcji wbudowanej. W Pythonie 3.0 zostało przesunięte do modułu `imp` biblioteki standardowej — stąd w 3.0 występuje teraz jako `imp.reload`. Oznacza to po prostu, że w celu załadowania tego narzędzia (tylko w wersji 3.0) wymagane jest użycie dodatkowej instrukcji `import` lub `from`. Czytelnicy książki korzystający z wersji 2.6 mogą zignorować te polecenia w przykładach lub po prostu je wykonać — w wersji tej `reload` można także znaleźć w module `imp`, w celu ułatwienia przejścia na Pythona 3.0. Przeładowywianie działa tak samo bez względu na opakowanie funkcji w moduł.

Podstawy przeładowywania modułów

W przeciwieństwie do instrukcji `import` oraz `from`:

- `reload` jest w Pythonie funkcją, a nie instrukcją,
- do `reload` przekazuje się istniejący obiekt modułu, a nie zmienną,
- `reload` w Pythonie 3.0 znajduje się w module i funkcję tę trzeba specjalnie zimportować.

Ponieważ funkcja `reload` oczekuje przekazania obiektu, przed przeładowaniem modułu trzeba go najpierw zimportować; jeśli operacja importowania nie powiodła się z powodu błędu składniowego lub innego typu błędu, będziemy musieli ją powtórzyć przed przeładowaniem modułu. Co więcej, składnia instrukcji `import` i wywołań funkcji `reload` różni się. Przeładowywanie wymaga zastosowania nawiasów, natomiast importowanie już nie. Przeładowywanie wygląda jak poniższy kod.

```
import module                                # Początkowa operacja importowania  
...użycie składni moduł.atrybuty...  
...  
...  
from imp import reload                         # Uzyskanie samej funkcji reload (w 3.0)  
reload(module)                               # Uaktualnienie importowanego modułu  
...użycie składni moduł.atrybuty...
```

Typowy wzorzec użycia polega na zimportowaniu modułu, następnie modyfikacji jego kodu źródłowego w edytorze tekstu, a później przeładowaniu. Kiedy wywołamy funkcję `reload`, Python ponownie wczytuje kod źródłowy pliku modułu i raz jeszcze wykonuje jego instrukcję najwyższego poziomu. Chyba najważniejsze, o czym należy pamiętać w przypadku funkcji `reload`, jest to, że modyfikuje ona obiekt modułu *w miejscu* — nie usuwa ona obiektu modułu i nie tworzy go na nowo. Z tego powodu przeładowanie modułu wpływa na każdą referencję do obiektu modułu w dowolnym miejscu programu. Poniżej znajdują się szczegółowe informacje na ten temat.

- **Funkcja `reload` wykonuje nowy kod pliku modułu w bieżącej przestrzeni nazw modułu.** Ponowne wykonanie kodu pliku modułu nadpisuje jego istniejącą przestrzeń nazw, zamiast ją usuwać i odtwarzać.
- **Przypisania najwyższego poziomu w pliku następują zmienne nowymi wartościami.** Ponowne wykonanie instrukcji `def` następuje na przykład poprzednią wersję funkcji w przestrzeni nazw modułu, przypisując raz jeszcze nazwę funkcji.
- **Przeładowanie ma wpływ na każdy kod wykorzystujący instrukcję `import` do pobrania modułów.** Ponieważ kod wykorzystujący `import` używa składni kwalifikującej w celu pobrania atrybutów, po przeładowaniu w obiekcie modułu znajdzie on nowe wartości.
- **Przeładowanie ma wpływ jedynie na przeszły kod wykorzystujący instrukcję `from`.** Przeładowanie nie ma wpływu na kod, który wcześniej wykorzystał instrukcję `from` do pobrania atrybutów modułu — kod ten nadal będzie zawierał referencje do starych obiektów pobranych przed przeładowaniem.

Przykład przeładowywania z użyciem `reload`

W celu zademonstrowania omówionych zagadnień poniżej znajduje się bardziej konkretny przykład zastosowania funkcji `reload`. W poniższym przykładzie zmodyfikujemy i przeładowamy plik modułu bez zatrzymywania sesji interaktywnej Pythona. Przeładowywanie można

również wykorzystywać w wielu innych sytuacjach (zobacz ramkę „Znaczenie przeładowywania modułów”), jednak dla ułatwienia wszystko tutaj nieco uprościmy. Najpierw w wybranym edytorze tekstu należy utworzyć plik modułu o nazwie *changer.py* oraz następującej zawartości.

```
message = "Pierwsza wersja"  
def printer():  
    print(message)
```

Moduł ten tworzy i eksportuje dwie zmienne — jedną powiązaną z łańcuchem znaków, drugą z funkcją. Teraz należy uruchomić interpreter Pythona, zaimportować moduł i wywołać eksportowaną przez niego funkcję. Funkcja wyświetla wartość zmiennej globalnej *message*.

```
% python  
>>> import changer  
>>> changer.printer()  
Pierwsza wersja
```

Nie wyłączając interpretera, należy teraz w osobnym oknie dokonać edycji pliku modułu.

```
...modyfikacja modułu changer.py bez zatrzymywania Pythona...  
% vi changer.py
```

Teraz należy zmodyfikować zmienną globalną *message*, a także ciało funkcji *printer*.

```
message = "Po edycji"  
def printer():  
    print('przeładowany:', message)
```

Następnie należy powrócić do okna Pythona i przeładować moduł w celu pobrania nowego kodu. Warto w poniższym przykładzie zwrócić uwagę na to, że ponowne zaimportowanie modułu nie ma żadnego efektu — otrzymujemy oryginalny komunikat, mimo że plik się zmienił. By uzyskać dostęp do aktualizowanej wersji, musimy wywołać funkcję *reload*.

```
...powrót do interpretera Pythona i programu...
```

```
>>> import changer  
>>> changer.printer() # Brak efektu — wykorzystuje załadowany moduł  
Pierwsza wersja  
>>> from imp import reload  
>>> reload(changer) # Wymusza załadowanie i wykonanie nowego kodu  
<module 'changer' from 'changer.py'>  
>>> changer.printer() # Teraz wykonuje nową wersję  
przeładowany: Po edycji
```

Warto zauważyć, że funkcja *reload* tak naprawdę *zwrota* obiekt modułu — jej wynik jest zazwyczaj ignorowany, jednak ponieważ wynik wyrażenia jest wyświetlany w sesji interaktywnej, Python pokazuje domyślną reprezentację `<module 'nazwa' ...>`.

Podsumowanie rozdziału

Niniejszy rozdział skupił się na omówieniu podstaw tworzenia kodu modułów — instrukcji *import* oraz *from* i wywołania funkcji *reload*. Nauczyliśmy się, że instrukcja *from* dodaje po prostu dodatkowy etap kopiący zmienne z zaimportowanego pliku, a także że funkcja *reload* wymusza ponowne zaimportowanie pliku bez zatrzymywania i ponownego uruchamiania Pythona. Zapoznaliśmy się również z koncepcjami związanymi z przestrzeniami nazw i zoba-

Znaczenie przeładowywania modułów

Poza możliwością przeładowywania (i tym samym wykonywania) modułów w sesji interaktywnej funkcja `reload` przydaje się również w większych systemach, w szczególności kiedy koszt ponownego uruchomienia całej aplikacji jest duży. Dobrymi kandydatami do dynamicznego przeładowywania modułów są na przykład systemy, które muszą po uruchomieniu łączyć się z serwerami za pośrednictwem sieci.

Możliwość taka przydaje się również w pracy z graficznym interfejsem użytkownika (działanie zwrotne widgetu można zmienić w czasie, gdy GUI działa), a także kiedy Python wykorzystywany jest jako język osadzony w programie napisanym w C czy C++ (program go zawierający może zażądać przeładowania wykonywanego kodu w Pythonie bez konieczności zatrzymywania się). Więcej informacji na temat przeładowywania wywołań zwrotnych GUI oraz osadzonego kodu Pythona można znaleźć w książce *Programming Python*.

Przeładowywanie pozwala programom udostępniać bardzo dynamiczne interfejsy. Python jest na przykład często wykorzystywany jako język pozwalający na *dostosowanie większych systemów do własnych potrzeb*. Użytkownicy mogą dostosować produkty do swoich upodobań, pisząc fragmenty kodu w Pythonie na miejscu, bez konieczności ponownej komplikacji całego produktu (a nawet bez posiadania dostępu do jego kodu źródłowego). W takim świecie już sam kod napisany w Pythonie dodaje programowi dynamiki.

By jednak być jeszcze bardziej dynamicznymi, takie systemy mogą automatycznie i periodycznie przeładowywać kod napisany w Pythonie w czasie wykonywania. W ten sposób zmiany wprowadzone przez użytkownika są uwzględniane w czasie działania systemu — nie ma konieczności zatrzymywania go i ponownego uruchamiania za każdym razem, gdy modyfikujemy kod napisany w Pythonie. Nie wszystkie systemy wymagają tak dynamicznego podejścia, jednak dla tych, które to robią, przeładowywanie modułów jest łatwym w użyciu narzędziem umożliwiającym dynamiczne dostosowanie aplikacji do własnych potrzeb.

czyliśmy, co dzieje się w przypadku zagnieźdżenia operacji importowania. Wiemy także, w jaki sposób pliki stają się przestrzeniami nazw modułów, a także jakie są potencjalne pułapki związane z wykorzystywaniem instrukcji `from`.

Choć dowiedzieliśmy się już wystarczająco dużo, by potrafić obsługiwać moduły we własnych programach, w kolejnym rozdziale rozszerzymy naszą wiedzę dotyczącą modelu importowania, prezentując *importowanie pakietów* — sposób podania w instrukcji `import` fragmentu ścieżki do katalogu zawierającego pożądanym moduł. Jak zobaczymy, importowanie pakietów daje nam hierarchię przydatną w większych systemach i pozwala rozwiązać konflikty pomiędzy modułami o tych samych nazwach. Zanim jednak przejdziemy dalej, czas zająć się krótkim quizem podsumowującym omówione tutaj zagadnienia.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób tworzy się moduł?
2. W jaki sposób instrukcja `from` powiązana jest z instrukcją `import`?
3. W jaki sposób funkcja `reload` powiązana jest z operacją importowania?

4. Kiedy musimy użyć instrukcji `import` zamiast `from`?
5. Należy podać trzy potencjalne pułapki związane w instrukcją `from`.
6. Jaka jest prędkość lotu jaskółki bez obciążenia?

Sprawdź swoją wiedzę — odpowiedzi

1. By utworzyć moduł, należy utworzyć plik tekstowy zawierający instrukcje języka Python. Każdy plik kodu źródłowego automatycznie staje się modulem i nie istnieje składnia służąca do osobnej deklaracji modułu. Operacje importowania ładują pliki modułów do obiektów modułów znajdujących się w pamięci. Moduł można również utworzyć, pisząc kod w języku zewnętrznym, takim jak C lub Java, jednak tego typu moduły rozszerzeń pozostają poza zakresem niniejszej książki.
2. Instrukcja `from` importuje cały moduł, podobnie do instrukcji `import`, jednak dodatkowo kopiuje jeszcze jedną lub większą liczbę zmiennych z importowanego modułu do zakresu, w którym się pojawia. Pozwala to na bezpośrednie używanie zaimportowanych zmiennych (na przykład zmiennej *nazwa*) w miejscu konieczności uwzględniania w wyrażeniu nazwy modułu (w formie *moduł.nazwa*).
3. Domyślnie moduł jest importowany tylko raz na proces. Funkcja `reload` wymusza ponowne zaimportowanie modułu. Jest to wykorzystywane przede wszystkim w celu uzyskania nowszej wersji kodu źródłowego modułu w czasie programowania, a także w sytuacjach wymagających dynamicznego dostosowywania aplikacji do potrzeb użytkownika.
4. Instrukcji `import` używa się zamiast `from` jedynie wtedy, gdy musimy uzyskać dostęp do zmiennych o tej samej nazwie występujących w dwóch różnych modułach. Ponieważ musimy w takiej sytuacji dodatkowo podać nazwę modułu zawierającego zmienną, nazwy te będą unikalne.
5. Instrukcja `from` może zaciemnić znaczenie zmiennej (to, w którym module została zdefiniowana), może powodować problemy w czasie wywoływania funkcji `reload` (zmienna może odnosić się do poprzedniej wersji obiektów), a także może niszczyć przestrzeń nazw (po cichu nadpisując zmienne wykorzystywane w zakresie bieżącym). Forma `from *` jest pod wieloma względami gorsza — może poważnie zniszczyć przestrzeń nazw i zaciemnić znaczenie zmiennych. Najlepiej jest używać jej oszczędnie.
6. Jakiej jaskółki? Afrykańskiej czy europejskiej?

Pakiety modułów

Dotychczas kiedy importowaliśmy moduły — ładowaliśmy pliki. To typowy model użycia modułów i technika, której w początkach naszej kariery programisty Pythona będziemy używać najczęściej. Importowanie modułów to jednak coś więcej, niż dotychczas sugerowałem.

Poza nazwą modułu w operacji importowania można również wymienić ścieżkę do katalogu. Katalog kodu Pythona nazywa się *pakietem*, dlatego tego typu operacje importowania znane są jako *importowanie pakietów* (ang. *package import*). W rezultacie importowanie pakietów zamienia katalog z naszego komputera na kolejną przestrzeń nazw Pythona, z atrybutami odpowiadającymi podkatalogom oraz plikom modułów znajdujących się w tym katalogu.

Jest to opcja nieco bardziej zaawansowana, ale udostępniana przez nią hierarchia okazuje się przydatna do organizowania plików w większe systemy i zazwyczaj upraszcza ustawienia ścieżki wyszukiwania modułów. Jak zobaczymy, importowanie pakietów jest czasami wymagane do rozwiązymania problemów z operacjami importowania powstających, kiedy na jednym komputerze zainstalowanych jest kilka plików programów o tej samej nazwie.

W niniejszym rozdziale omówimy również wprowadzony w najnowszych wersjach Pythona mechanizm *importów względnych*, który jest ściśle powiązany z pakietami i modułami. Jak zobaczymy, ta nowość wzbogaca logikę obsługi ścieżek wyszukiwania i rozszerza możliwości instrukcji `from` w zakresie importowania nazw z pakietów.

Podstawy importowania pakietów

Jak zatem działa importowanie pakietów? W miejscu, w którym w instrukcji `import` normalnie podalibyśmy zwykłą nazwę pliku, zamiast tego podajemy ścieżkę nazw rozdzielonych kropkami.

```
import dir1.dir2.mod
```

Tak samo wygląda to w przypadku instrukcji `from`.

```
from dir1.dir2.mod import x
```

Ścieżka z kropkami w tych instrukcjach ma odpowiadać ścieżce przez hierarchię katalogów naszego komputera, prowadzącej do pliku `mod.py` (lub podobnego — rozszerzenia mogą być różne). Powyższe instrukcje wskazują zatem, że na naszym komputerze istnieje katalog `dir1`, a w nim podkatalog `dir2` zawierający plik modułu `mod.py` (lub podobny).

Co więcej, takie operacje importowania sugerują, że katalog *dir1* znajduje się w jakimś katalogu-pojemniku *dir0*, dostępnym dla ścieżki wyszukiwania modułów Pythona. Innymi słowy, powyższe instrukcje sugerują, że na komputerze istnieje struktura przypominająca poniższą (pokazaną z separatorami ukośników lewych systemu DOS).

```
dir0\dir1\dir2\mod.py # Lub mod.pyc, mod.so i tak dalej
```

Katalog-pojemnik *dir0* musi być dodany do ścieżki wyszukiwania modułów (o ile nie jest katalogiem głównym pliku najwyższego poziomu) — dokładnie tak samo, jakby *dir1* był plikiem modułu.

Ogólnie: pierwszy od lewej element ścieżki importu jest nazwą względną w ramach ścieżki wyszukiwania `sys.path`, którą mieliśmy okazję poznać w rozdziale 21. Od tego miejsca do końca ścieżki instrukcje `import` z naszego skryptu w jawnym sposobie określają ścieżki katalogów prowadzących do modułów.

Pakiety a ustawienia ścieżki wyszukiwania

Jeśli korzystamy z tej opcji, należy pamiętać, że ścieżki katalogów w instrukcjach `import` mogą być tylko zmiennymi rozdzielonymi kropkami. Nie można w tych instrukcjach użyć żadnej składni ścieżek specyficznej dla określonej platformy, takiej jak `C:\dir1, Moje dokumenty.dir2` czy `../dir1` — takie ścieżki nie będą działały. Zamiast tego składni specyficznej dla platformy należy użyć w ustawieniach ścieżki wyszukiwania modułów i określić w ten sposób nazwę katalogu-pojemnika.

W poprzednim przykładzie katalog *dir0* — nazwa katalogu dodawana do ścieżki wyszukiwania modułów — może być dowolnie dłużna, specyficzną dla platformy ścieżką do katalogu, prowadzącą do katalogu *dir1*. Zamiast używać niepoprawnej instrukcji, takiej jak poniższa:

```
import C:\mycode\dir1\dir2\mod # Błąd — niepoprawna składnia
```

należy dodać katalog `C:\mycode` do zmiennej środowiskowej `PYTHONPATH` lub pliku `.pth` (zakładając, że nie jest to katalog główny programu, ponieważ w tym przypadku krok ten będzie zbędny), a następnie wpisać w kodzie programu jedynie:

```
import dir1.dir2.mod
```

W rezultacie wpisy ze ścieżki wyszukiwania modułów zawierają przedrostki katalogu specyficzne dla platformy, prowadzące do nazw znajdujących się po lewej stronie instrukcji `import`. Instrukcje `import` udostępniają ścieżki do katalogów w formie neutralnej dla platformy.¹

Pliki pakietów `_init_.py`

Jeśli zdecydujemy się na wykorzystanie importowania pakietów, istnieje jedno ograniczenie, którego należy przestrzegać. Każdy katalog wymieniony w ścieżce instrukcji importu pakietu musi zawierać plik o nazwie `_init_.py` — inaczej importowanie się nie powiedzie. W przykładzie wykorzystanym wyżej katalogi *dir1* oraz *dir2* muszą zawierać plik o nazwie `_init_.py`.

¹ Składnia z kropkami została wybrana ze względu na neutralność, ale także dlatego, że ścieżki z instrukcją `import` stają się prawdziwymi ścieżkami obiektów zagnieżdżonych. Składnia ta oznacza również, że jeśli w instrukcjach `import` zapomnijmy o pominięciu rozszerzenia `.py`, otrzymamy dziwne błędy. Python zakłada na przykład, że instrukcja `import mod` jest operacją importowania ścieżki do katalogu, i najpierw załaduje `mod.py`, później spróbuje załadować `mod\py.py`, a na końcu zwróci dosyć mylący komunikat o błędzie.

Katalog-pojemnik *dir0* nie musi zawierać tego pliku, ponieważ nie jest on wymieniony w samej instrukcji `import`. Z formalnego punktu widzenia, jeśli struktura katalogów wygląda następująco:

```
dir0\dir1\dir2\mod.py
```

a instrukcja `import` ma następującą postać:

```
import dir1.dir2.mod
```

to zastosowanie mają poniższe reguły:

- katalogi *dir1* oraz *dir2* muszą zawierać plik `__init__.py`,
- katalog *dir0* będący pojemnikiem nie musi zawierać pliku `__init__.py`; jeśli plik ten będzie się w nim znajdował, zostanie zignorowany,
- katalog *dir0*, a nie *dir0\dir1*, musi zostać wymieniony w ścieżce wyszukiwania modułów (czyli musi być albo katalogiem głównym, albo znajdująć się w zmiennej `PYTHONPATH` lub pliku `.pth`).

Rezultat jest taki, że struktura katalogów tego przykładu byłaby następująca, z indentacją oznaczającą zagnieźdzenie katalogów.

```
dir0\  
    dir1\  
        __init__.py  
    dir2\  
        __init__.py  
        mod.py
```

Pojemnik ze ścieżki wyszukiwania modułów

Pliki `__init__.py` mogą zawierać kod Pythona, podobnie do normalnych plików modułów. Są jednak obecne jako deklaracje dla Pythona i mogą być całkowicie puste. Jako deklaracje pliki te mają zapobiec ukrywaniu prawdziwych modułów leżących niżej na ścieżce wyszukiwania modułów przez katalogi o popularnych nazwach. Dzięki takiemu zabezpieczeniu Python może wybrać katalog, który nie ma nic wspólnego z naszym kodem, tylko dlatego że znajduje się on we wcześniejszym katalogu na ścieżce wyszukiwania.

Pliki `__init__.py` służą jako punkty zaczepienia dla działań odbywających się w czasie inicjalizowania pakietów, generują przestrzeń nazw dla katalogów i implementują zachowanie instrukcji `from *` (na przykład `from ... import *`), kiedy wykorzystuje się je w połączeniu z importowaniem pakietów.

Inicjalizacja pakietów

Za pierwszym razem, gdy Python importuje coś za pomocą katalogu, automatycznie wykonuje cały kod z pliku `__init__.py` tego katalogu. Z tego powodu pliki te są naturalnym miejscem do wstawienia kodu inicjalizującego stan wymagany przez pliki w pakiecie. Pakiet może na przykład wykorzystać plik inicjalizujący do utworzenia wymaganych plików z danymi czy otwarcia połączenia z bazą danych. Zazwyczaj pliki `__init__.py` nie mają być przydatne przy bezpośrednim wykonywaniu. Są one wykonywane automatycznie przy pierwszym dostępie do pakietu.

Inicjalizacja przestrzeni nazw modułu

W modelu importowania pakietów ścieżki katalogów ze skryptu stają się po zainportowaniu prawdziwymi ścieżkami zagnieżdżonych obiektów. W poprzednim przykładzie po zainportowaniu wyrażenie `dir1.dir2.mod` działa i zwraca obiekt modułu, którego przestrzeń nazw zawiera wszystkie zmienne przypisane przez plik `__init__.py` katalogu *dir2*.

Takie pliki udostępniają przestrzeń nazw dla obiektów modułów tworzonych dla katalogów, które nie mają prawdziwych powiązanych plików modułów.

Zachowanie podobne do instrukcji `from *`

Jako zaawansowaną opcję możemy wykorzystać listy `_all_` z plików `__init__.py` do zdefiniowania, co jest eksportowane, kiedy katalog importowany jest za pomocą instrukcji `from *`. W pliku `__init__.py` lista `_all_` ma być listą nazw podmodułów, które powinny zostać zaimportowane, kiedy instrukcji `from *` użyjemy na nazwie pakietu (katalogu). Jeśli lista `_all_` nie zostanie ustawiona, instrukcja `from *` nie załaduje podmodułów zagnieżdżonych w katalogu automatycznie. Zamiast tego załaduje tylko zmienne zdefiniowane przez przypisania z pliku `__init__.py` katalogu, w tym wszystkie podmoduły w jawnym sposób zaimportowane przez kod tego pliku. Instrukcja `from submodule import X` w pliku `__init__.py` katalogu sprawia, że zmienna `X` z podmodułu `submodule` będzie dostępna w przestrzeni nazw tego katalogu (z `_all_` spotkamy się w rozdziale 24.).

Możemy te pliki również po prostu pozostawić puste, jeśli ich rola wykracza poza nasze potrzeby. By jednak importowanie katalogów działało, muszą one przynajmniej istnieć.



Nie należy mylić plików `__init__.py` w pakietach z metodami konstruktorów `__init__` w klasach, które poznamy w następnej części książki. Pierwsze z nich są modułami ładowanymi przez interpreter w ramach importu pakietu, drugie są metodami wywoływanymi w celu utworzenia instancji. Obie te koncepcje realizują funkcje inicjalizacji, ale znacznie różnią się od siebie.

Przykład importowania pakietu

Utwórzmy teraz prawdziwy kod przykładu, o jakim mówiliśmy, w celu pokazania, w jaki sposób działa inicjalizacja plików oraz ścieżek. Poniższe trzy pliki zapisane są w katalogu `dir1` oraz jego podkatalogu `dir2`.

```
# Plik dir1\__init__.py
print 'dir1 init'
x = 1

# Plik dir1\dir2\__init__.py
print 'dir2 init'
y = 2

# Plik dir1\dir2\mod.py
print 'w mod.py'
z = 3
```

Katalog `dir1` będzie tutaj albo podkatalogiem tego, w którym już pracujemy (czyli katalogu głównego), albo podkatalogiem katalogu wymienionego w ścieżce wyszukiwania modułów (z technicznego punktu widzenia: w `sys.path`). W obu sytuacjach pojemnik katalogu `dir1` nie musi zawierać pliku `__init__.py`.

Instrukcje `import` wykonują pliki inicjalizacyjne każdego katalogu przy pierwszym przeходzeniu tego katalogu, w miarę jak Python schodzi w dół ścieżki. Instrukcje `print` są wstawione do kodu, by prześledzić wykonywanie. Tak jak w przypadku plików modułów, zaimportowany już katalog może zostać przekazany do funkcji `reload` w celu wymuszenia ponownego wykonania tego elementu. Jak widać poniżej, funkcja `reload` przyjmuje nazwę ścieżki z kropkami, by móc przeładować zagnieżdżone katalogi oraz pliki.

```
% python
>>> import dir1.dir2.mod                                # Pierwszy import wykonuje pliki inicjalizacyjne
dir1 init
dir2 init
w mod.py
>>>
>>> import dir1.dir2.mod                            # Kolejne importy tego nie robią
>>>
>>> from imp import reload                         # Wymagane w 3.0
>>> reload(dir1)
dir1 init
<module 'dir1' from 'dir1\__init__.pyc'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
```

Po zimportowaniu ścieżka z instrukcji `import` staje się ścieżką zagnieżdzonego obiektu w skrypcie. W kodzie tym `mod` jest obiektem zagnieżdżonym w obiekcie `dir2`, który jest z kolei zagnieżdżony w obiekcie `dir1`.

```
>>> dir1
<module 'dir1' from 'dir1\__init__.pyc'>
>>> dir1.dir2
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from 'dir1\dir2\mod.pyc'>
```

Tak naprawdę każda nazwa katalogu ze ścieżki staje się zmienną przypisaną do obiektu modułu, którego przestrzeń nazw inicjalizowana jest przez wszystkie przypisania z pliku `__init__.py` tego katalogu. Zmienna `dir1.x` odnosi się do zmiennej `x` przypisanej w pliku `dir1__init__.py`, tak samo jak zmienna `mod.z` odnosi się do zmiennej `z` przypisanej w pliku `mod.py`.

```
>>> dir1.x
1
>>> dir1.dir2.y
2
>>> dir1.dir2.mod.z
3
```

Instrukcja `from` a instrukcja `import` w importowaniu pakietów

Instrukcje `import` mogą być nieco niewygodne w połączeniu z pakietami, ponieważ często musimy w programie wpisywać te same ścieżki. W przykładzie wyżej za każdym razem, gdy chcemy dotrzeć do zmiennej `z`, musimy ponownie wpisać i wykonać pełną ścieżkę od katalogu `dir1`. Jeśli próbujemy uzyskać bezpośredni dostęp do katalogu `dir2`, otrzymamy błąd.

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```

Często w przypadku pakietów wygodniejsze jest zatem skorzystanie z instrukcji `from`, co pozwala uniknąć ponownego wpisywania całych ścieżek w każdym dostępie do obiektów. Co jednak ważniejsze, jeśli kiedykolwiek zmienimy strukturę drzewa katalogów, instrukcja `from` wymaga aktualnienia tylko jednej ścieżki w kodzie, podczas gdy `import` może wiązać się w większą liczbą zmian. Omówione w kolejnym rozdziale rozszerzenie `import as` może również być pomocne, gdyż podaje krótszy synonim pełnej ścieżki.

```
% python
>>> from dir1.dir2 import mod          # Ścieżka podana jedynie tutaj
dir1 init
dir2 init
w mod.py
>>> mod.z                            # Nie powtarzamy ścieżki
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod      # Użycie krótszej nazwy (w kolejnym, 24. rozdziale)
>>> mod.z
3
```

Do czego służy importowanie pakietów?

Osoby zaczynające swoją przygodę z Pythonem powinny przed przejściem do pakietów opanować proste moduły, ponieważ pakiety są już opcją bardziej zaawansowaną. Pełnią jednak użyteczne role, w szczególności w większych programach. Sprawiają, że operacje importowania zawierają więcej informacji, mogą służyć jako narzędzia organizacyjne, upraszczają ścieżkę wyszukiwania modułów i mogą rozwiązać różne niejasności.

Przede wszystkim jednak, ponieważ importowanie pakietów podaje nam pewne informacje o katalogach w plikach programów, ułatwia nam lokalizację plików i służy także jako narzędzie organizacyjne. Bez ścieżek pakietów często musielibyśmy się odwoływać do ścieżki wyszukiwania pakietów, która pomagałaby nam odnaleźć pliki. Co więcej, jeśli zorganizujemy swoje pliki w podkatalogi zgodne z pewnymi obszarami funkcjonalnymi, importowanie pakietów sprawia, że bardziej oczywiste staje się, jaką rolę pełni moduł, dzięki czemu kod jest bardziej czytelny. Normalne zimportowanie pliku w katalogu znajdującym się gdzieś na ścieżce wyszukiwania modułów, takie jak poniższe:

```
import utilities
```

oferuje nam o wiele mniej informacji niż operacja importowania uwzględniająca ścieżkę:

```
import database.client.utilities
```

Importowanie pakietów może również znacznie uprościć nasze ustawienia zmiennej PYTHONPATH oraz plików *.pth*. Tak naprawdę jeśli we wszystkich operacjach importowania pomiędzy katalogami zastosujemy importowanie pakietów względne w stosunku do wspólnego katalogu głównego, w którym przechowywany jest cały nasz kod napisany w Pythonie, w ścieżce wyszukiwania potrzebny nam będzie tylko jeden wpis — tego wspólnego katalogu. Wreszcie importowanie pakietów pomaga rozwiązać często spotykane problemy, wskazując jednoznacznie, które pliki chcemy zimportować. W kolejnym podrozdziale zajmiemy się bardziej szczegółowo tą właśnie rolą.

Historia trzech systemów

Jedyna sytuacja, w której importowanie pakietów jest właściwie wymagane do rozwiązania niejasności, pojawia się, kiedy na jednym komputerze zainstalowana jest większa liczba programów z plikami noszącymi te same nazwy. Jest to problem instalacyjny, jednak w praktyce może stać się dotkliwy. By go zilustrować, zajmijmy się pewnym hipotetycznym scenariuszem wydarzeń.

Załóżmy, że programista tworzy w Pythonie program zawierający plik o nazwie *utilities.py* (ze wspólnym kodem narzędzi), a także plik najwyższego poziomu *main.py* wykorzystywany przez użytkowników do uruchomienia programu. W całym programie pliki wykorzystują kod `import utilities` w celu załadowania wspólnego kodu narzędzi i skorzystania z niego. Kiedy program jest dostarczany klientom, stanowi jedno archiwum *.tar* czy *.zip* zawierające wszystkie pliki programu, a po instalacji rozpakowuje wszystkie pliki do jednego katalogu na komputerze docelowym o nazwie *system1*.

```
system1\  
    utilities.py          # Wspólne funkcje i klasy narzędzi  
    main.py               # Uruchamia program  
    other.py              # Importuje utilities w celu załadowania narzędzi
```

Załóżmy teraz, że drugi programista tworzy inny program z plikami o nazwach *utilities.py* oraz *main.py* i ponownie wykorzystuje instrukcję `import utilities` w całym programie w celu załadowania pliku ze wspólnym kodem. Kiedy drugi system zostanie pobrany i zainstalowany na tym samym komputerze co pierwszy, jego pliki zostaną rozpakowane do nowego katalogu o nazwie *system2* gdzieś na komputerze klienta, tak by nie nadpisały plików z pierwszego systemu o tych samych nazwach.

```
system2\  
    utilities.py          # Wspólne narzędzia  
    main.py               # Uruchamia program  
    other.py              # Importuje narzędzia
```

Jak na razie nie ma żadnych problemów — oba systemy mogą współistnieć i mogą być wykonywane na tym samym komputerze. Tak naprawdę nie musimy nawet konfigurować ścieżki wyszukiwania modułów, by skorzystać z obu programów — ponieważ Python zawsze najpierw przeszukuje katalog główny (czyli katalog zawierający plik najwyższego poziomu), operacje importowania w plikach obu systemów automatycznie zobaczą wszystkie pliki w katalogu danego systemu. Jeśli na przykład klikniemy plik *system1\main.py*, wszystkie operacje importowania najpierw będą przeszukiwały katalog *system1*. W podobny sposób po uruchomieniu pliku *system2\main.py* jako pierwszy przeszukany zostanie katalog *system2*. Należy pamiętać, że ustawienia ścieżki wyszukiwania modułów są potrzebne tylko wtedy, gdy importujemy pliki pomiędzy katalogami.

Załóżmy jednak, że po zainstalowaniu tych dwóch programów na komputerze decydujemy się użyć jakieś części kodu z każdego z plików *utilities.py* we własnym programie. W końcu jest to wspólny kod narzędzi, a kod napisany w Pythonie z natury służy do ponownego użycia. W takim przypadku musimy móc napisać poniższe instrukcje w kodzie pliku utworzonego w trzecim katalogu, tak by załadować jeden z dwóch plików z narzędziami.

```
import utilities  
utilities.func('mielonka')
```

I teraz zaczynamy dostrzegać problem. By taki kod działał, musimy ustawić ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalogi zawierające pliki *utilities.py*. Który katalog należy jednak umieścić jako pierwszy — *system1* czy *system2*?

Problemem jest *liniowa* natura ścieżki wyszukiwania. Zawsze jest ona przeglądana od lewej do prawej strony, więc bez względu na to, ile byśmy nie kombinowali, zawsze otrzymamy plik *utilities.py* z katalogu wymienionego jako pierwszy (bardziej na lewo) w ścieżce wyszukiwania. W takiej postaci nigdy nie będziemy w stanie zimportować tego pliku z innego katalogu.

Możemy spróbować zmodyfikować `sys.path` w skrypcie przed każdą operacją importowania, ale to dodatkowy wysiłek, na dodatek bardzo podatny na błędy. Domyślne rozwiązanie sprawia, że jesteśmy w kropce.

Problem ten może rozwiązać właśnie importowanie pakietów. Zamiast instalować programy jako płaskie listy plików w osobnych katalogach, możemy je spakować i zainstalować w *pod-katalogach* znajdujących się we wspólnym katalogu głównym. Możemy na przykład zorganizować cały kod tego przykładu w postaci hierarchii wyglądającej następująco:

```
root\
    system1\
        __init__.py
        utilities.py
        main.py
        other.py
    system2\
        __init__.py
        utilities.py
        main.py
        other.py
    system3\          # Tutaj lub w dowolnym innym miejscu
        __init__.py      # Tutaj nasz nowy kod
        myfile.py
```

Do ścieżki wyszukiwania dodajemy wtedy tylko wspólny katalog główny. Jeśli wszystkie operacje importowania w naszym kodzie są wykonywane względem wspólnego katalogu głównego, możemy zaimportować plik narzędzi *z dowolnego* programu za pomocą importowania pakietu — nazwa katalogu zawierającego plik sprawia, że ścieżka (i tym samym referencja do modułu) staje się unikalna. Tak naprawdę możemy nawet zaimportować *oba* pliki narzędzi w tym samym module, dopóki wykorzystujemy instrukcję `import` i powtarzamy pełną ścieżkę za każdym razem, gdy odnosimy się do modułów narzędzi.

```
import system1.utilities
import system2.utilities
system1.utilities.function('mielonka')
system2.utilities.function('jajka')
```

Nazwa modułu zawierającego plik sprawia, że referencja do modułu staje się unikalna.

Warto zauważyć, że musimy użyć instrukcji `import` zamiast `from` w przypadku importowania pakietów tylko wtedy, gdy musimy uzyskać dostęp do tego samego atrybutu z dwóch lub większej liczby ścieżek. Gdyby nazwa wywoływanej funkcji była w każdej ścieżce inna, można by było użyć instrukcji `from`, co pozwalałoby uniknąć powtarzania pełnej ścieżki za każdym wywołaniem którejś z funkcji, tak jak opisano to wcześniej.

Warto również zwrócić uwagę na to, że w pokazanej wcześniej hierarchii zainstalowanych plików pliki `__init__.py` zostały dodane do katalogów `system1` oraz `system2`, tak by importowanie pakietów działało, jednak nie znalazły się w katalogu głównym o nazwie `root`. Pliki te wymagane są jedynie w katalogach wymienionych w instrukcjach `import`. Jak pamiętamy, są one wykonywane automatycznie przez proces Pythona przy pierwszym importowaniu za pośrednictwem katalogu pakietów.

Z technicznego punktu widzenia w tym przypadku katalog `system3` nie musi się znajdować pod katalogiem `root` — dotyczy to jedynie pakietów kodu, z których będziemy importować. Ponieważ jednak nigdy nie wiemy, kiedy nasze własne moduły będą mogły się przydać innym programom, możemy równie dobrze od razu umieścić je we wspólnym katalogu głównym w celu uniknięcia problemów z konfliktami między takimi samymi nazwami w przyszłości.

Wreszcie warto zauważyć, że instrukcje importowania z obu oryginalnych systemów działają bez zmian. Ponieważ najpierw przeszukiwane są ich katalogi *głonne*, dodanie wspólnego katalogu do ścieżki wyszukiwania nie ma znaczenia dla kodu z katalogów *system1* oraz *system2*. Nadal można w nich zastosować kod `import utilities` i oczekwać, że odnajdą w ten sposób własne pliki. Co więcej, jeśli będziemy uważnie rozpakowywać wszystkie programy napisane w Pythonie w jednym katalogu, tak jak zaprezentowano to powyżej, konfiguracja ścieżki staje się prosta — wystarczy tylko raz dodać wspólny katalog główny.

Względne importowanie pakietów

Dotychczas importowanie pakietów omawialiśmy w zakresie importowania modułów pakietu *spoza* pakietu. W ramach samego pakietu również można importować moduły tego samego pakietu i do tego stosuje się tę samą składnię, co w przypadku importów z zewnątrz, ale można też stosować specjalne, uproszczone reguły importowania wewnątrz pakietu. Zamiast określać całą ścieżkę do modułu w pakiecie, można zastosować ścieżkę względową wewnątrz pakietu.

Mechanizm importów względnych działa w różny sposób w różnych wersjach Pythona. W 2.6 katalogi pakietu są przeszukiwane w pierwszej kolejności, natomiast w 3.0 istnieje konieczność użycia specjalnej składni wymuszającej wyszukiwanie względne. Ta zmiana wprowadzona w 3.0 wpływa na polepszenie czytelności kodu, ponieważ niektóre ścieżki importów stają się bardziej oczywiste. Jeśli ktoś swoją przygodę z Pythonem rozpoczyna od 3.0, powinien skupić się na nowej składni importów. Programiści znający starsze wersje Pythona z pewnością zainteresują się również różnicami w działaniu mechanizmu względnych importów w różnych wersjach języka.

Zmiany w Pythonie 3.0

Mechanizm importowania z pakietów uległ drobnym zmianom w Pythonie 3.0. Ta zmiana dotyczy wyłącznie importów z modułów należących do tego samego pakietu, które omawialiśmy w niniejszym rozdziale. Importy z innych plików działają tak samo jak dotychczas. W przypadku importów wewnątrz pakietu Python 3.0 wprowadza następujące zmiany:

- Modyfikuje mechanizm importowania w taki sposób, aby bieżący katalog pakietu był pomijany, przez co ścieżki są przeszukiwane w pozostałych elementach ścieżki wyszukiwania. Tego typu import nazywa się *bezwzględny* (ang. *absolute*).
- Rozszerza składnię instrukcji importu, pozwalając na określenie ścieżki wewnątrz bieżącego pakietu. Tego typu import nazywa się *względny* (ang. *relative*).

Te dwie zmiany obowiązują od Pythona 3.0. Nowa składnia importów względnych jest również dostępna w Pythonie 2.6, ale zmiana mechanizmu domyślnej ścieżki wyszukiwania musi być włączona jako opcja. Domyślne włączenie tej opcji odłożono na później, ponieważ ta zmiana prysuje zgodność z poprzednimi wersjami Pythona.

Skutkiem tej zmiany jest konieczność użycia w Pythonie 3.0 (i opcjonalnie w 2.6) specjalnej składni względnego importu w celu zimportowania modułów znajdujących się w tym samym pakiecie albo zastosowania pełnej ścieżki bezwzględnej. Bez użycia tej składni bieżący pakiet nie będzie przeszukiwany.

Podstawy importów względnych

W Pythonie 3.0 i 2.6 instrukcje `from` mogą wykorzystywać wiodące kropki, które sygnalizują, że wymagane są moduły zlokalizowane w tym samym pakiecie (nazywamy to *importem względnym*), a nie w dowolnym miejscu ścieżki wyszukiwania (co nazywamy *importem bezwzględnym*). A dokładniej:

- W Pythonie 3.0 i 2.6 można użyć wiodącej kropki w instrukcjach `from`, aby wymusić import względny w ramach pakietu: tego typu importy wyszukują moduły tylko w bieżącym pakiecie i nie znajdą modułów o tych samych nazwach zapisanych w innych miejscach ścieżki wyszukiwania (`sys.path`). W efekcie moduły pakietu mogą przeciązać moduły zewnętrzne.
- W Pythonie 2.6 importy bez wiodącej kropki powodują wyszukiwanie w trybie względnym, a następnie bezwzględnym, to znaczy wyszukiwanie odbywa się w pierwszej kolejności w bieżącym pakiecie. W Pythonie 3.0 natomiast importy w ramach pakietu są domyślnie bezwzględne (jeśli nie zastosowano wiodącej kropki): import pomija moduły pakietu i wyszukuje wyłącznie w ścieżce `sys.path`.

Na przykład w Pythonie 3.0 i 2.6 można zastosować taką instrukcję:

```
from . import spam # Import względny w stosunku do pakietu
```

Ta instrukcja spowoduje zimportowanie modułu o nazwie `spam`, położonego w tym samym katalogu, co moduł, w którym występuje ta instrukcja. Podobnie instrukcja:

```
from .spam import name
```

oznacza „z modułu `spam` znajdującego się w tym samym katalogu zimportuj zmienną o nazwie `name`”.

Zachowanie instrukcji importu *bez użycia* wiodących kropek jest różne w różnych wersjach Pythona. W 2.6 import tego typu powoduje wyszukiwanie względne (czyli w katalogu bieżącego pakietu), a jeśli się nie powiedzie, bezwzględne w ramach ścieżki wyszukiwania. To domyślne zachowanie można zmienić, wywołując następującą instrukcję:

```
from __future__ import absolute_import # Wymagane do wersji 2.7?
```

W przypadku zastosowania takiego importu w wersji 2.6 uzyskamy zgodność z 3.0 w zakresie domyślnego wyszukiwania w importach bezwzględnych.

W 3.0 import bez wiodących kropek powoduje, że Python w wyszukiwaniu modułów pomija bieżący katalog i wyszukuje nazwy wyłącznie w ścieżce `sys.path`. Na przykład poniższa instrukcja w 3.0 spowoduje zimportowanie modułu `string` ze ścieżki `sys.path`, nie załaduje modułu `string` zdefiniowanego w bieżącym pakiecie:

```
import string # Pominiecie wersji zdefiniowanej w bieżącym pakiecie
```

W 2.6, jeśli nie została zastosowana instrukcja `from __future__`, Python w pierwszej kolejności podejmie próbę zimportowania modułu w bieżącym pakiecie. Aby uzyskać ten efekt w 3.0 oraz w 2.6 w trybie zgodności z 3.0, należy jawnie zastosować składnię importu względnego:

```
from . import string # Wyszukuje w bieżącym pakiecie
```

Taka składnia importu działa w Pythonie 2.6 i 3.0. Różnica między wersjami Pythona polega jedynie na tym, że w 3.0 zastosowanie tej składni jest konieczne w celu załadowania modułu z bieżącego katalogu, jeśli posiada taką samą nazwę, jak moduł w ścieżce wyszukiwania.

Należy pamiętać, że wiodące kropki mogą być zastosowane jedynie w instrukcji `from`, nie w instrukcji `import`. W Pythonie 3.0 instrukcja `import` moduł zawsze działa w trybie bezwzględnym, czyli pomija bieżący katalog pakietu. W 2.6 ta forma importu najpierw wywołuje import względny, a jeśli zakończy się niepowodzeniem, następuje wyszukiwanie w ścieżce, ale od wersji 2.7 zachowanie stanie się zgodne z 3.0. Instrukcje `from` bez kropek wiodących działają tak samo, jak instrukcje `import`.

Składania importów względnych obsługuje kilka opcji. Oto kilka przykładów tego typu wywołań w kontekście pakietu `mypkg`, z opisem działania w komentarzach do kodu:

```
from .string import name1, name2 # Importuje nazwy z mypkg.string
from . import string             # Importuje mypkg.string
from .. import string            # Importuje string z tego samego poziomu, na którym znajduje się mypkg
```

Aby lepiej zrozumieć mechanizm działania tych importów, należy poznać przyczynę ich wprowadzenia.

Do czego służą importy względne?

Ten mechanizm został wprowadzony w celu rozstrzygnięcia wieloznaczności, jakie powstają w sytuacji, gdy moduł o tej samej nazwie występuje w kilku miejscach w ścieżce wyszukiwania. Rozważmy następującą strukturę pakietu:

```
mypkg\
    __init__.py
    main.py
    string.py
```

W ten sposób zdefiniowany jest pakiet `mypkg` zawierający moduły `mypkg.main` i `mypkg.string`. Założymy teraz, że moduł główny próbuje zaimportować moduł o nazwie `string`. W wersji 2.6 i nowszych Python zacznie wyszukiwanie w katalogu `mypkg`, realizując import względny. Znajdzie zapisany w nim plik o nazwie `print.py` i zaimportuje go, przypisując nazwę `string` w przestrzeni nazw `mypkg.main` pakietu `mypkg`.

Może się jednak okazać, że intencją programisty było zaimportowanie modułu `print` standardowej biblioteki Pythona. Niestety, w starszych wersjach Pythona nie ma prostego sposobu na zignorowanie modułu `mypkg.string` i wymuszenie importu z biblioteki standardowej lub innego modułu znajdującego się w ścieżce wyszukiwania. Co więcej, tego problemu nie możemy również rozwiązać, wykorzystując ścieżki importowania w pakiecie, ponieważ nie możemy zakładać, że układ ścieżek biblioteki standardowej będzie taki sam na każdej maszynie.

Innymi słowy, importy pakietów mogą wprowadzać niejednoznaczności: w ramach pakietu nie wiadomo, czy instrukcja `import spam` odnosi się do modułu w pakiecie, czy poza nim. A dokładniej, lokalny moduł lub pakiet może przesłonić (celowo lub przypadkowo) inny, znajdujący się w ścieżce wyszukiwania.

W praktyce użytkownicy Pythona unikają stosowania dla własnych modułów nazw zdefiniowanych w standardowej bibliotece (jeśli potrzebujesz standardowego modułu `string`, nie twórz modułu o tej samej nazwie). Jednak ta zasada nie wystarczy, jeśli w pakiecie zostanie przypadkowo przesłonięty moduł standardowej biblioteki; co więcej, w nowej wersji Pythona mogą pojawić się nowe moduły o nazwach użytych przez nas w programie. Kod wykorzystujący względne importy jest też trudniejszy do zrozumienia, ponieważ osoba czytająca może nie zrozumieć tego, który pakiet miał być zaimportowany. Lepiej jest dać w kodzie jednoznacznie do zrozumienia, jakie były intencje programisty.

Importy względne w 3.0

W celu rozwiązania tego problemu w Pythonie 3.0 (oraz opcjonalnie w 2.6) została zmodyfikowana reguła wyszukiwania nazw w importach wykonywanych w pakietach: importy są domyślnie bezwzględne. W tym modelu instrukcja `import` w postaci użytej w pliku `mypkg/main.py` zawsze spowoduje wyszukanie pliku poza pakietem, to znaczy zostanie użyty import bezwzględny w ramach ścieżki `sys.path`:

```
import string # Importuje moduł string spoza pakietu
```

Bezwzględne są również importy z użyciem instrukcji `from`, o ile nie zostanie zastosowana ścieżka z użyciem wiodących kropek.

```
from string import name # Importuje name z modułu string poza pakietem
```

Jeśli chcemy zaimportować moduł zdefiniowany w pakiecie bez podawania jego pełnej ścieżki w pakiecie, możemy użyć importu względnego, stosując kropkę w instrukcji `from`:

```
from . import string # Importuje mypkg.string (import względny)
```

Ta forma powoduje zaimportowanie modułu `string` zdefiniowanego w bieżącym pakiecie i jest odpowiednikiem instrukcji `from mypkg import string`. W przypadku zastosowania tej składni importu wyszukiwanie jest wykonywane wyłącznie w katalogu pakietu.

Składni importu względnego można użyć również do zaimportowania nazw z modułu:

```
from .string import name1, name2 # Importuje nazwy z mypkg.string
```

Ta instrukcja odwołuje się do modułu `string` zdefiniowanego w bieżącym pakiecie. Jeśli ten kod wystąpi w module `mypkg.main`, nastąpi import nazw `name1` i `name2` z `mypkg.string`.

Pojedyncza kropka w importie względnym efektywnie oznacza *bieżący* pakiet, czyli katalog, w którym zapisany jest plik, w którym zadeklarowano `import`. Dodatkowa kropka spowoduje względny import, rozpoczynając od katalogu *nadrzędnego*, na przykład:

```
from .. import spam # Importuje spam sąsiadujący z mypkg
```

Powyższy kod spowoduje zaimportowanie pakietu `spam` zdefiniowanego na tym samym poziomie, co `mypkg`. Oto inne przykłady ścieżek względnych na kilku poziomach wywoywanych z modułu `A.B.C`:

```
from . import D # Importuje A.B.D (. oznacza A.B)
from .. import E # Importuje A.E (. oznacza A)
from .D import X # Importuje A.B.D.X (. oznacza A.B)
from ..E import X # Importuje A.E.X (. oznacza A)
```

Względne importy a bezwzględne ścieżki pakietów

W module można też wskazać pełną ścieżkę do pakietu, wykorzystując składnię importów bezwzględnych. W poniższym przykładzie pakiet `mypkg` zostanie wczytany ze ścieżki wyszukiwania `sys.path`:

```
from mypkg import string # Importuje mypkg.string (import bezwzględny)
```

Mechanizm ten opiera swoje działanie na konfiguracji i zdefiniowanej kolejności ścieżek, natomiast w przypadku importów względnych nie ma tego typu niejednoznaczności. Co więcej, taki import jak przedstawiony w przykładzie wymaga, aby katalog, w którym zdefiniowany jest pakiet `mypkg`, był zadeklarowany w ścieżce wyszukiwania. Bezwzględne importy muszą określać pełną ścieżkę do pakietu od ścieżki wyszukiwania `sys.path`.

```
from system.section.mypkg import string # system znajduje się w katalogu zdefiniowanym w sys.path
```

W przypadku pakietów zawierających kilka poziomów podkatalogów ścieżka będzie dłuża, a wówczas szczególnie przydają się importy względne.

```
from . import string # Składnia importów względnych
```

W powyższym przykładzie pakiet bieżący jest przeszukiwany automatycznie, niezależnie od konfiguracji ścieżki wyszukiwania.

Zakres importów względnych

Importy względne mogą przy pierwszym kontakcie wydać się dość skomplikowanym zagadnieniem, ale wszystko znacznie się upraszcza po poznaniu pewnych podstawowych zasad:

- **Importy względne są stosowane wyłącznie w ramach pakietów.** Importy modułów spoza pakietu stosuje się tak, jak opisałem wcześniej, co powoduje przeszukanie katalogu roboczego skryptu oraz ścieżki wyszukiwania.
- **W względne importy stosuje się wyłącznie w instrukcji from.** Importy względne rozpoznawane są po tym, że po słowie kluczowym `from` następuje jedna lub więcej kropek. Nazwy modułów zawierające kropki, ale niezawierające kropki wiodącej są realizowane w sposób standardowy (zależnie od wersji Pythona), nie wzajemny.
- **Terminologia jest niejednoznaczna.** Niestety przy wprowadzaniu nowego mechanizmu importów względnych nie udało się uniknąć niejednoznaczności w stosowanej terminologii. Wszystkie importy są bowiem względne. Poza pakietem importy są względne w stosunku do katalogów zadeklarowanych w ścieżce wyszukiwania `sys.path`. Jak pokazano w rozdziale 21., ta ścieżka zawiera katalog roboczy programu, ścieżki zadeklarowane w zmiennej środowiska `PYTHONPATH` oraz standardowe biblioteki. W konsoli interaktywnej katalogiem roboczym programu jest po prostu katalog, z którego została uruchomiona konsola.

W przypadku importów wykonywanych z pakietów Python 2.6 w pierwszej kolejności przeszukuje katalog pakietu. W Pythonie 3.0 zmieniono ten mechanizm w taki sposób, aby importy niebędące importami jawnie względnymi działały z pominięciem przeszukiwania lokalnego katalogu pakietu. Do importów w ramach pakietu służy natomiast specjalna, nowa składnia importów względnych. Gdy mówimy o bezwzględnych importach w 3.0, mamy tak naprawdę na myśli importy względne w stosunku do ścieżki wyszukiwania `sys.path`. Gdy mówimy o importach względnych, mamy na myśli importy względne w stosunku do samego katalogu pakietu. Niektóre ścieżki zdefiniowane w `sys.path` mogą być oczywiście stosowane jako względne lub bezwzględne (można wymyślić jeszcze bardziej skomplikowane przykłady, ale nie chcę wprowadzać jeszcze większego zamieszania).

Innymi słowy: „wzajemne importy z pakietów” w 3.0 to w zasadzie rezygnacja ze stosowanej w Pythonie 2.X reguły wyszukiwania modułów w pakietach oraz dodanie składni wymuszającej wyszukiwanie względne w pakiecie. Jeśli ktoś pisał kod dla Pythona 2.X, pilnując, aby nie korzystać z wyszukiwania względnego (na przykład zawsze podawał pełną ścieżkę do modułu) zmiany wprowadzone w Pythonie 3.0 raczej nie będą stanowiły zagrożenia z punktu widzenia kompatybilności. W przeciwnym razie należy zmodyfikować takie importy, stosując składnię ich wymuszonego importu względnego.

Podsumowanie reguł wyszukiwania modułów

Reguły importowania pakietów i modułów w Pythonie 3.0 można podsumować w następujący sposób:

- Nazwy modułów (np. A) są wyszukiwane w każdym katalogu z listy `sys.path`, od lewej do prawej. Ta lista jest budowana z systemowych wartości domyślnych i uzupełniana o ustawienia konfigurowane przez użytkownika.
- Pakiety są po prostu katalogami zawierającymi moduły Pythona oraz specjalny plik `__init__.py`, który umożliwia stosowanie w importach ścieżki typu `A.B.C`. W takim importie katalog `A` powinien znajdować się w ścieżce wyszukiwania `sys.path`, `B` to podkatalog katalogu `A`, natomiast `C` jest modułem lub inną nazwą importowaną z `B`.
- W plikach pakietów zwykłe instrukcje importów stosują tę samą zasadę. Importy w pakietach wykorzystujące instrukcję `from` oraz ścieżkę modułu rozpoczynającą się od kropki stosują specjalną regułę importów względnych, to znaczy nazwa jest wyszukiwana wyłącznie w odniesieniu do pakietu, a wyszukiwanie w ścieżce `sys.path` nie jest wykonywane. Na przykład w instrukcji `from . import A` wyszukiwanie modułu `A` będzie realizowane wyłącznie w katalogu zawierającym plik, w którym zdefiniowano tę instrukcję.

Importy względne w działaniu

Wystarczy teorii, przeprowadźmy kilka testów, aby zademonstrować w praktyce koncepcję importów względnych.

Importy spoza pakietów

Jak wspomniałem wcześniej, mechanizm importów względnych nie ingeruje w możliwość importowania spoza pakietów. Dzięki temu poniższy kod importujący moduł `string` standardowej biblioteki Pythona zadziała zgodnie z oczekiwaniami:

```
C:\test> c:\Python30\python
>>> import string
>>> string
<module 'string' from 'c:\Python30\lib\string.py'>
```

Jeśli jednak do katalogu, w którym pracujemy, dodamy moduł o nazwie `string`, to załadowany zostanie ten moduł, ponieważ na pierwszym miejscu w ścieżce wyszukiwania znajduje się bieżący katalog roboczy.

```
# test\string.py
print('string' * 8)

C:\test> c:\Python30\python
>>> import string
stringstringstringstringstringstringstring
>>> string
<module 'string' from 'string.py'>
```

Innymi słowy, zwykłe importy nadal są względne w stosunku do katalogu roboczego (katalogu, w którym zapisany jest skrypt, lub katalogu bieżącego, z którego został uruchomiony). W rzeczywistości składnia importów względnych nie może być użyta w kodzie nieznajdującym się w modulem będącym elementem pakietu.

```
>>> from . import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Attempted relative import in non-package
```

W powyższym przykładzie oraz w demonstrowanych sesjach konsoli interaktywnej zachowanie będzie takie samo, jak w przypadku uruchomienia skryptu, ponieważ pierwszym elementem w `sys.path` będzie katalog roboczy, w którym została uruchomiona konsola interaktywna, lub katalog roboczy, w którym zapisany jest uruchamiany skrypt. Jedyna różnica polega na tym, że pierwszy element ścieżki `sys.path` jest bezwzględną ścieżką do katalogu, a nie pustym ciągiem znaków:

```
# test\main.py
import string
print(string)

C:\test> C:\python30\python main.py                                # Wyniki w 2.6 będą takie same
stringstringstringstringstringstringstring
<module 'string' from 'C:\test\string.py'>
```

Importy wewnętrz pakietów

Usuńmy lokalny moduł `string` i zbudujmy katalog pakietu zawierający dwa moduły oraz wymagany, ale pusty plik `test\pkg__init__.py` (jego utworzenie pominę w poniższym listingu):

```
C:\test> del string*
C:\test> mkdir pkg

# test\pkg\spam.py
import eggs                                         # <== Działa w 2.6, ale nie w 3.0!
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)
```

W pierwszym module tego pakietu próbujemy zaimportować drugi za pomocą zwykłej instrukcji `import`. W 2.6 w pierwszej kolejności Python próbuje wykonać import względny, zatem import zadziała, jednak w 3.0 taka instrukcja wykonuje wyłącznie import bezwzględny, przez co import się nie uda. Innymi słowy: w 2.6 najpierw przeszukiwany jest pakiet modułu, w którym wykonywany jest import; w 3.0 porzucono ten etap. Jest to jeden ze szczególnów implementacji 3.0, który psuje zgodność wstecz i należy mieć ten fakt na uwadze.

```
C:\test> c:\Python26\python
>>> import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test> c:\Python30\python
>>> import pkg.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "pkg\spam.py", line 1, in <module>
      import eggs
ImportError: No module named eggs
```

Aby nasze moduły działały prawidłowo w 2.6 i 3.0, należy zmodyfikować instrukcję importu w pierwszym z plików w taki sposób, by wykorzystać składnię importów względnych i wskażć Pythonowi, aby modułu `eggs` szukał w module lokalnym:

```

# test\pkg\spam.py
from . import eggs           # <== użycie względnych importów w 2.6 i 3.0
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)

C:\test> c:\Python26\python
>>> import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test> c:\Python30\python
>>> import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>
99999

```

Importy są nadal względne w stosunku do katalogu roboczego

Należy zwrócić uwagę na to, że moduły w pakietach nadal mają dostęp do modułów biblioteki standardowej, jak `string`. Jednak importy są nadal względne w stosunku do ścieżki wyszukiwania, nawet jeśli wykorzystują składnię wyszukiwania względnego. Jeśli do katalogu roboczego dodamy moduł `string`, to przy próbie importu zostanie załadowany właśnie on, nie moduł biblioteki standardowej. W 3.0 wyszukiwanie nazw w bieżącym pakiecie nie jest wykonywane, ale nie ma możliwości pominięcia katalogu roboczego programu (czyli katalogu, z którego został uruchomiony).

```

# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import eggs
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string           # <== Znajduje string w katalogu roboczym, nie w bibliotece standardowej!
print(string)

C:\test> c:\Python30\python    # W 2.6 wynik będzie taki sam
>>> import pkg.spam
stringstringstringstringstringstringstring
<module 'string' from 'string.py'>
99999

```

Użycie importów względnych i bezwzględnych

Sprawdźmy, w jaki sposób zasady wyszukiwania modułów wpływają na użycie modułów biblioteki standardowej. W tym celu jeszcze raz wyczyszcmy nasz pakiet. Znów usuniemy lokalny moduł `string` i zdefiniujemy nowy, ale w pakiecie.

```

C:\test> del string*
# test\pkg\spam.py
import string           # <== Względny w 2.6, bezwzględny w 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)

```

To, która wersja modułu `string` zostanie zimportowana, zależy od użytej wersji Pythona. Wersja 3.0 import w pierwszym pliku traktuje jako bezwzględny, pomijając moduły pakietu, ale w 2.6 tak się nie dzieje.

```
C:\test> c:\Python30\python
>>> import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Użycie składni importów względnych w 3.0 wymusza przeszukiwanie pakietu podobnie jak w 2.6, ale dzięki składni importów względnych mamy jawną kontrolę nad tym, który pakiet zostanie zimportowany. Dokładnie o obsługę takich sytuacji chodziło przy wprowadzaniu tej zmiany w Pythonie 3.0.

```
# test\pkg\spam.py
from . import string          # <== Import względny w 2.6 i 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test> c:\Python30\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Należy zwrócić uwagę na to, że składnia importów względnych jest w rzeczywistości deklaracją wiązania, nie jedynie preferencji. Jeśli usuniemy moduł `string.py` z naszego modułu, import względny zadeklarowany w `spam.py` nie uda się w 3.0 ani w 2.6, Python nie podejmie próby zainportowania modułu `string` ze standardowej ścieżki wyszukiwania (czyli z biblioteki standardowej ani jakiejkolwiek innej zdefiniowanej w ścieżce).

```
# test\pkg\spam.py
from . import string          # <== Nie działa, brak lokalnego string.py!

C:\test> C:\python30\python
>>> import pkg.spam
...pominęta część komunikatu...
ImportError: cannot import name string
```

Moduły wskazywane przez importy względne muszą istnieć w katalogu bieżącego pakietu.

Importy są nadal względne w stosunku do katalogu roboczego (cd.)

Importy bezwzględne pozwalają pominąć moduły pakietów, ale nadal są zależne od elementów ścieżki `sys.path`. W ostatnim teście zdefiniujmy dwa własne moduły `string`. Zapiszemy je w ten sposób, aby program miał je dostępne w pakiecie bieżącym, w katalogu roboczym programu oraz w bibliotece standardowej.

```
# test\string.py
print('string' * 8)
```

```
# test\pkg\spam.py
from . import string          # <== Względny w 2.6 i 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

Gdy zaimportujemy moduł `string` z użyciem składni importów względnych, otrzymamy wersję zapisaną w pakiecie, dokładnie tak, jak się tego spodziewamy:

```
C:\test> c:\Python30\python      # Takie same wyniki w 2.6
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Gdy użyjemy składni importów bezwzględnych, wynik importu będzie zależny od wersji Pythona: w 2.6 import ten będzie względny, w 3.0 „bezwzględny”, co w tym przypadku oznacza pominięcie wersji zdefiniowanej w pakiecie i użycie pakietu zdefiniowanego w ścieżce roboczej (wersja z biblioteki standardowej *nie będzie użyta*).

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
import string                  # <== Import względny w 2.6, "bezwzględny" w 3.0: ścieżka robocza!
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test> c:\Python30\python
>>> import pkg.spam
stringstringstringstringstringstringstring
<module 'string' from 'string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.pyc'>
```

Jak widzimy, pakiety mogą deklarować import modułów z lokalnych pakietów, ale importy pozostają względne w stosunku do ścieżki wyszukiwania. W tym przypadku plik zdefiniowany w katalogu roboczym programu przesłania moduł biblioteki standardowej o tej samej nazwie. Zmiany mechanizmu wyszukiwania modułów wprowadzone w 3.0 realizują jedynie możliwość wyboru modułu z bieżącego pakietu lub spoza niego (w ramach importów względnych lub bezwzględnych). Jednak mechanizm importu jest zależny od środowiska, w którym jest wykonywany, bezwzględne importy w 3.0 nadal nie są zabezpieczeniem przed efektami ubocznymi przesłonięcia nazw modułów przez inne zapisane w ścieżce wyszukiwania.

Warto przeprowadzić kilka eksperymentów z tymi przykładami w celu lepszego zrozumienia zasad importów. W rzeczywistości problemy z importami nie są tak powszechnne, jak można by się obawiać, wnioskując z przykładów: importy można ustukturalizować, ścieżkę wyszukiwania można modyfikować do własnych potrzeb, aby importy działały dokładnie tak, jak tego oczekujemy. Należy jednak mieć na uwadze, że importy w bardziej skomplikowanych konfiguracjach systemowych mogą być zależne od kontekstu, a na protokół importu modułów ma wpływ jakość projektu biblioteki aplikacji.



Wiedząc o możliwości użycia importów względnych, należy również pamiętać, że nie zawsze są najlepszym wyborem. Importy bezwzględne, które są względne w stosunku do ścieżki wyszukiwania, są nadal preferowaną formą importów zarówno w stosunku do niejawnych importów względnych (Python 2), jak i w jawnych deklaracjach importów względnych (Python 2 i 3).

Składania importów względnych oraz wprowadzone w 3.0 nowe reguły wyszukiwania w importach powodują, że importowanie modułów staje się bardziej czytelne, dzięki czemu kod jest łatwiejszy do zrozumienia i utrzymania. Pliki wykorzystujące importy względne są jednak związane z katalogiem pakietu i nie mogą być przeniesione do innych pakietów bez wprowadzania zmian w kodzie.

Oczywiście te fakty mogą mieć niewielki wpływ na pisany kod, dużo zależy od organizacji pakietów. Importy bezwzględne również mogą wymagać zmian w kodzie w przypadku zmian w układzie plików w pakietach.

Znaczenie pakietów modułów

Teraz, gdy pakiety stały się standardową częścią Pythona, często można spotkać większe rozszerzenia zewnętrzne udostępniane jako zbiór katalogów pakietów, a nie płaska lista modułów. Pakiet rozszerzeń *win32all* dla systemu Windows był na przykład jednym z pierwszych, które przeszły na system pakietów. Wiele z jego modułów z narzędziami znajduje się w pakietach importowanych za pomocą ścieżek. By na przykład załadować narzędzia COM po stronie klienta, należy wykorzystać poniższą instrukcję.

```
from win32com.client import constants, Dispatch
```

Powyższy wiersz kodu pobiera zmienne z modułu *client* pakietu *win32com* (podkatalogu instalacyjnego).

Importowanie pakietów jest również wszechobecne w kodzie wykonywanym w oparciu na języku Java implementacji Jython, ponieważ biblioteki Javy także organizowane są w hierarchii. W nowszych wydaniach Pythona narzędzia do obsługi poczty elektronicznej oraz XML są w podobny sposób zorganizowane w podkatalogi pakietów biblioteki standardowej, w Pythonie 3.0 w pakietach znajdziemy jeszcze więcej modułów (między innymi narzędzia GUI *tkinter*, narzędzia do obsługi HTTP i wiele innych).

Oto przykłady importów narzędzi ze standardowej biblioteki Pythona w 3.0:

```
from email.message import Message
from tkinter.filedialog import askopenfilename
from http.server import CGIHTTPRequestHandler
```

Bez względu na to, czy sami tworzymy katalogi pakietów, z pewnością będziemy kiedyś z nich coś importować.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy model importowania pakietów Pythona — opcjonalną, lecz przydatną metodę jawnego wymienienia części ścieżki katalogów prowadzących do modułów. Importowanie pakietów nadal odbywa się względem katalogu ze ścieżki wyszukiwania modułów, jednak zamiast pozostawić Pythonowi ręczne przejście ścieżki wyszukiwania, skrypt może podać resztę ścieżki do modułu w sposób jednoznaczny.

Jak widzieliśmy, pakiety nie tylko sprawiają, że importowanie w większych systemach jest bardziej zrozumiałe, ale także upraszczają ustawienia ścieżki wyszukiwania (jeśli wszystkie operacje importowania pomiędzy katalogami odbywają się względem wspólnego katalogu głównego). Pomagają one również usunąć niejednoznaczność w sytuacji, gdy istnieje większa liczba modułów o tej samej nazwie (dodanie nazwy katalogu zawierającego moduł do importu pakietu pomaga odróżnić moduły).

Omówiliśmy również model importów względnych, stosowany wyłącznie w pakietach. Jest to sposób wymuszenia importu plików zdefiniowanych w tym samym pakiecie polegający na zastosowaniu wiodących kropek w ścieżce importu w instrukcji `from`. Metoda ta zastępuje stosowaną w starszych wersjach Pythona regułę domyślnego wyszukiwania modułów w bieżącym katalogu pakietu.

W kolejnym rozdziale omówimy kilka bardziej zaawansowanych zagadnień związanych z modułami, takich jak składnia importowania względnego oraz zmienna trybu użycia `__name__`. Jak zawsze jednak zamknijmy rozdział krótkim quizem sprawdzającym wiadomości w nim przedstawione.

Sprawdź swoją wiedzę — quiz

1. Jaki jest cel umieszczania pliku `__init__.py` w katalogu pakietu modułu?
2. W jaki sposób możemy uniknąć powtarzania pełnej ścieżki pakietu za każdym razem, gdy odnosimy się do zawartości pakietu?
3. Które katalogi wymagają, by znajdował się w nich plik `__init__.py`?
4. Kiedy w przypadku importowania pakietów musimy użyć instrukcji `import` zamiast `from`?
5. Jaka jest różnica między instrukcją `from mypkg import spam` a `from . import spam`?

Sprawdź swoją wiedzę — odpowiedzi

1. Plik `__init__.py` służy do deklarowania i inicjalizacji pakietu modułu. Python automatycznie wykonuje jego kod za pierwszym razem, gdy w procesie importujemy moduł za pośrednictwem katalogu. Przypisane zmienne pliku stają się atrybutami obiektu modułu utworzonego w pamięci i odpowiadającego temu katalogowi. Nie jest on opcjonalny — nie możemy importować modułów za pomocą składni pakietów, jeśli katalog nie zawiera tego pliku.
2. By bezpośrednio skopiować zmienne z pakietu, należy użyć instrukcji `from` lub rozszerzenia `as` w połączeniu z instrukcją `import`, zastępując ścieżkę krótszym synonimem. W obu przypadkach ścieżka wymieniana jest tylko w jednym miejscu, czyli instrukcji `from` bądź `import`.
3. Każdy katalog wymieniony w instrukcjach `import` oraz `from` musi zawierać plik `__init__.py`. Pozostałe katalogi, w tym katalog zawierający komponent ścieżki pakietu znajdujący się najbardziej na lewo, nie muszą zawierać tego pliku.

4. W przypadku pakietów instrukcji `import` musimy użyć w miejscu `from` jedynie wtedy, gdy potrzebujemy uzyskać dostęp do tej samej zmiennej zdefiniowanej w więcej niż jednej ścieżce. Dzięki instrukcji `import` ścieżka sprawia, że referencje stają się unikalne, natomiast instrukcja `from` pozwala na wykorzystywanie tylko jednej wersji danej nazwy zmiennej.
5. Instrukcja `from mypkg import spam` definiuje import *bezwzględny*: pakiet `mypkg` jest wyszukiwany z pominięciem katalog pakietu, w którym występuje ta instrukcja, to znaczy przeszukiwana jest wyłącznie ścieżka `sys.path`. Instrukcja `from . import spam` definiuje import *względny*: nazwa `spam` jest poszukiwana w odniesieniu do pakietu, w którym występuje ta instrukcja.

Zaawansowane zagadnienia związane z modułami

Niniejszy rozdział kończy tę część książki zbiorem bardziej zaawansowanych zagadnień związanych z modułami — ukrywaniem danych, modułem `_future_`, zmienną `_name_`, zmianami `sys.path`, narzędziami służącymi do wyświetlania danych, wykonywaniem modułów po łańcuchu znaków nazwy czy przeładowywaniem przechodnim. Oprócz tego znajdzie się w nim umieszczone pod koniec każdej części książki omówienie pułapek związanych z przedstawionymi tutaj kwestiami, a także ćwiczenia końcowe.

Po drodze utworzymy bardziej rozbudowane niż dotychczas i przydatne narzędzia łączące w sobie funkcje oraz moduły. Podobnie do funkcji, moduły są bardziej wydajne, kiedy ich interfejsy są dobrze zdefiniowane, dlatego niniejszy rozdział zawiera również omówienie koncepcji związanych z projektowaniem modułów; o części z nich wspomnieliśmy w poprzednich rozdziałach.

Pomimo umieszczenia słowa „zaawansowane” w tytule rozdziału, tekst ten gromadzi także sporo dodatkowych zagadnień związanych z modułami. Ponieważ niektóre z nich są szeroko stosowane (zwłaszcza sztuczka ze zmienną `_name_`), należy im się koniecznie przyjrzeć przed przejściem do omówienia klas w kolejnej części książki.

Ukrywanie danych w modułach

Jak widzieliśmy, moduł Pythona eksportuje wszystkie zmienne przypisane na najwyższym poziomie jego pliku. Nie ma czegoś takiego, jak deklarowanie, które zmienne powinny być widoczne poza modelem, a które nie. Tak naprawdę nie da się zapobiec modyfikacji zmiennych wewnętrz modułu przez klienta.

W Pythonie ukrywanie danych w modułach jest konwencją, a nie ograniczeniem składniowym. Jeśli chcemy zniszczyć moduł, usuwając wszystkie jego zmienne, możemy to zrobić, jednak na szczęście nie zdarzyło mi się jeszcze spotkać programisty z takimi zapędami. Niektórzy puryści protestują przeciwko tak liberalnemu stosunkowi do ukrywania danych i twierdzą, że oznacza to, iż Python nie może implementować hermetyzacji (enkapsulacji). Hermetyzacja w Pythonie polega jednak raczej na tworzeniu pakietów, a nie ograniczeń.

Minimalizacja niebezpieczeństw użycia `from * — _X` oraz `_all_`

W specjalnym przypadku możemy poprzedzić nazwy zmiennych pojedynczym znakiem `_` (na przykład `_X`), by zapobiec skopiowaniu ich po zaimportowaniu zmiennych modułu za pomocą instrukcji `from *`. W zamierzeniach ma to zapobiec zanieczyszczaniu przestrzeni nazw. Ponieważ instrukcja `from *` kopiuje wszystkie nazwy zmiennych, kod importujący może otrzymać więcej, niż żądał (w tym zmienne nadpisujące jego własne). Znaki `_` nie są deklaracjami zmiennych jako „prywatnych” — takie zmienne nadal widzimy i możemy modyfikować za pomocą innych form importowania, takich jak instrukcja `import`.

Alternatywnie możemy uzyskać efekt ukrycia podobny do konwencji składniowej `_X`, przypisując listę łańcuchów nazw zmiennych do zmiennej `_all_` na najwyższym poziomie modułu, jak w poniższym kodzie.

```
_all_ = ["Error", "encode", "decode"]      # Eksportuje tylko te zmienne
```

Kiedy korzystamy z tej opcji, instrukcja `from *` skopiuje jedynie zmienne wymienione w liście `_all_`. W rezultacie jest ona przeciwieństwem konwencji zapisu `_X — _all_` identyfikującej zmienne, które powinny być skopiowane, natomiast `_X` te, które nie powinny. Python najpierw szuka w module listy `_all_`. Jeśli nie została ona zdefiniowana, instrukcja `from *` kopiuje wszystkie zmienne niezawierające pojedynczego znaku `_`.

Podobnie do konwencji zapisu `_X`, lista `_all_` ma znaczenie jedynie dla instrukcji `from *` i nie ma nic wspólnego z deklarowaniem zmiennej jako prywatnej. Twórcy modułów mogą wybrać dowolną z tych technik w implementacji modułów, których zachowanie zmienia się, kiedy wykorzystuje się je w połączeniu z instrukcją `from *`. W rozdziale 23. przy okazji omawiania list `_all_` w plikach pakietów `__init__.py` widzieliśmy, że listy te deklarują podmoduły, które mają być załadowane przez instrukcję `from *`.

Włączanie opcji z przyszłych wersji Pythona

Zmiany wprowadzane do Pythona, które potencjalnie mogą uniemożliwić działanie jakiegoś kodu, są wprowadzane stopniowo. Na początku pojawiają się jako opcjonalne rozszerzenia, które domyślnie są wyłączone. By je włączyć, należy skorzystać ze specjalnej postaci instrukcji `import`.

```
from __future__ import nazwa_opcji
```

Taka instrukcja powinna się pojawiać na górze pliku modułu (być może po łańcuchu znaków dokumentacji), ponieważ włącza ona specjalną komplikację kodu dla modułu. Można również wstawić tę instrukcję w sesji interaktywnej, by móc eksperymentować z przyszłymi modyfikacjami języka. W takiej sytuacji opcja taka będzie dostępna do końca sesji interaktywnej.

W poprzednich wydaniach książki musieliśmy na przykład korzystać z tej instrukcji, by móc zademonstrować funkcje generatora wymagające słowa kluczowego, które domyślnie nie były włączone (nosiły one nazwę `generators`). Użyliśmy tej instrukcji również w celu aktywowania prawdziwego dzielenia z wersji 3.0 w rozdziale 5., wywołania `print` z wersji 3.0 w rozdziale 11. oraz importowania bezwzględnego dla pakietów z wersji 3.0 w rozdziale 23.

Wszystkie takie zmiany mogą zniszczyć działanie istniejącego kodu w Pythonie 2.6, dlatego są włączane stopniowo, jako funkcje opcjonalne uruchamiane za pomocą specjalnej postaci instrukcji import.

Mieszane tryby użycia — `__name__` oraz `__main__`

A oto kolejna sztuczka powiązana z modułami, pozwalająca zarówno zaimportować plik jako moduł, jak i wykonać go jako samodzielny program. Każdy moduł ma wbudowany atrybut o nazwie `__name__`, który Python automatycznie ustawia w następujący sposób:

- Jeśli plik jest wykonywany jako plik programu najwyższego poziomu, atrybut `__name__` ustawiany jest po uruchomieniu na łańcuch znaków "`__main__`".
- Jeśli plik jest importowany, atrybut `__name__` jest zamiast tego ustawiany na nazwę modułu w formie znanej przez klienta.

W rezultacie moduł może sprawdzać swój własny atrybut `__name__` w celu sprawdzenia, czy jest wykonywany, czy też importowany. Założymy na przykład, że tworzymy poniższy plik modułu o nazwie `runme.py`, eksportujący jedną funkcję o nazwie `tester`.

```
def tester():
    print("Jest Gwiazdka w niebie...")

if __name__ == '__main__':
    tester()                                # Tylko przy wykonywaniu
                                                # A nie przy importowaniu
```

Moduł ten definiuje funkcję, którą inne pliki mogą importować i wykorzystywać w normalny sposób.

```
% python
>>> import runme
>>> runme.tester()
Jest Gwiazdka w niebie...
```

Moduł ten zawiera na dole również kod ustawiony w taki sposób, by wywoływać funkcję, kiedy plik ten wykonywany jest jako program.

```
% python runme.py
Jest Gwiazdka w niebie...
```

W rezultacie zmienna `__name__` modułu służy jako *opcja trybu użycia*, pozwalając na wykorzystanie kodu zarówno w postaci biblioteki, którą można zaimportować, jak i skryptu najwyższego poziomu. Choć sztuczka ta jest bardzo prosta, można ją spotkać w prawie każdym prawdziwym pliku programu Pythona, z jakim będziemy mieli do czynienia.

Miejscem, w którym najczęściej stosuje się testy `__name__`, jest tak zwany *kod samosprawdzający*. Mówiąc w skrócie, można umieścić kod testujący eksportowanie modułu w samym module, opakowując go w test `__name__` zapisany na dole pliku. W ten sposób możemy wykorzystać plik w kliencie, importując go, ale także sprawdzić jego logikę, wykonując go z powłoki systemowej lub za pomocą innej metody uruchamiania. W praktyce kod samosprawdzający na dole pliku, umieszczony pod testem `__name__`, jest chyba najprostszym i najczęściej spotykanym protokołem testów jednostkowych (ang. *unit testing*) w Pythonie. W rozdziale 35. omówimy inne często wykorzystywane opcje testowania kodu w Pythonie — jak zobaczymy, moduły `unittest` oraz `doctest` z biblioteki standardowej udostępniają bardziej zaawansowane narzędzia testujące.

Sztuczka z atrybutem `_name_` jest również często wykorzystywana, kiedy pisze się pliki, które można wykorzystywać zarówno jako narzędzia wiersza poleceń, jak i biblioteki narzędzi. Założymy na przykład, że piszemy w Pythonie skrypt odnajdujący pliki. Większe korzyści z kodu odniesiemy, kiedy spakujemy go w funkcje i dodamy na dole pliku test `_name_` automatycznie wywołujący te funkcje, gdy plik wykonywany jest samodzielnie. W ten sposób kod skryptu można wykorzystać również w innych programach.

Testy jednostkowe z wykorzystaniem `_name_`

Tak naprawdę już wcześniej w książce spotkaliśmy się ze świetnym przykładem sytuacji, w której sprawdzanie atrybutu `_name_` mogłoby być przydatne. W podrozdziale poświęconym argumentom w rozdziale 18. napisaliśmy skrypt obliczający wartość minimalną przekazanego zbioru argumentów.

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Kod testu samosprawdzającego
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Skrypt ten zawiera na dole kod samosprawdzający, zatem możemy go sprawdzać bez konieczności ponownego wpisywania wszystkiego w interaktywnym wierszu poleceń za każdym razem, gdy go wykonujemy. Problem z jego obecnym zapisem polega na tym, że dane wyjściowe wywołania kodu samosprawdzającego pokażą się za każdym razem, gdy plik ten zostanie zimportowany przez inny plik, który chce go wykorzystać jako narzędzie — nie do końca jest to opcja przyjazna dla użytkownika. By to poprawić, musimy opakować kod samosprawdzający w sprawdzenie atrybutu `_name_`, tak by był on uruchamiany tylko wtedy, gdy plik wykonywany jest jako skrypt najwyższego poziomu, a nie zimportowany moduł.

```
print('Jestem:', __name__)

def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y
def grtrthan(x, y): return x > y

if __name__ == '__main__':
    print(minmax(lessthan, 4, 2, 1, 5, 6, 3))      # Kod testu samosprawdzającego
    print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))
```

Wyświetlamy tutaj również wartość atrybutu `_name_` na górze pliku, by móc ją śledzić. Python tworzy i przypisuje tę zmienną trybu użycia, gdy tylko zacznie ładować plik. Kiedy wykonujemy ten plik jako skrypt najwyższego poziomu, wartość tego atrybutu ustawiana jest na `_main_`, dzięki czemu kod samosprawdzający startuje automatycznie.

```
% python min.py
Jestem: __main__
1
6
```

Jeśli jednak importujemy plik, atrybut nie ma wartości `__main__`, więc by wykonać funkcję, musimy w jawnym sposobie ją wywołać.

```
>>> import min
Jestem: min
>>> min.minmax(min.lessthan, 'm', 'i', 'e', 'l', 'o', 'n', 'k', 'a')
'a'
```

I znów, bez względu na to, czy technika ta wykorzystana zostanie do testowania, rezultat będzie taki, że nasz kod możemy wykorzystywać w dwóch rolach — jako moduł biblioteki narzędzi lub jako program wykonywalny.

Użycie argumentów wiersza poleceń z `__name__`

Poniżej znajduje się bardziej rozbudowany przykład modułu, demonstrujący inne częste zastosowanie sztuczki z `__name__`. Moduł `formats.py` definiuje narzędzia formatujące łańcuchy znaków na potrzeby kodu importującego, jednak sprawdza także swoją nazwę w celu przekonania się, czy wykonywany jest jako skrypt najwyższego poziomu. Jeśli tak jest, testuje i wykorzystuje argumenty podane w systemowym wierszu poleceń w celu wykonania gotowego lub przekazanego testu. W Pythonie lista `sys.argv` zawiera argumenty wiersza poleceń — to lista łańcuchów znaków odzwierciedlających słowa wpisane w wierszu poleceń, gdzie pierwszy element jest zawsze nazwą wykonywanego skryptu.

```
"""
Różne wyspecjalizowane narzędzia służące do formatowania wyświetlanego łańcuchów znaków.
Przetestuj mnie za pomocą gotowego testu samosprawdzającego lub argumentów wiersza poleceń.
"""

def commas(N):
    """
    Formatowanie N podobnego do liczby dodatniej w celu wyświetlenia z przecinkami pomiędzy grupami cyfr,
    jak w xxx,yyy,zzz.
    """
    digits = str(N)
    assert(digits.isdigit())
    result = ''
    while digits:
        digits, last3 = digits[:-3], digits[-3:]
        result = (last3 + ',' + result) if result else last3
    return result

def money(N, width=0):
    """
    Formatowanie liczby N w celu wyświetlenia jej jako kwoty w dolarach, z przecinkami, dwoma miejscami dziesiętnymi,
    początkowym symbolem waluty $ i znakiem, a także opcjonalnym dopełnieniem, jak w $ -xxx,yyy.zz.
    """
    sign = '-' if N < 0 else ''
    N = abs(N)
    whole = commas(int(N))
    fract = ('%.2f' % N)[-2:]
    format = '%s%s.%s' % (sign, whole, fract)
    return '$%s' % (width, format)

if __name__ == '__main__':
    def selftest():
        pass
```

```

tests = 0, 1                      # Nie działa: -1, 1.23
tests += 12, 123, 1234, 12345, 1234567
tests += 2 ** 32, 2 ** 100
for test in tests:
    print(commas(test))

print('')
tests = 0, 1, 1, 1.23, 1., 1.2, 3.14159
tests += 12.34, 12.344, 12.345, 12.346
tests += 2 ** 32, (2 ** 32 + .2345)
tests += 1.2345, 1.2, 0.2345
tests += 1.2345, 1.2, 0.2345
tests += (2 ** 32), (2**32 + .2345)
tests += (2 ** 100), (2 ** 100)
for test in tests:
    print('%s [%s]' % (money(test, 17), test))

import sys
if len(sys.argv) == 1:
    selftest()
else:
    print(money(float(sys.argv[1]), int(sys.argv[2])))

```

Powyższy plik działa tak samo w Pythonie 2.6 oraz 3.0. Po wykonaniu w sposób bezpośredni sprawdza samego siebie jak poprzednio, tym razem jednak wykorzystując opcje z wiersza poleceń w celu kontrolowania działania testu. By przekonać się, co wyświetla kod testu samospawdzającego, wystarczy wykonać powyższy plik w sposób bezpośredni bez podawania własnych argumentów wiersza poleceń. W celu przetestowania określonych łańcuchów znaków należy je przekazać w wierszu poleceń wraz z minimalną szerokością pola:

```

C:\misc> python formats.py 999999999 0
$999,999,999.00

C:\misc> python formats.py 999999999 0
$-999,999,999.00

C:\misc> python formats.py 123456789012345 0
$123,456,789,012,345.00

C:\misc> python formats.py 123456789012345 25
$ 123,456,789,012,345.00

C:\misc> python formats.py 123.456 0
$123.46

C:\misc> python formats.py 123.454 0
$-123.45

C:\misc> python formats.py
...gotowe testy - do wypróbowania samodzielnie...

```

Tak jak wcześniej, ponieważ kod ten wykorzystywany jest w podwójnym trybie użycia, możemy normalnie zimportować jego narzędzia w innych kontekstach w postaci komponentów biblioteki:

```

>>> from formats import money, commas
>>> money(123.456)
'$123.46'
>>> money(-9999999.99, 15)
'$ 9,999,999.99'
>>> X = 99999999999999999999
>>> '%s (%s)' % (commas(X), X)
'99,999,999,999,999,999 (99999999999999999999)'

```

Ponieważ plik ten wykorzystuje także opcję łańcuchów znaków dokumentacji wprowadzoną w rozdziale 15., w celu zapoznania się z jego narzędziami możemy również skorzystać z funkcji `help` — służy ona jako narzędzie ogólnego przeznaczenia:

```
>>> import formats
>>> help(formats)
Help on module formats:

NAME
    formats

FILE
    c:\misc\formats.py

DESCRIPTION
    Różne wyspecjalizowane narzędzia służące do formatowania wyświetlanych łańcuchów
    ↵znaków.
    Przetestuj mnie za pomocą gotowego testu samosprawdzającego lub argumentów wiersza
    ↵poleceń.

FUNCTIONS
    commas(N)
        Formatowanie N podobnego do liczby dodatniej w celu wyświetlenia z przecinkami
        ↵pomiędzy grupami cyfr, jak w xxx,yyy,zzz.

    money(N, width=0)
        Formatowanie liczby N w celu wyświetlenia jej jako kwoty w dolarach,
        ↵z przecinkami, dwoma miejscami dziesiętnymi, początkowym symbolem waluty $
        ↵i znakiem, a także opcjonalnym dopełnieniem, jak w $ -xxx,yyy.zz.
```

Argumenty wiersza poleceń można wykorzystać na podobne sposoby w celu podania danych wejściowych do skryptów, które mogą pakować kod w postaci funkcji i klas, tak by mógł on zostać wykorzystany ponownie przez kod importujący. Bardziej zaawansowane możliwości przetwarzania oparte na wierszu poleceń można znaleźć w modułach getopt oraz optparse w bibliotece standardowej Pythona, a także w dokumentacji. W pewnych sytuacjach można także skorzystać z wbudowanej funkcji `input` wprowadzonej w rozdziale 3. i wykorzystanej w rozdziale 10. w celu zachęcenia użytkownika powłoki do podania danych wejściowych do testów zamiast pobierania ich z wiersza poleceń.



Warto zapoznać się również z zamieszczonym w rozdziale 7. omówieniem nowej składni metody formatowania łańcuchów `{,d}`, która zostanie udostępniona w Pythonie 3.1 i nowszych wersjach. To rozszerzenie formatowania rozdziela grupy tysięczne za pomocą przecinków w sposób podobny do powyższego kodu. Przedstawiony tutaj moduł dodaje jednak formatowanie waluty i służy jako alternatywa dla ręcznego wstawiania przecinka w wersjach Pythona starszych od 3.1.

Modyfikacja ścieżki wyszukiwania modułów

W rozdziale 21. dowiedzieliśmy się, że ścieżka wyszukiwania modułów jest listą katalogów, którą można dostosować do własnych potrzeb za pomocą zmiennej środowiskowej `PYTHONPATH`, a także plików z rozszerzeniem `.pth`. Czego na razie nie pokazałem, to to, w jaki sposób program napisany w Pythonie może tak naprawdę zmodyfikować ścieżkę wyszukiwania, zmieniając wbudowaną listę o nazwie `sys.path` (atrybut `path` wbudowanego modułu `sys`). Lista `sys.path` inicjalizowana jest po uruchomieniu, a później możemy jej komponenty usuwać, dodawać i ustawiać od nowa w dowolny sposób.

```
>>> import sys  
>>> sys.path  
[ '', 'C:\\\\users', 'C:\\Windows\\\\system32\\\\python30.zip', ...pozostałe usunięto...]  
  
>>> sys.path.append('C:\\\\sourcedir')          # Rozszerzenie ścieżki wyszukiwania modułów  
>>> import string                          # Wszystkie importy przeszukują nowy katalog na końcu
```

Po dokonaniu takiej zmiany będzie ona miała wpływ na wszystkie przyszłe operacje importowania w tym programie Pythona, ponieważ wszystkie operacje importowania i wszystkie pliki współdzielą jedną listę sys.path. Listę tę można tak naprawdę modyfikować w dowolny sposób.

```
>>> sys.path = [r'd:\\temp']                  # Zmiana ścieżki wyszukiwania modułów  
>>> sys.path.append('c:\\lp4e\\\\examples')    # Tylko dla tego procesu  
>>> sys.path  
['d:\\temp', 'c:\\lp4e\\\\examples']  
  
>>> import string  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ImportError: No module named string
```

Tym samym technikę tę można wykorzystać do dynamicznej konfiguracji ścieżki wyszukiwania wewnętrz programu napisanego w Pythonie. Należy jednak uważać — jeśli usuniemy ze ścieżki kluczowy katalog, możemy utracić dostęp do kluczowych narzędzi. W poprzednim przykładzie nie mieliśmy już na przykład dostępu do modułu `string`, ponieważ usunęliśmy katalog biblioteki źródłowej Pythona ze ścieżki wyszukiwania.

Należy również pamiętać, że takie ustawienia `sys.path` trwają jedynie na czas wykonywania sesji Pythona lub programu (z technicznego punktu widzenia: procesu), który je utworzył. Nie są one zachowywane po zakończeniu Pythona. Konfiguracja ścieżki ze zmiennej środowiskowej `PYTHONPATH` i plików `.pth` zachowywana jest w systemie operacyjnym, a nie w działającym programie utworzonym w Pythonie, przez co jest bardziej globalna. Jest pobierana przez każdy program działający na komputerze i zachowywana również po jego zakończeniu.

Rozszerzenie as dla instrukcji import oraz from

Instrukcje `import` oraz `from` zostały rozszerzone w taki sposób, by pozwalać na nadanie zaimportowanym zmiennym innych nazw w naszym skrypcie. Poniższa instrukcja `import`:

```
import nazwamodułu as nazwa
```

jest odpowiednikiem następującego kodu:

```
import nazwamodułu  
nazwa = nazwamodułu  
del nazwamodułu                                # Nie zachowujemy oryginalnej nazwy
```

Po takiej operacji importowania możemy (i tak naprawdę musimy) użyć nazwy wymienionej po `as` w celu odniesienia się do modułu. Działa to również dla instrukcji `from`, przypisując zmienną zaimportowaną z pliku do innej zmiennej w skrypcie docelowym.

```
from nazwamodułu import nazwaatrybutu as nazwa
```

Rozszerzenie to jest często wykorzystywane do podawania krótszych synonimów dłuższych nazw, a także w celu uniknięcia konfliktu nazw w przypadku wykorzystywania w skrypcie zmiennej, która w przeciwnym razie zostałaby nadpisana za pomocą normalnej instrukcji importowania.

```

import naprawdędługanazwamodułu as nazwa      # Użycie krótszego skrótu
name.func()

from moduł1 import narzędzie as narz1          # "narzędzie" może istnieć tylko raz
from moduł2 import narzędzie as narz2
narz1(); narz2()

```

Przydaje się to również w przypadku podawania krótkich, prostych nazw całych ścieżek do katalogów, kiedy wykorzystujemy importowanie pakietów przedstawione w rozdziale 23.

```

import dir1.dir2.mod as mod                      # Pełna ścieżka podana tylko raz
mod.func()

```

Moduły są obiektami — metaprogramy

Ponieważ moduły udostępniają większość swoich najciekawszych właściwości w postaci wbudowanych atrybutów, łatwo jest pisać programy zarządzające innymi programami. Zazwyczaj nazywamy je *metaprogramami*, ponieważ działają one ponad innymi systemami. Nazywa się to również *introspekcją*, ponieważ programy mogą widzieć i przetwarzać mechanizmy wewnętrzne obiektów. Introspekcja jest koncepcją zaawansowaną, jednak może się przydać do tworzenia narzędzi programistycznych.

By na przykład otrzymać atrybut o nazwie `name` z modułu o nazwie `M`, możemy skorzystać z kwalifikacji (składni z kropką) lub zindeksować słownik atrybutów modułu udostępniany jako wbudowany atrybut `__dict__` — spotkaliśmy się z nim w rozdziale 22. Python eksportuje również listę wszystkich załadowanych modułów jako słownik `sys.modules` (czyli atrybut `modules` modułu `sys`) i udostępnia wbudowaną funkcję `getattr` pozwalającą na pobranie atrybutów z ich łańcuchów nazw. Przypomina to kod `object.attr`, jednak `attr` jest wyrażeniem zwracającym łańcuch znaków w czasie wykonania. Z tego powodu poniższe wyrażenia pozwalają uzyskać dostęp do tego samego atrybutu i obiektu.

```

M.name                                         # Kwalifikacja obiektu
M.__dict__['name']                            # Ręczne indeksowanie słownika przestrzeni nazw
sys.modules['M'].name                         # Ręczne indeksowanie tabeli załadowanych modułów
getattr(M, 'name')                           # Wywołanie wbudowanej funkcji pobierającej

```

Udostępniając w ten sposób mechanizmy wewnętrzne modułu, Python pomaga nam tworzyć programy o programach.¹ Poniżej znajduje się na przykład moduł `mydir.py`, wprowadzający te koncepcje w życie w celu zaimplementowania własnej wersji wbudowanej funkcji `dir`. Definiuje on i eksportuje funkcję o nazwie `listing`, przyjmującą obiekt modułu jako argument i wyświetlającą sformatowany listing przestrzeni nazw modułu.

```

"""
mydir.py: moduł wymieniający przestrzeń nazw innych modułów
"""

seplen = 60
sepchr = '-'

def listing(moduł):
    """ Wyświetla zawartość przestrzeni nazw modułu 'moduł'.
        Wyswietlany jest nazwa modułu, nazwy jego atrybutów i wartości tych atrybutów.
    """
    print(moduł.__name__)
    for name in moduł.__dict__:
        if name[0] != '_':
            print(sepchr * seplen, name, moduł.__dict__[name])

```

¹ Jak widzieliśmy w rozdziale 17., ponieważ funkcja może uzyskać dostęp do zawierającego ją modułu za pomocą przejścia tabeli `sys.modules`, można również emulować efekt instrukcji `global`. Wynik zastosowania kodu `global X; X = 0` można symulować (choć zwiększać ilość kodu!), wpisując wewnętrz funkcji kod `import sys; glob = sys.modules[__name__]; glob.X = 0`. Należy pamiętać, że każdy moduł otrzymuje atrybut `__name__`, „za darmo”. Jest on widoczny jako zmienna globalna wewnętrz funkcji tego modułu. Ta sztuczka udostępnia kolejny sposób modyfikacji zmiennych lokalnych oraz globalnych o tej samej nazwie wewnętrz funkcji.

```

def listing(module, verbose=True):
    sepline = sepchr * seplen
    if verbose:
        print(sepline)
        print('nazwa:', module.__name__, 'plik:', module.__file__)
        print(sepline)

    count = 0
    for attr in module.__dict__:                      # Przeszukanie kluczy przestrzeni nazw
        print('%02d) %s' % (count, attr), end = ' ')
        if attr.startswith('__'):
            print('<zmienna wbudowana>')           # Pominiecie __file__ itd.
        else:
            print(getattr(module, attr))             # To samo co __dict__[attr]
        count += 1

    if verbose:
        print(sepline)
        print(module.__name__, 'ma %d zmiennych' % count)
        print(sepline)

if __name__ == '__main__':
    import mydir
    listing(mydir)                                    # Test samosprawdzający — lista samego siebie

```

Warto zwrócić uwagę na łańcuch znaków dokumentacji u góry kodu. Tak jak w poprzednim przykładzie pliku *formats.py*, z uwagi na to, że możemy chcieć użyć modułu jako uniwersalnego narzędzia, dodaliśmy do niego łańcuch znaków dokumentacji, by informacje na jego temat były dostępne za pośrednictwem atrybutów *__doc__* lub funkcji *help* (więcej informacji na ten temat znajduje się w rozdziale 15.).

```

>>> import mydir
>>> help(mydir)
Help on module mydir:

NAME
    mydir - mydir.py: moduł wymieniający przestrzeń nazw innych modułów

FILE
    c:\users\veramark\mark\mydir.py

FUNCTIONS
    listing(module, verbose=True)

DATA
    sepchr = '-'
    seplen = 60

```

Na dole modułu znajduje się również logika *samosprawdzająca*, która importuje i wymienia zmienne tego modułu. Poniżej widać, co zwraca moduł *mydir.py* w Pythonie 3.0 (by użyć go w Pythonie 2.6, należy włączyć obsługę wywołań *print* za pomocą importowania z *__future__* opisanego w rozdziale 11.; słowo kluczowe *end* działa tylko w wersji 3.0):

```

C:\Users\veramark\Mark> c:\Python30\python mydir.py
-----
nazwa: mydir plik: C:\Users\veramark\Mark\mydir.py
-----
00) seplen 60
01) __builtins__ <zmienna wbudowana>
02) __file__ <zmienna wbudowana>
03) __package__ <zmienna wbudowana>
04) listing <function listing at 0x026D3B70>
05) __name__ <zmienna wbudowana>

```

```
06) sepchr -
07) __doc__ <zmienna wbudowana>
-----
mydir ma 8 zmiennych
-----
```

By użyć modułu jako narzędzia przedstawiającego listę dla innych modułów, wystarczy przekazać moduły jako obiekty do funkcji tego pliku. Poniżej wymieniono atrybuty modułu GUI tkinter z biblioteki standardowej (w Pythonie 2.6 — Tkinter):

```
>>> import mydir
>>> import tkinter
>>> mydir.listing(tkinter)
-----
nazwa: tkinter plik: c:\PYTHON30\lib\tkinter\__init__.py
-----
00) getdouble <class 'float'>
01) MULTIPLE multiple
02) mainloop <function mainloop at 0x02913B70>
03) Canvas <class 'tkinter.Canvas'>
04) AtSellast <function AtSellast at 0x028FA7C8>
...pominęto wiele zmiennych...
151) StringVar <class 'tkinter.StringVar'>
152) ARC arc
153) At <function At at 0x028FA738>
154) NSEW nsew
155) SCROLL scroll
-----
tkinter ma 156 zmiennych
-----
```

Z setattr spotkamy się ponownie później. Należy tutaj zwrócić uwagę na to, że mydir jest programem pozwalającym na przeglądanie innych programów. Ponieważ Python udostępnia mechanizmy wewnętrzne obiektów, możemy je przetwarzać w sposób generyczny.²

Importowanie modułów za pomocą łańcucha znaków nazwy

Nazwa modułu w instrukcji import lub from jest zapisaną na stałe nazwą zmiennej. Czasami jednak program otrzyma w czasie wykonywania nazwę importowanego modułu w postaci łańcucha znaków (na przykład kiedy użytkownik wybierze nazwę modułu w GUI). Niestety, nie da się wykorzystać instrukcji import do bezpośredniego załadowania modułu, mając podaną jego nazwę w postaci łańcucha znaków. Python oczekuje tutaj nazwy zmiennej, a nie łańcucha znaków, co widać w poniższym przykładzie.

```
>>> import "string"
      File "<stdin>", line 1
          import "string"
                  ^
SyntaxError: invalid syntax
```

² Narzędzia takie, jak mydir.listing mogą być ładowane z wyprzedzeniem do interaktywnej przestrzeni nazw za pomocą importowania ich w pliku, do którego odnosi się zmienna środowiskowa PYTHONPATH. Ponieważ kod z pliku startowego działa w interaktywnej przestrzeni nazw (moduле __main__), importowanie wspólnych narzędzi w pliku startowym może nam zaoszczędzić nieco wpisywania. Więcej informacji na ten temat znajduje się w dodatku A.

Nie zadziała również proste przypisanie łańcucha znaków do zmiennej.

```
x = "string"  
import x
```

Python spróbuje tutaj zainportować plik *x.py*, a nie moduł *string*. Nazwa podana w instrukcji *import* staje się zmienną przypisaną do załadowanego modułu, a także identyfikuje plik zewnętrzny.

By obejść te ograniczenia, musimy skorzystać ze specjalnych narzędzi, które załadują moduł dynamicznie z łańcucha znaków generowanego w czasie wykonywania. Najbardziej uniwersalnym podejściem jest skonstruowanie instrukcji *import* jako łańcucha kodu Pythona i przekazanie go do wbudowanej funkcji *exec* w celu wykonania (*exec* w Pythonie 2.6 jest instrukcją, jednak można jej użyć w dokładnie ten sam sposób jak poniżej — nawiasy zostaną po prostu zignorowane).

```
>>> modname = "string"  
>>> exec('import ' + modname) # Wykonanie łańcucha kodu  
>>> string # Zainportowany w tej przestrzeni nazw  
<module 'string' from 'c:\Python30\lib\string.py'>
```

Funkcja *exec* (i jej krewniak przeznaczony dla wyrażeń — *eval*) kompiluje łańcuch kodu i przekazuje go do interpretera Pythona w celu wykonania. W Pythonie kompilator kodu bajtowego jest dostępny w czasie wykonywania, dlatego w taki sposób możemy tworzyć programy konstruujące i wykonujące inne programy. Domyślnie instrukcja *exec* wykonuje kod w zakresie bieżącym, jednak możemy to zmienić, przekazując opcjonalne słowniki przestrzeni nazw.

Jedyną prawdziwą wadą instrukcji *exec* jest to, że musi ona kompilować instrukcję *import* za każdym wykonaniem. Jeśli wykonywana jest wiele razy, kod może działać szybciej, jeśli zamiast tego wykorzysta funkcję wbudowaną *__import__* do załadowania łańcucha znaków z nazwą modułu. Efekt będzie podobny, jednak funkcja *__import__* zwraca obiekt modułu, zatem by obiekt ten zachować, trzeba go przypisać do zmiennej.

```
>>> modname = "string"  
>>> string = __import__(modname)  
>>> string  
<module 'string' from 'c:\Python30\lib\string.py'>
```

Przechodnie przeładowywane modułów

Przeładowywane modułów omawialiśmy w rozdziale 22. jako sposób pobrania aktualnień kodu bez zatrzymywania i wznowiania działania programu. Kiedy jednak przeładowujemy moduł, Python przeładowuje jedynie plik tego określonego modułu. Nie przeładowuje automatycznie wszystkich modułów, które ten plik importuje.

Jeśli na przykład zainportujemy moduł A, a z kolei A importuje moduły B oraz C, operacja przeładowania odnosi się tylko do A, a nie również do B oraz C. Instrukcje wewnętrz modułu A importujące moduły B i C są w czasie przeładowywania wykonywane raz jeszcze, jednak po prostu pobierają one załadowane obiekty plików B oraz C (zakładając, że zostały one wcześniej zainportowane). W prawdziwym kodzie plik *A.py* wyglądałby następująco.

```
import B  
import C  
  
% python # Nie jest przeładowywany razem z A  
          # Importuje załadowany już moduł
```

```
>>> . . .
>>> from imp import reload
>>> reload(A)
```

Domyślnie oznacza to, że nie należy polegać na tym, iż przeładowywane pobierze zmiany we wszystkich modułach w sposób przechodni. Zamiast tego należy użyć kilku osobnych wywołań funkcji `reload`, które niezależnie aktualnią poszczególne komponenty. W przypadku dużych systemów testowanych interaktywnie może to wymagać sporo pracy. Jeśli jest to pożądane, możemy zaprojektować nasz system w taki sposób, by przeładowywał on swoje komponenty automatycznie, dodając wywołanie `reload` w module nadrzędnym, takim jak `A`, jednak komplikuje to kod modułu.

Lepszym rozwiążaniem jest napisanie uniwersalnego narzędzia, które przeładowania tego typu wykonuje automatycznie, przeglądając atrybuty `__dict__` modułów i sprawdzając `type` dla każdego elementu w celu odnalezienia zagnieżdżonych modułów, które można przeładować. Takie narzędzie mogłoby być funkcją pomocniczą wywołującą samą siebie w sposób *rekurencyjny*, co pozwoliłoby jej przechodzić dowolnie zbudowane łańcuchy zależnych od siebie importów. Atrybuty `__dict__` modułów zostały wprowadzone nieco wcześniej, w podrozdziale „Moduły są obiektami — metaprogramy” znajdującym się na początku rozdziału, natomiast wywołanie `type` zostało zaprezentowane w rozdziale 9.; teraz musimy tylko połączyć ze sobą te dwa narzędzia.

Wymieniony poniżej moduł `reloadall.py` zawiera na przykład funkcję `reload_all`, która automatycznie przeładowuje moduł, każdy moduł importowany przez ten moduł i tak dalej aż do końca każdego łańcucha importowanych plików. Wykorzystuje słownik do przechowywania informacji o przeładowanych już modułach, a także rekurencję, która pozwala przechodzić łańcuch modułów. Oprócz tego w kodzie zastosowano moduł `types` z biblioteki standardowej, który z wyprzedzeniem definiuje wyniki wywołania `type` dla typów wbudowanych. Technika ze słownikiem `visited` została tutaj wykorzystana w celu uniknięcia cykli w przypadku, gdy importowanie jest rekurencyjne lub zbędne, ponieważ obiekty modułów mogą być kluczami słownika (jak wiemy z rozdziału 5., podobną funkcjonalność dałby nam zbiór, gdybyśmy do wstawiania użyli `visited.add(module)`).

```
"""
reloadall.py: przechodnie przeładowanie zagnieżdżonych modułów
"""

import types
from imp import reload                                # from wymagane w wersji 3.0

def status(module):
    print('przeładowanie' + module.__name__)

def transitive_reload(module, visited):
    if not module in visited:
        status(module)                                # Przechwycenie cykli i powtarzających się modułów
        visited[module] = None                         # Przeladowanie modulu
        reload(module)                                # Przejście do jego modułów podrzędnych
        for attrobj in module.__dict__.values():      # Dla wszystkich atrybutów
            if type(attrobj) == types.ModuleType:       # Rekurencja, jeśli to moduł
                transitive_reload(attrobj, visited)

def reload_all(*args):
    visited = {}
    for arg in args:
        if type(arg) == types.ModuleType:
```

```
transitive_reload(arg, visited)

if __name__ == '__main__':
    import reloadall
    reload_all(reloadall)                                # Kod samosprawdzający: przeladowuje siebie
                                                        # Powinien to przeladować
```

By skorzystać z tego narzędzia, należy zaimportować jego funkcję `reload_all` i przekazać do niej nazwę załadowanego już modułu (tak samo, jak robiliśmy to dla funkcji `reload`). Kiedy plik jest wykonywany samodzielnie, jego test samosprawdzający powoduje przetestowanie samego siebie — musi siebie zaimportować, ponieważ jego własna nazwa nie jest zdefiniowana w pliku bez operacji importowania. Kod ten działa zarówno w wersji 3.0, jak i 2.6 i wyświetla identyczne dane wyjściowe, ponieważ w wywołaniu `print` użyliśmy znaku + zamiast przecinka.

```
C:\misc> C:\Python30\python reloadall.py
przeładowanie reloadall
przeładowanie types
```

Poniżej znajduje się przykład działania tego kodu w Pythonie 3.0 na modułach biblioteki standardowej. Warto zwrócić uwagę na importowanie `os` przez `tkinter`; `tkinter` dociera do `sys`, zanim `os` będzie to w stanie zrobić. By przetestować ten kod w Pythonie 2.6, należy `tkinter` zastąpić za pomocą `Tkinter`.

```
>>> from reloadall import reload_all
>>> import os, tkinter

>>> reload_all(os)
przeładowanie os
przeładowanie copyreg
przeładowanie ntpath
przeładowanie genericpath
przeładowanie stat
przeładowanie sys
przeładowanie errno

>>> reload_all(tkinter)
przeładowanie tkinter
przeładowanie _tkinter
przeładowanie tkinter._fix
przeładowanie sys
przeładowanie ctypes
przeładowanie os
przeładowanie copyreg
przeładowanie ntpath
przeładowanie genericpath
przeładowanie stat
przeładowanie errno
przeładowanie ctypes._endian
przeładowanie tkinter.constants
```

A oto jeszcze jedna sesja zestawiająca efekt działania normalnego i przechodniego przeładowywania. Zmiany wprowadzone do dwóch zagnieżdżonych plików nie są pobierane przez przeładowania, jeśli nie skorzystamy z narzędzia do przeładowywania przechodniego:

```
import b                                              # Plik a.py
X = 1

import c                                              # Plik b.py
Y = 2

Z = 3                                                 # Plik c.py

C:\misc> C:\Python30\python
```

```

>>> import a
>>> a.X, a.b.Y, a.b.c.Z
(1, 2, 3)

# Modyfikacja wartości przypisów we wszystkich plikach i zapisanie zmian

>>> from imp import reload
>>> reload(a)                                # Normalne przeładowanie tylko na najwyższym poziomie
<module 'a' from 'a.py'>
>>> a.X, a.b.Y, a.b.c.Z
(111, 2, 3)

>>> from reloadall import reload_all
>>> reload_all(a)
przeładowanie a
przeładowanie b
przeładowanie c
>>> a.X, a.b.Y, a.b.c.Z                      # Przeładowanie także wszystkich zagnieżdżonych modułów
(111, 222, 333)

```

Zachęcam każdego do zapoznania się i samodzielnego eksperymentowania z tym kodem. Moduł ten to kolejne narzędzie do importowania, które można włączyć do własnej biblioteki kodu źródłowego.

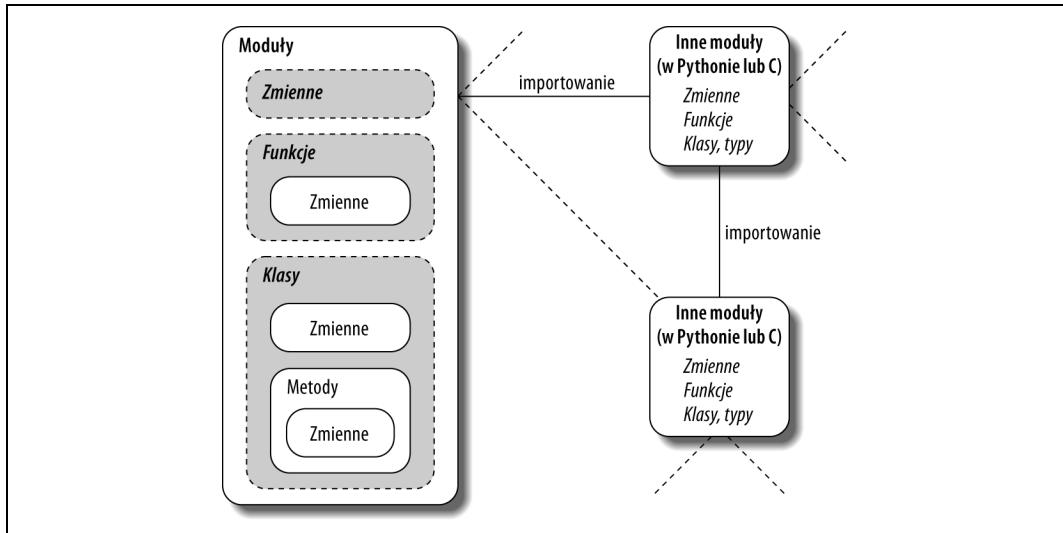
Projektowanie modułów

Podobnie jak w funkcjach, w przypadku modułów konieczne są pewne kompromisy w zakresie projektowania. Musimy się zastanowić, które funkcje mają trafić do których modułów czy jak zbudować mechanizmy komunikowania między modułami. Wszystko to stanie się jasne, kiedy zaczniemy w Pythonie pisać większe systemy. Istnieją jednak pewne ogólne reguły, o których warto pamiętać.

- **W Pythonie zawsze jesteśmy w module.** Nie da się pisać kodu, który nie jest częścią jakiegoś modułu. Tak naprawdę kod wpisany w sesji interaktywnej w rzeczywistości trafia do wbudowanego modułu o nazwie `__main__`. Jedyną cechą odróżniającą sesję interaktywną od zwykłych modułów jest to, że kod jest natychmiast wykonywany i usuwany, a wyniki wyrażeń są automatycznie wyświetlane.
- **Należy minimalizować połączenia pomiędzy modułami w postaci zmiennych globalnych.** Tak jak funkcje, moduły działają najlepiej, kiedy są pisane jako zamknięte pojemniki. Powinny one być tak niezależne od zmiennych globalnych z innych modułów, jak to tylko możliwe, z wyjątkiem funkcji i klas z nich importowanych.
- **Maksymalizacja spójności modułów — jeden cel.** Możemy zminimalizować liczbę połączeń i zależności między modułami, maksymalizując jednocześnie ich spójność. Jeśli wszystkie komponenty modułu mają wspólny cel, mniej prawdopodobna jest konieczność polegania na zmiennych zewnętrznych.
- **Moduły powinny rzadko modyfikować zmienne z innych modułów.** Tę kwestię w rozdziale 17. ilustrowaliśmy kodem, jednak warto o niej przypomnieć. Można wykorzystywać zmienne globalne zdefiniowane w innym module (w taki sposób klient importuje usługi), jednak modyfikacja zmiennych globalnych z innego modułu jest często symptomem problemu w zakresie projektowania. Istnieją oczywiście wyjątki od tej reguły, jednak powinniśmy próbować komunikować wyniki za pomocą narzędzi takich, jak argumenty i wartości zwracane z funkcji, a nie zmiany wprowadzane w innym module. W przeciwnym razie

wartości zmiennych globalnych stają się zależne od kolejności różnych odległych operacji przypisania w innych plikach, a same moduły trudniej jest zrozumieć oraz wykorzystać ponownie.

Jako podsumowanie na rysunku 24.1 naszkicowano środowisko, w którym operują moduły. Moduły zawierają zmienne, funkcje, klasy oraz inne moduły (jeśli są one importowane). Funkcje mają własne zmienne lokalne, podobnie jak klasy, czyli obiekty znajdujące się wewnątrz modułów, z którymi spotkamy się w rozdziale 25.



Rysunek 24.1. Środowisko wykonywania modułów. Moduły są importowane, jednak jednocześnie mogą importować i wykorzystywać inne moduły napisane w Pythonie lub innym języku programowania, takim jak C. Moduły zawierają kolejno zmienne, funkcje oraz klasy, które wykonują ich pracę. Funkcje i klasy mogą natomiast zawierać zmienne oraz inne własne elementy. Programy są po prostu zbiorami modułów

Pułapki związane z modułami

W niniejszym podrozdziale przyjrzymy się tradycyjnemu zbiorowi przypadków granicznych, które sprawiają, że życie początkujących programistów Pythona nabiera barw. Niektóre są tak dziwne, że trudno było wymyślić dla nich przykłady, jednak większość ilustruje jakąś ważną cechę języka.

W kodzie najwyższego poziomu kolejność instrukcji ma znaczenie

Kiedy moduł jest importowany po raz pierwszy (lub przeładowywany), Python wykonuje jego instrukcje jedna po drugiej, od góry pliku do dołu. Ma to kilka konsekwencji w zakresie referencji, o których należy wspomnieć.

- Kod na najwyższym poziomie pliku modułu (niezagnieżdżony w funkcji) wykonywany jest, kiedy tylko Python dotrze do niego w czasie importowania. Z tego powodu nie może się odnosić do zmiennych przypisanych niżej w pliku.

- Kod wewnętrz ciała funkcji nie jest wykonywany, dopóki funkcja nie zostanie wywołana. Ponieważ zmienne z funkcji nie są obliczane, dopóki funkcja nie zostanie faktycznie wykonana, zazwyczaj mogą odnosić się do zmiennych znajdujących się w dowolnym miejscu pliku.

Zasadniczo referencje odnoszące się do zmiennych zdefiniowanych niżej są problemem jedynie w kodzie najwyższego poziomu modułu, który wykonywany jest natychmiast. Funkcje mogą się odnosić do dowolnych zmiennych. Poniżej znajduje się przykład ilustrujący to zjawisko.

```
func1()                                # Błąd: "func1" nie została jeszcze przypisana

def func1():
    print(func2())                      # OK: "func2" zostanie wyszukana później

func1()                                # Błąd: "func2" nie została jeszcze przypisana

def func2():
    return "Witam"

func1()                                # OK: "func1" i "func2" zostały już przypisane
```

Kiedy powyższy plik zostanie zimportowany (lub wykonany jako samodzielny program), Python wykonuje jego instrukcje od góry do dołu. Pierwsze wywołanie `func1` nie powiedzie się, ponieważ instrukcja `def` dla `func1` nie była jeszcze wykonana. Wywołanie `func2` wewnętrz `func1` działa, jeśli tylko Python dotrze do instrukcji `def` dla `func2` przed wywołaniem funkcji `func1` (a tak nie było, kiedy wykonywane było drugie wywołanie `func1`). Ostatnie wywołanie `func1` na dole pliku działa, ponieważ zmienne `func1` oraz `func2` zostały już przypisane.

Mieszanie instrukcji `def` z kodem najwyższego poziomu jest nie tylko trudne do odczytania, ale również zależne od kolejności instrukcji. Z reguły jeśli musimy mieszać kod wykonywany natychmiast z instrukcjami `def`, instrukcje `def` umieszcza się na górze pliku, a kod najwyższego poziomu na dole. W ten sposób funkcje na pewno zostaną zdefiniowane i przypisane, zanim wykonany zostanie kod je wykorzystujący.

Instrukcja `from` kopiuje nazwy, jednak łączy już nie

Choć instrukcja `from` jest powszechnie używana, jest także w Pythonie źródłem różnych potencjalnych pułapek. Instrukcja ta jest tak naprawdę przypisaniem do zmiennych w zakresie kodu importującego — operacją kopiowania nazw, a nie tworzenia do nich aliasów. Implikacje takiego rozwiązania są takie same jak dla wszystkich operacji przypisania w Pythonie, jednak dość subtelne, w szczególności w przypadku kodu współdzielącego obiekty istniejące w różnych plikach. Założymy na przykład, że definiujemy następujący moduł o nazwie `nested1.py`.

```
# Plik nested1.py
X = 99
def printer(): print(X)
```

Jeśli zimportujemy jego dwie zmienne za pomocą instrukcji `from` użytej w innym module (`nested2.py`), otrzymamy kopie zmiennych, a nie łączy do nich. Modyfikacja zmiennej w module importującym zmienia wiązanie jedynie lokalnej wersji tej zmiennej, a nie zmiennej znajdującej się w module `nested1.py`.

```
# Plik nested2.py
from nested1 import X, printer
X = 88                                # Kopiuje zmienne
                                         # Modyfikuje tylko X lokalne!
```

```
printer()  
# X w nested1 nadal ma wartość 99  
  
% python nested2.py  
99
```

Jeśli instrukcję import wykorzystamy do pobrania całego modułu, a następnie przypiszemy coś do zmiennej ze składnią kwalifikującą, modyfikujemy oryginalną zmienną z pliku *nested1.py*. Kwalifikacja kieruje Pythona do zmiennej w obiekcie modułu, a nie do zmiennej w pliku importującym (*nested3.py*).

```
# Plik nested3.py  
import nested1  
nested1.X = 88  
nested1.printer()  
  
# Pobranie modułu jako całości  
# OK: modyfikuje X z modułu nested1  
  
% python nested3.py  
88
```

Instrukcja from * może zaciemnić znaczenie zmiennych

Wspomniałem o tym wcześniej, ale z pominięciem szczegółów. Ponieważ kiedy używamy instrukcji `from *`, nie wymieniamy zmiennych, które chcemy zaimportować, możemy w ten sposób przypadkowo nadpisać zmienne już wykorzystywane w zakresie modułu importującego. Co gorsza, może nam być trudno ustalić, skąd wzięła się jakaś zmienna. Jest tak szczególnie wtedy, gdy forma `from *` jest wykorzystywana na większej liczbie importowanych plików niż jeden.

Jeśli na przykład użyjemy instrukcji `from *` na trzech modułach, nie będziemy wiedzieli, co tak naprawdę znaczy jakieś wywołanie funkcji, o ile nie przeszukamy wcześniej wszystkich trzech modułów zewnętrznych (z których każdy może się znajdować w innym katalogu).

```
>>> from module1 import *  
>>> from module2 import *  
>>> from module3 import *  
>>> ...  
  
>>> func()  
# Źle: może po cichu nadpisać nasze zmienne  
# Gorzej: skąd mamy wiedzieć, co dostajemy?  
# Hę???
```

Rozwiążanie tego problemu jest proste — nie należy tak robić. W instrukcjach `from` trzeba spróbować w jawny sposób wymieniać listę atrybutów, które chcemy zaimportować, a także ograniczyć użycie instrukcji `from *` do jednego importowanego modułu na plik. W ten sposób wszystkie niezdefiniowane zmienne muszą należeć do modułu podanego w tej jednej instrukcji `from *`. Możemy zresztą całkowicie pozbyć się tego problemu, jeśli zamiast `from` zawsze będziemy używali instrukcji `import`, jednak taka rada jest zbyt surowa. Tak jak wszystkie inne elementy języka programowania, `from` jest przydatnym narzędziem, jeśli używa się go z rozwagą. Nawet powyższy przykład nie jest zresztą całkowicie niedopuszczalny — program może wykorzystywać tę technikę w celu zebrania zmiennych w jednym miejscu dla wygody, o ile wszyscy o tym wiedzą.

Funkcja reload może nie mieć wpływu na obiekty importowane za pomocą from

A oto kolejna pułapka związana z wykorzystywaniem instrukcji `from`. Jak wspominaliśmy wcześniej, ponieważ instrukcja `from` po wykonaniu kopiuje (przypisuje) zmienne, nie istnieje żadne łącze z powrotem do modułu, z którego pochodzi dana zmienna. Zmienne zaimpor-

towane za pomocą `from` stają się po prostu referencjami do obiektów, do których odnoszą się również zmienne o tych samych nazwach w pliku importowanym.

Ze względu na to zachowanie przeładowanie importowanego modułu nie ma żadnego wpływu na moduły, które zainportowały jego zmienne za pomocą instrukcji `from`. Oznacza to, że zmienne klienta nadal będą się odnosiły do oryginalnych obiektów pobranych za pomocą `from`, nawet jeśli zmienne te w oryginalnym module mają teraz inną wartość.

```
from module import X  
.  
from imp import reload  
reload(module)  
X
```

X może nie odzwierciedlać przeładowanych modułów!
Modyfikacja modułu, ale nie zmiennych
Nadal odnosi się do starego obiektu

By przeładowywanie było bardziej efektywne, należy zamiast `from` użyć instrukcji `import` oraz składni kwalifikującej zmienną. Ponieważ kwalifikacja zawsze oznacza przejście do modułu, w ten sposób po przeładowaniu odnalezione zostaną nowe wiązania w zmiennych modułu.

```
import module  
.  
from imp import reload  
reload(module)  
module.X
```

Pobranie modułu, nie zmiennych
Modyfikuje moduł w miejscu
Bieżące X: odzwierciedla przeładowanie modułu

Funkcja `reload` i instrukcja `from` a testowanie interaktywne

Tak naprawdę problem zasygnalizowany powyżej może być jeszcze bardziej subtelny, niż mogłoby się wydawać. W rozdziale 3. ostrzegałem, że zazwyczaj lepiej jest nie uruchamiać programów za pomocą importowania i przeładowywania ze względu na związane z takim rozwiązaniem trudności. Wszystko jeszcze się pogarsza, kiedy do tej techniki dolożymy instrukcję `from`. Osoby początkujące często napotykają takie problemy w sytuacjach jak opisana poniżej. Założymy, że po otwarciu pliku modułu w oknie edytora tekstu uruchamiamy sesję interaktywną, która ładuje moduł za pomocą `from` w celu przetestowania go.

```
from module import function  
function(1, 2, 3)
```

Po odnalezieniu błędu wracamy do okna edytora, wprowadzamy poprawkę i próbujemy w następujący sposób przeładować moduł.

```
from imp import reload  
reload(module)
```

Takie rozwiązanie jednak nie działa, ponieważ instrukcja `from` przypisała zmienną `function`, a nie `module`. By odwołać się do modułu z funkcji `reload`, musimy przynajmniej raz załadować ten moduł za pomocą instrukcji `import`.

```
from imp import reload  
import module  
reload(module)  
function(1, 2, 3)
```

Takie rozwiązanie również nie działa — funkcja `reload` aktualnia obiekt modułu, ale, zgodnie z informacjami przedstawionymi wyżej, zmienne takie jak `function`, skopiowane z modułu w przeszłości, nadal odnoszą się do *starych wersji obiektów* (w tym przypadku oryginalnej wersji funkcji). By naprawdę otrzymać nową funkcję, musimy odwołać się do niej jako `module.function` po funkcji `reload`, ewentualnie raz jeszcze wykonać instrukcję `from`.

```
from imp import reload
import module
reload(module)
from module import function
function(1, 2, 3) # Można też się poddać i użyć module.function()
```

Dopiero teraz nowa wersja funkcji zostanie wykonana.

Jak widać, wykorzystywanie funkcji `reload` w połączeniu z `from` wiąże się z pewnymi problemami — nie tylko musimy pamiętać o przeładowywaniu modułu po jego zaimportowaniu, ale także musimy po przeładowaniu wykonać instrukcję `from` raz jeszcze. Jest to na tyle skomplikowane, że niejeden ekspert ma z tym od czasu do czasu problemy. Tak naprawdę w Pythonie 3.0 sytuacja jeszcze się pogorszyła, ponieważ trzeba dodatkowo pamiętać o zaimportowaniu samego `reload`!

W skrócie, nie powinniśmy oczekwać, że funkcja `reload` i instrukcja `from` będą ze sobą dobrze współpracować. Najlepiej jest w ogóle ich ze sobą nie łączyć, `reload` wykorzystywać w połączeniu z `import` lub uruchamiać programy w inny sposób, zgodnie z sugestiami z rozdziału 3. (na przykład korzystając z opcji *Run Module* z menu *Run* aplikacji IDLE, klikając ikony programu lub używając systemowego wiersza poleceń czy wbudowanej funkcji `exec`).

Rekurencyjne importowanie za pomocą `from` może nie działać

Najdziwniejszą (i na szczęście rzadko spotykana) pułapkę zostawiłem na koniec. Ponieważ operacja importowania wykonuje instrukcje pliku od góry do dołu, musimy uważać, kiedy wykorzystujemy moduły, które się wzajemnie importują (co znane jest jako *importowanie rekurencyjne*). Ponieważ nie wszystkie instrukcje z modułu mogą zostać wykonane, kiedy zaimportuje on inny moduł, niektóre zmienne mogą jeszcze nie istnieć.

Jeśli wykorzystujemy instrukcję `import` do pobrania modułu w całości, może to mieć znaczenie lub nie. Dostęp do zmiennych odbędzie się dopiero przy późniejszym zastosowaniu składni kwalifikującej w celu pobrania ich wartości. Jeśli jednak w celu pobrania określonych zmiennych użyjemy `from`, musimy pamiętać, że będziemy mieli dostęp jedynie do tych zmiennych modułu, które zostały już przypisane.

Rozważmy na przykład dwa poniższe moduły o nazwach `recur1` i `recur2`. Moduł `recur1` przypisuje zmienną `X`, a następnie importuje moduł `recur2` przed przypisaniem zmiennej `Y`. W tym momencie `recur2` może pobrać `recur1` jako całość za pomocą instrukcji `import` (moduł ten istnieje już wewnętrznej tabeli modułów Pythona), jeśli jednak używa `from`, zobaczy tylko zmienną `X`. Zmienna `Y` przypisana poniżej instrukcji `import` w `recur1` jeszcze nie istnieje, dla tego otrzymamy błąd.

```
# Plik recur1.py
X = 1
import recur2 # Wykonuje recur2, jeśli obiekt modułu nie istnieje
Y = 2

# Plik recur2.py
from recur1 import X # OK: "X" jest już przypisane
from recur1 import Y # Błąd: "Y" nie jest przypisane

C:\misc> C:\Python30\python
>>> import recur1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
File "recur1.py", line 2, in <module>
    import recur2
File "recur2.py", line 2, in <module>
    from recur1 import Y
ImportError: cannot import name Y
```

Python unika ponownego wykonywania instrukcji modułu `recur1`, kiedy jest on importowany rekurencyjnie z `recur2` (w przeciwnym razie operacja importowania powodowałaby nieskończoną pętlę skryptu). W momencie zimportowania przez `recur2` przestrzeń nazw modułu `recur1` jest jednak niekompletna.

Rozwiążanie? Nie należy używać instrukcji `from` w rekurencyjnych operacjach importowania. Jeśli jednak tak zrobimy, Python nie zablokuje się co prawda w cyklu, ale nasze programy znowu będą uzależnione od kolejności instrukcji w modułach.

Istnieją dwa wyjścia z tej sytuacji:

- Zazwyczaj można wyeliminować takie cykle importowania za pomocą rozważnego projektowania — maksymalizacja spójności i minimalizacja połączeń to dobre pierwsze kroki.
- Jeśli nie możemy całkowicie przerwać cykli, należy opóźnić dostęp do nazwy modułu za pomocą wykorzystania instrukcji `import` oraz składni kwalifikującej (zamiast stosowania `from`) lub wykonania instrukcji `from` albo wewnątrz funkcji (zamiast na najwyższym poziomie modułu), albo na dole pliku, tak by opóźnić ich wykonanie.

Podsumowanie rozdziału

W niniejszym rozdziale omówiono niektóre bardziej zaawansowane koncepcje związane z modułami. Zapoznaliśmy się między innymi z technikami ukrywania danych, z włączaniem nowych możliwości języka za pomocą modułu `__future__`, ze zmienną trybu użycia `__name__` czy przeładowywaniem przechodnim i importowaniem po łańcuchu znaków nazwy. Przyjrzelismy się również różnym zagadnieniom związanym z projektowaniem modułów, a także często popełnianym błędem związanym z modułami, co powinno pomóc nam uniknąć popełniania ich we własnym kodzie.

Kolejny rozdział rozpoczyna nasze omówienie narzędzia programowania zorientowanego obiektywo w Pythonie — klasy. Większość informacji z ostatnich kilku rozdziałów będzie miała zastosowanie również tutaj — klasy istnieją w modułach i są przestrzeniami nazw, jednak dodają do wyszukiwania atrybutów dodatkowy komponent o nazwie „wyszukiwanie dziedziczenia”. Ponieważ jest to ostatni rozdział tej części książki, zanim zagłębiemy się w ten nowy temat, warto zapoznać się ze zbiorem ćwiczeń końcowych. Najpierw jednak pora na quiz podsumowujący informacje przedstawione w niniejszym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Co specjalnego jest w zmiennych znajdujących się na najwyższym poziomie modułu, rozpoczynających się od pojedynczego znaku `_`?
2. Co oznacza ustawienie zmiennej `__name__` modułu na łańcuch znaków `"__main__"`?

3. Jeśli użytkownik wpisze w sesji interaktywnej nazwę modułu do przetestowania, w jaki sposób możemy ten moduł zimportować?
4. Czym różni się modyfikacja `sys.path` od ustawienia `PYTHONPATH` w celu zmodyfikowania ścieżki wyszukiwania modułów?
5. Skoro moduł `_future_` pozwala nam importować przyszłe opcje języka, czy istnieje jakiś sposób na importowanie opcji z przeszłości?

Sprawdź swoją wiedzę — odpowiedzi

1. Zmienne na najwyższym poziomie pliku modułu, których nazwy rozpoczynają się od pojedynczego znaku `_`, nie są kopiowane do zakresu importującego, kiedy używamy instrukcji `from *`. Można jednak uzyskać do nich dostęp za pomocą instrukcji `import` czy normalnej postaci instrukcji `from`.
2. Jeśli zmienna modułu `_name_` jest łańcuchem znaków `"__main__"`, oznacza to, że plik jest wykonywany jako skrypt najwyższego poziomu, a nie importowany z innego pliku programu. Inaczej mówiąc, plik ten jest wykorzystywany jako program, a nie biblioteka.
3. Dane od użytkownika zazwyczaj trafiają do skryptu w postaci łańcucha znaków. By zaimportować podany moduł, kiedy mamy łańcuch znaków jego nazwy, można utworzyć i wykonać instrukcję `import` za pomocą `exec` lub przekazać łańcuch znaków nazwy do wywołania funkcji `_import_`.
4. Modyfikacja `sys.path` ma wpływ jedynie na działający program i jest tymczasowa — zmiana znika, kiedy program kończy swe działanie. Ustawienia `PYTHONPATH` zachowywane są w systemie operacyjnym i są globalnie odczytywane przez wszystkie programy znajdujące się na komputerze. Modyfikacja tych ustawień jest trwała i wykracza poza czas działania programu.
5. Nie, nie da się importować opcji Pythona dostępnych w przeszłości. Możemy instalować (lub uparcie wykorzystywać) starsze wersje języka, jednak najnowszy Python jest zazwyczaj najlepszym Pythonem.

Sprawdź swoją wiedzę — ćwiczenia do części piątej

Rozwiązań ćwiczeń znajdują się w podrozdziale „Część V Moduły” w dodatku B.

1. *Podstawy importowania.* Należy napisać program zliczający wiersze oraz znaki pliku (podobnie do aplikacji `wc` w systemie Unix). Za pomocą edytora tekstu należy utworzyć moduł Pythona o nazwie `mymod.py` eksportujący trzy zmienne najwyższego poziomu:
 - funkcję `countLines(name)` odczytującą plik wejściowy i zliczającą liczbę znajdujących się w nim wierszy (wskazówka: większość pracy wykona za nas metoda `file.readlines`; resztą zajmie się `len`),
 - funkcję `countChars(name)` wczytującą plik wejściowy i zliczającą liczbę znajdujących się w nim znaków (wskazówka: metoda `file.read` zwraca jeden łańcuch znaków),
 - funkcję `test(name)` wywołującą obie funkcje zliczające dla określonego pliku wejściowego. Nazwa pliku może zostać przekazana, wpisana na stałe, podana za pomocą wbudowanej

funkcji `input` lub pobrana z wiersza poleceń za pomocą listy `sys.argv` zaprezentowanej w przykładzie z `formats.py` w tym rozdziale. Na początek możemy założyć, że będzie przekazanym argumentem funkcji.

Wszystkie trzy funkcje modułu `mymod` powinny oczekiwąć podania łańcucha znaków z nazwą pliku. Jeśli któraś z funkcji będzie miała więcej niż dwa czy trzy wiersze, oznacza to, że próbujemy za dużo zrobić ręcznie — należy skorzystać z podanych wskazówek!

Następnie należy przetestować moduł w sesji interaktywnej, wykorzystując instrukcję `import` oraz referencje atrybutów w celu pobrania wyeksportowanych funkcji. Czy zmienna środowiskowa `PYTHONPATH` musi zawierać katalog, w którym utworzyliśmy moduł `mymod.py`? Należy spróbować wykonać moduł na samym sobie, na przykład za pomocą `test →("mymod.py")`. Warto zauważyc, że funkcja `test` otwiera moduł dwukrotnie. Osoby szczególnie ambitne mogą spróbować to naprawić, przekazując obiekt otwartego pliku do dwóch funkcji zliczających (wskazówka: `file.seek(0)` przewija plik).

2. *Instrukcje `from` i `from *`.* Należy przetestować moduł `mymod` z ćwiczenia 1. w sesji interaktywnej, wykorzystując instrukcję `from` do bezpośredniego załadowania eksportowanych funkcji — najpierw po ich nazwach, a później za pomocą wariantu `from *`, który pobiera wszystko.
3. *Wartość `__main__`.* Należy do modułu `mymod` dodać wiersz wywołujący automatycznie funkcję `test` tylko wtedy, gdy moduł uruchamiany jest jako skrypt, a nie importowany. Dodany wiersz będzie najprawdopodobniej sprawdzał wartość `__name__` pod kątem łańcucha znaków "`__main__`", tak jak robiliśmy to w tym rozdziale. Należy spróbować wykonać moduł w systemowym wierszu poleceń, a później zaimportować go i przetestować jego funkcje w sesji interaktywnej. Czy nadal działa on w obu trybach?
4. *Zagnieżdżone importowanie.* Należy utworzyć drugi moduł — `myclient.py` — importujący `mymod` i testujący jego funkcje. Następnie należy wykonać moduł `myclient` z systemowego wiersza poleceń. Jeśli `myclient` używa instrukcji `from` do pobrania atrybutów modułu `mymod`, czy funkcje `mymod` będą dostępne z najwyższego poziomu `myclient`? Co będzie, jeśli importujemy moduł za pomocą instrukcji `import`? Należy spróbować zapisać w module `myclient` oba warianty i przetestować je interaktywnie, importując ten moduł i badając jego atrybut `__dict__`.
5. *Importowanie pakietów.* Należy zaimportować plik z pakietu. Po utworzeniu podkatalogu o nazwie `mypkg` zagnieżdżonego w katalogu znajdującym się w ścieżce wyszukiwania importowanych modułów należy przenieść tam plik `mymod.py` utworzony w trzech pierwszych ćwiczeniach i spróbować zaimportować go za pomocą operacji importowania pakietu w formie `import mypkg.mymod`.

Żeby takie rozwiązanie zadziałało, będziemy musieli do katalogu, do którego przenieśliśmy plik modułu, dodać plik `__init__.py`. Technika ta powinna jednak działać na wszystkich najważniejszych platformach (między innymi z tego powodu Python wykorzystuje znak kropki jako separator katalogów). Utworzony katalog pakietów może po prostu być podkatalogiem tego, w którym aktualnie pracujemy, dzięki czemu zostanie on odnaleziony przez komponent katalogu głównego ścieżki wyszukiwania i nie będziemy musieli tej ścieżki dodatkowo konfigurować. Do pliku `__init__.py` należy dodać jakiś kod i sprawdzić, czy jest on wykonywany przy każdej operacji importu.

6. *Przeładowywania.* Należy poeksperymentować z przeładowywaniem modułów — wykonać testy z przykładu `changer.py` z rozdziału 22., modyfikując komunikat wywoływanej funkcji,

a także jej zachowanie — bez zatrzymywania interpretera Pythona. W zależności od systemu możemy modyfikować moduł `chainer` w innym oknie lub zawiesić interpreter Pythona i edytować plik w tym samym oknie (w systemie Unix skrót `Ctrl+Z` zazwyczaj zawiesza bieżący proces, a polecenie `fg` go wznowia).

7. *Importowanie wzajemne*³. W podrozdziale poświęconym pułapkom związanych z importowaniem rekurencyjnym widać było, że importowanie `recur1` powodowało wystąpienie błędu. Jeśli jednak ponownie uruchomimy Pythona i zaimportujemy `recur2` interaktywnie, błąd ten nie pojawi się, co można samodzielnie sprawdzić. Dlaczego importowanie `recur2` działa, ale `recur1` już nie? Wskazówka: Python przechowuje nowe moduły we wbudowanej tabeli `sys.modules` (słowniku) przed wykonaniem ich kodu. Późniejsze operacje importowania pobierają najpierw moduł z tabeli, bez względu na to, czy jest on „kompletny”, czy też nie. Teraz można spróbować wykonać moduł `recur1` jako plik skryptu najwyższego poziomu za pomocą polecenia `python recur1.py`. Czy otrzymujemy ten sam błąd, który pojawi się przy interaktywnym importowaniu modułu? Dlaczego? Wskazówka: kiedy moduły są wykonywane jako programy, nie są importowane, dlatego ten przypadek daje taki sam efekt jak interaktywne importowanie `recur2`; `recur2` jest pierwszym importowanym modułem. Co się dzieje, kiedy wykonamy `recur2` jako skrypt?

³ Warto zauważyć, że operacje importowania tego typu są w praktyce wyjątkową rzadkością. Z drugiej strony, jeśli rozumiemy, dlaczego takie rozwiązanie jest potencjalnie problematyczne, wiemy bardzo dużo o semantyce importowania w Pythonie.

Klasy i programowanie zorientowane obiektowo

Programowanie zorientowane obiektowo

Dotychczas w książce używaliśmy pojęcia „obiekt” bardzo ogólnie. Tak naprawdę kod napisany przez nas dotychczas był *oparty na obiektach* — przekazywaliśmy obiekty w skryptach, wykorzystywaliśmy je w wyrażeniach, wywoływaliśmy ich metody. By nasz kod mógł się jednak zaliczać do prawdziwie *zorientowanego obiektowo* (ang. *object-oriented*, *OO*), nasze obiekty będą musiały uczestniczyć w czymś, co znane jest pod nazwą *hierarchii dziedziczenia*.

Niniejszy rozdział rozpoczyna nasze omówienie *klasy Pythona* — narzędzi wykorzystywanego do implementowania w Pythonie nowych typów obiektów obsługujących dziedziczenie. Klasy są w Pythonie podstawowym narzędziem programowania zorientowanego obiektowo (ang. *object-oriented programming*, *OOP*), dlatego w niniejszej części książki przyjrzymy się również podstawom tej koncepcji. Programowanie zorientowane obiektowo oferuje inne i często bardziej wydajne sposoby rozumienia programowania, w których kod jest poddawany faktoryzacji w celu zminimalizowania jego powtarzalności. Nowe programy pisze się natomiast, dostosowując istniejący kod, zamiast modyfikować go w miejscu.

W Pythonie klasy tworzone są za pomocą nowej instrukcji `class`. Jak zobaczymy, obiekty zdefiniowane za pomocą klas mogą w dużej mierze wyglądać tak, jak typy wbudowane omówione wcześniej. Tak naprawdę klasy stosują jedynie — i rozszerzają — koncepcje, o których już wspominaliśmy. Są one w przybliżeniu pakietami funkcji, które wykorzystują oraz przetwarzają wbudowane typy obiektów. Klasy są jednak zaprojektowane w taki sposób, by tworzyły nowe obiekty i nimi zarządzaly. Obsługują również *dziedziczenie* — mechanizm dostosowywania kodu oraz jego ponownego wykorzystywania, który wykracza poza wszystko, co dotychczas widzieliśmy.

Jedna ważna uwaga: w Pythonie programowanie zorientowane obiektowo jest całkowicie opcjonalne i nie musimy z klas korzystać od razu. Tak naprawdę mnóstwo zadań możemy wykonać za pomocą prostszych konstrukcji, takich jak funkcje, a nawet prosty kod skryptu najwyższego poziomu. Ponieważ wykorzystywanie klas wymaga pewnego planowania z wyprzedzeniem, zazwyczaj jest bardziej interesujące dla osób, które pracują w trybie *strategcznym* (zaangażowanych w długofalowe tworzenie produktów) niż tych pracujących w trybie *taktycznym* (gdzie nigdy nie ma za dużo czasu).

Mimo to — jak zobaczymy w tej części książki — klasy okazują się jednym z najbardziej użytecznych narzędzi udostępnianych przez Pythona. Kiedy się ich właściwie używa, mogą znacznie

skrócić czas tworzenia oprogramowania. Są one wykorzystywane w popularnych narzędziach Pythona, takich jak API graficznego interfejsu użytkownika tkinter, dlatego większość programistów Pythona uznałaby, że znajomość klas na podstawowym poziomie może być naprawdę przydatna.

Po co używa się klas?

Jakiś czas temu powiedziałem, że programy „robią coś z różnymi rzeczami”. W uproszczeniu klasy to tylko sposób definiowania nowych rodzajów owych „rzeczy”, odzwierciedlających prawdziwe obiekty w domenie programu. Założymy na przykład, że decydujemy się zaimplementować hipotetycznego robota wytwarzającego pizzę, który posłużył nam za przykład w rozdziale 16. Jeśli zaimplementujemy go za pomocą klas, można będzie modelować większą część jego prawdziwej struktury oraz relacji. Przydadzą nam się tutaj dwa aspekty programowania zorientowanego obiektowo.

Dziedziczenie

Roboty wytwarzające pizzę są rodzajami robotów, dlatego posiadają cechy właściwe wszystkim robotom. W terminologii programowania zorientowanego obiektowo mówimy, że „dziedziczą” one właściwości po ogólnej kategorii wszystkich robotów. Te wspólne właściwości wystarczy zaimplementować tylko raz dla przypadku ogólnego, a później użyć ponownie dla wszystkich typów robotów tworzonych w przyszłości.

Kompozycja

Roboty wytwarzające pizzę są tak naprawdę zbiorami komponentów, które działają razem jako zespół. Żeby nasz robot odniósł sukces, musi na przykład mieć ramiona zagniatujące ciasto, napęd umożliwiający przetransportowanie pizzy do pieca i tym podobne. W terminologii programowania zorientowanego obiektowo nasz robot jest przykładem kompozycji — zawiera inne obiekty, które aktywuje, by móc wykonywać swoje funkcje. Każdy komponent może być utworzony jako klasa definiująca własne zachowanie i relacje.

Ogólne pojęcia z dziedziny programowania zorientowanego obiektowo, takie jak dziedziczenie oraz kompozycja, mają zastosowanie do każdej aplikacji, którą można rozbić na zbiór obiektów. W typowych systemach GUI interfejsy są na przykład napisane jako zbiór widgetów — przycisków, podpisów i tym podobnych — które są rysowane, kiedy narysowany zostanie ich pojemnik (*kompozycja*). Co więcej, możemy również pisać własne widgety — przyciski z unikalnymi czcionkami, podpisy z nowymi schematami kolorów, które będą wyspecjalizowanymi wersjami bardziej ogólnych narzędzi interfejsu (*dziedziczenie*).

Z punktu widzenia programowania klasy są jednostkami programów Pythona, podobnie jak funkcje oraz moduły. Są kolejnym pojęciem służącym do pakowania logiki oraz danych. Tak naprawdę klasy definiują również nowe przestrzenie nazw, podobnie jak moduły. W porównaniu z innymi jednostkami programów, jakie już widzieliśmy, klasy mają trzy kluczowe i wyróżniające cechy, które sprawiają, że są one bardziej użyteczne przy tworzeniu nowych obiektów.

Wiele instancji

Klasy są tak naprawdę fabrykami generującymi jeden lub większą liczbę obiektów. Za każdym razem, gdy wywołujemy klasę, generujemy nowy obiekt z osobną przestrzenią nazw. Każdy obiekt wygenerowany przez klasę ma dostęp do jej atrybutów *oraz* otrzymuje własną przestrzeń nazw dla swoich danych, które różnią się między obiektami.

Dostosowywanie do własnych potrzeb dzięki dziedziczeniu

Klasy obsługują również koncepcję dziedziczenia z programowania zorientowanego obiektowo. Klasę możemy rozszerzyć, redefiniując jej atrybuty poza samą klasą. Klasy mogą tworzyć hierarchie przestrzeni nazw definiujące zmienne, które będą wykorzystywane przez obiekty utworzone przez klasy tej hierarchii.

Przeciążanie operatorów

Udostępniając specjalne metody protokołów, klasy mogą definiować obiekty odpowiadające na rodzaje operacji, jakie widzieliśmy w przypadku typów wbudowanych. Z obiektów utworzonych za pomocą klas można na przykład tworzyć wycinki, można je poddawać konkatenacji czy indeksowaniu. Python udostępnia punkty zaczepienia, które klasy wykorzystują w celu przejęcia i zaimplementowania dowolnej operacji na typie wbudowanym.

Programowanie zorientowane obiektowo z dystansu

Zanim zobaczymy, jak to wszystko wygląda na poziomie kodu, chciałbym powiedzieć kilka słów o ogólnych koncepcjach leżących u podstaw programowania zorientowanego obiektowo. Jeśli dotychczas nie robiliśmy nic zorientowanego obiektowo, część terminologii wykorzystywana w niniejszym rozdziale może dla nas przy pierwszym podejściu brzmieć niepokojąco. Co więcej, uzasadnienie dla tych pojęć może wydawać się nikłe, dopóki nie mamy okazji zapoznać się z tym, jak programiści stosują te koncepcje w większych systemach. Programowanie zorientowane obiektowo jest nie tylko technologią, ale i pewnym doświadczeniem.

Wyszukiwanie dziedziczenia atrybutów

Dobra wiadomość jest taka, że programowanie zorientowane obiektowo jest o wiele łatwiejsze do zrozumienia i użycia w Pythonie niż w innych językach, takich jak C++ czy Java. Jako język skryptowy z typami dynamicznymi Python usuwa wiele z bałaganu składniowego oraz stopnia skomplikowania obecnych w programowaniu zorientowanym obiektowo w innych narzędziach. Tak naprawdę większość koncepcji programowania zorientowanego obiektowo w Pythonie sprowadza się do jednego wyrażenia.

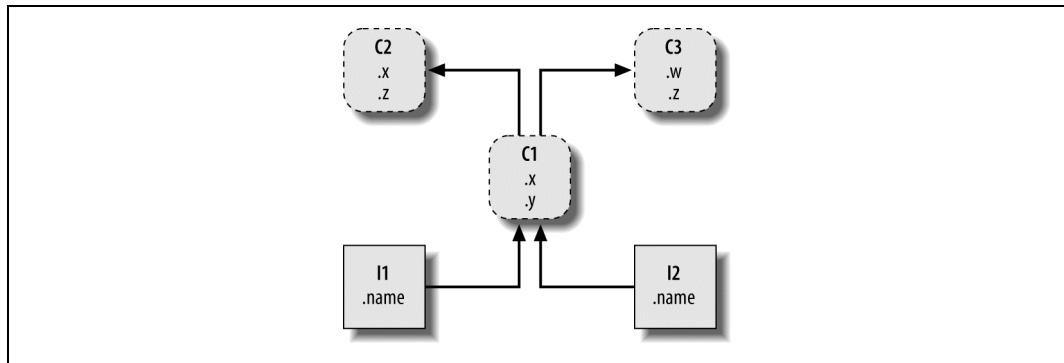
`obiekt.atrybut`

Wyrażenie to wykorzystywaliśmy w całej książce w celu uzyskania dostępu do atrybutów modułów czy wywołania metod obiektów. Kiedy jednak zastosujemy je do obiektu pochodzącego z instrukcji `class`, uruchamia ono w Pythonie *wyszukiwanie*. Przeszukuje drzewo połączonych obiektów, szukając pierwszego wystąpienia atrybutu, jakie może odnaleźć. Kiedy w grę wchodzą klasy, to wyrażenie Pythona można w efekcie przetołożyć na następujące wyrażenie z ludzkiego języka:

Znajdź pierwsze wystąpienie atrybutu, szukając w obiekcie, a następnie we wszystkich klasach powyżej niego, od dołu do góry i od lewej strony do prawej.

Innymi słowy, operacja pobrania atrybutu jest po prostu przeszukiwaniem drzewa. Pojęcie *dziedziczenia* jest stosowane, ponieważ obiekty znajdujące się niżej w drzewie dziedziczą atrybuty dołączone do obiektów umieszczonych w tym drzewie wyżej. W miarę postępowania wyszukiwania od dołu do góry — w pewnym sensie obiekty połączone w drzewo są sumą wszystkich atrybutów zdefiniowanych we wszystkich rodzicach, aż do samej góry drzewa.

W Pythonie wszystko to jest bardzo dosłowne. W kodzie naprawdę budujemy drzewa połączonych obiektów, a Python rzeczywiście wspina się w górę tego drzewa w trakcie wyszukiwania atrybutów w czasie wykonywania — za każdym razem, gdy użyjemy wyrażenia `obiekt.ATTRIB`. By to skonkretyzować, na rysunku 25.1 przedstawiono przykład jednego z takich drzew.



Rysunek 25.1. Drzewo klas z dwoma instancjami na dole (I1 oraz I2), klasą nad nimi (C1) i dwoma klasami nadzędnymi na górze (C2 oraz C3). Wszystkie te obiekty są przestrzeniami nazw (pakietami zmiennych), a dziedziczenie jest po prostu przeszukiwaniem drzewa od dołu do góry w celu odnalezienia najbliższego wystąpienia nazwy atrybutu. Kod sugeruje kształt takich drzew

Na rysunku widoczne jest drzewo z pięcioma obiektami podpisanymi za pomocą zmiennych; do każdego z nich dołączono atrybuty, a drzewo jest gotowe do przeszukania. Drzewo łączy razem trzy obiekty *klas* (owale C1, C2 oraz C3) i dwa obiekty *instancji* (prostokąty I1 oraz I2) w drzewo wyszukiwania dziedziczenia. Warto zauważyć, że w modelu obiektów Pythona klasy oraz instancje z nich wygenerowane są dwoma odrębnymi typami obiektów.

Klasy

Służą jako fabryki instancji. Ich atrybuty udostępniają zachowanie — dane oraz funkcje — dziedziczone przez wszystkie instancje z nich wygenerowane (na przykład funkcję obliczającą wynagrodzenie pracownika na podstawie jego płacy oraz przepracowanych godzin).

Instancje

Reprezentują konkretne elementy w domenie programu. Ich atrybuty zapisują dane różniące się dla określonych obiektów (na przykład numer NIP określonego pracownika).

W kategoriach drzew wyszukiwania instancja dziedziczy atrybuty po klasie, natomiast klasa dziedziczy atrybuty po wszystkich klasach znajdujących się nad nią w drzewie.

Na rysunku 25.1 możemy jeszcze podzielić owale zgodnie z ich pozycją względową w drzewie. Zazwyczaj nazywamy klasy znajdujące się wyżej w drzewie (jak C2 czy C3) *klasami nadzędnymi* lub *nadklasami*. Klasy umieszczone niżej w drzewie (jak C1) są znane jako *klasy podrzędne* lub *podklasy*.¹ Te pojęcia odnoszą się do względnych pozycji oraz ról w drzewie. Klasa nadzędna udostępniają zachowanie współdzielone przez wszystkie ich podklasy, jednak ponieważ wyszukiwanie następuje od dołu do góry, podklasy mogą przesyłać zachowanie zdefiniowane w klasach nadzędnych, redefiniując zmienne z klas nadzędnych niżej w drzewie.

¹ W innej literaturze można się również czasami spotkać z pojęciami *klasa podstawowa* oraz *klasa pochodna*, które służą do opisywania, odpowiednio, klas nadzędnych i podrzędnych.

Ponieważ kilka ostatnich zdań stanowi tak naprawdę sedno dostosowywania oprogramowania do własnych potrzeb w programowaniu zorientowanym obiektowo, rozwińmy nieco przedstawione w nich koncepcje. Założymy, że budujemy drzewo z rysunku 25.1 i w kodzie umieszczaemy poniższe wyrażenie.

I2.w

Ten kod natychmiast wywołuje dziedziczenie. Ponieważ jest to wyrażenie typu `obiekt.atrybut`, wywołuje ono przeszukiwanie drzewa z rysunku 25.1. Python będzie szukał atrybutu `w` w obiekcie I2 oraz powyżej niego. Mówiąc dokładniej, przeszuka połączone obiekty w następującej kolejności:

I2, C1, C2, C3

i zatrzyma się na pierwszym dołączonym atrybutem `w`, jaki potrafi znaleźć (lub zwróci błąd, jeśli nic nie zostanie odnalezione). W tym przypadku atrybut `w` zostanie odszukany dopiero w klasie C3, ponieważ pojawia się jedynie w tym obiekcie. Innymi słowy, I2.w okazuje się C3.w ze względu na wyszukiwanie automatyczne. W kategoriach programowania zorientowanego obiektowo I2 dziedziczy atrybut `w` po C3.

W rezultacie dwie instancje dziedziczą po swoich klasach cztery atrybuty — `w`, `x`, `y` oraz `z`. Inne referencje do atrybutów podążają innymi ścieżkami w drzewie. Na przykład:

- Wyrażenia I1.x oraz I2.x odnajdują atrybut `x` w klasie C1 i zatrzymują się, ponieważ C1 jest niższe od C2.
- Wyrażenia I1.y oraz I2.y odnajdują atrybut `y` w klasie C1, ponieważ jest to jedyne miejsce, w którym się on pojawia.
- Wyrażenia I1.z oraz I2.z odnajdują atrybut `z` w klasie C2, ponieważ C2 znajduje się bardziej na lewo od C3.
- Wyrażenie I2.name odnajduje atrybut `name` bez przechodzenia w góre drzewa.

Warto prześledzić te wyrażenia na drzewie z rysunku 25.1, żeby poczuć, w jaki sposób w Pythonie działa wyszukiwanie dziedziczenia.

Pierwszy element powyższej listy jest chyba najważniejszym, na jaki należy zwrócić uwagę. Ponieważ C1 redefiniuje atrybut `x` niżej w drzewie, w rezultacie *zastępuje* wersję znajdującą się wyżej w C2. Jak zobaczymy za moment, redefiniując i zastępując atrybut, C1 dostosowuje to, co dziedziczy po klasie nadrzędnej, do własnych potrzeb.

Klasy a instancje

Choć w modelu Pythona są to, ściśle rzecz biorąc, dwa osobne typy obiektów, klasy oraz instancje umieszczane w drzewach są prawie identyczne. Głównym celem każdego typu jest służenie jako kolejny rodzaj *przestrzeni nazw* — pakietu zmiennych oraz miejsca, do którego można dołączać atrybuty. Jeśli klasy i instancje przypominają moduły, to dobrze — tak powinno być. Obiekty w drzewach klas zawierają również automatycznie przeszukiwane łącza do innych obiektów przestrzeni nazw, natomiast klasy odpowiadają instrukcjom, nie całym plikom.

Podstawowa różnica pomiędzy klasami i instancjami polega na tym, że klasy są swego rodzaju *fabryką* generującą instancje. W prawdziwej aplikacji możemy na przykład mieć klasę Employee definiującą, co oznacza bycie pracownikiem. Z tej klasy generujemy konkretne instancje

pracowników. Jest to kolejna różnica pomiędzy klasami a modułami — w pamięci zawsze mamy tylko jedną instancję określonego modułu (dlatego do pobrania aktualnego kodu trzeba moduł przeładować), natomiast w przypadku klas możemy utworzyć dowolną liczbę instancji.

Z punktu widzenia działania do klas zazwyczaj dołączone są funkcje (na przykład `computeSalary` obliczająca wynagrodzenie), a instancje będą zawierały bardziej podstawowe elementy danych wykorzystywane przez funkcje klas (na przykład zmiennej `hoursWorked` wskazującą liczbę przepracowanych godzin). Tak naprawdę model zorientowany obiektowo nie różni się szczególnie od klasycznego modelu przetwarzania danych z *programami i rekordami*. W przypadku programowania zorientowanego obiektowo instancje są jak rekordy z „danymi”, a klasy są „programami” służącymi do przetwarzania tych rekordów. W programowaniu zorientowanym obiektowo mamy jednak dodatkowo pojęcie hierarchii dziedziczenia, która obsługuje możliwość dostosowania programu do własnych potrzeb lepiej od poprzednich modeli.

Wywołania metod klas

W poprzednim podrozdziale widzieliśmy, jak referencja do atrybutu `I2.w` w naszym przykładowym drzewie klas została przetłumaczona na `C3.w` za pomocą procedury wyszukiwania dziedziczenia w Pythonie. Chyba również ważne do zrozumienia jak dziedziczenie atrybutów jest to, co dzieje się, kiedy próbujemy wywołać metody klas (czyli funkcje dołączone do klas jako atrybuty).

Jeśli referencja `I2.w` jest wywołaniem *funkcji*, oznacza tak naprawdę: „Wywołaj funkcję `C3.w`, tak by przetworzyła ona `I2`”. W rezultacie Python automatycznie odwzorowuje wywołanie `I2.w()` na wywołanie `C3.w(I2)`, przekazując instancję jako pierwszy argument do dziedziczonej funkcji.

Tak naprawdę za każdym razem, gdy w ten sposób wywołujemy funkcję dołączoną do klasy, zasugerowana zostaje instancja klasy. Ten domniemany podmiot czy kontekst jest częścią powodu, dla którego model ten nazywamy *zorientowanym obiektowo* — kiedy operacja jest wykonywana, zawsze istnieje obiekt podmiotu. W bardziej realistycznym przykładzie możemy wywołać przyznającą podwyżkę metodę o nazwie `giveRaise`, dołączoną jako atrybut do klasy `Employee`. Takie wywołanie nie ma znaczenia, o ile nie zostanie do niego dodany pracownik, któremu powinna zostać przyznana podwyżka.

Jak zobaczymy później, Python przekazuje domniemaną instancję do specjalnego pierwszego argumentu metody, nazywanego konwencjonalnie `self`. Jak się niebawem dowiemy, metody można wywołać albo przez instancję (na przykład `aleksander.giveRaise()`), albo przez klasę (na przykład `Employee.giveRaise(aleksander)`), a obie formy będą w naszych skryptach pełnić określone role. By zobaczyć, w jaki sposób metody otrzymują podmioty, musimy przejść do twożenia kodu.

Tworzenie drzew klas

Choć mówimy tutaj w kategoriach abstrakcyjnych, za tymi wszystkimi koncepcjami stoi niewielki kod. Drzewa oraz ich obiekty konstruujemy za pomocą instrukcji `class` oraz wywołań klas, które bardziej szczegółowo omówimy później. W skrócie:

- Każda instrukcja `class` generuje nowy obiekt klasy.
- Za każdym razem gdy klasa jest wywoływana, generuje ona nowy obiekt instancji.
- Instancje są automatycznie połączone z klasami, przez które zostały utworzone.
- Klasa są połączone ze swoimi klasami nadzędnymi dzięki podaniu ich w nawiasach w wierszu nagłówka instrukcji `class`. Uporządkowanie od lewej do prawej strony podaje kolejność w drzewie.

By na przykład zbudować drzewo z rysunku 25.1, moglibyśmy wykonać kod Pythona w poniższej formie (pominąłem szczegóły instrukcji `class`).

```
class C2: ...                                # Utworzenie obiektu klasy (owalu)
class C3: ...                                # Połączona z klasami nadzędnymi
class C1(C2, C3): ...                         # Utworzenie obiektów instancji (prostokątów)
                                                # Połączenie z ich klasami
```

W powyższym kodzie budujemy trzy obiekty klas, wykonując trzy instrukcje `class`, a także tworzymy dwa obiekty instancji, wywołując dwukrotnie `C1`, tak, jakby była to funkcja. Instancje pamiętają, z których klas zostały utworzone, natomiast klasa `C1` pamięta swoje klasę nadzędne.

Z technicznego punktu widzenia przykład ten wykorzystuje coś o nazwie *dziedziczenia wielokrotnego* (ang. *multiple inheritance*), co oznacza po prostu, że klasa może mieć więcej niż jedną klasę nadzelną w drzewie klas. W Pythonie, jeśli w instrukcji `class` w nawiasach wymieniona jest większa liczba klas nadzędnych (jak w przypadku `C1` w kodzie wyżej), ich uporządkowanie od lewej do prawej strony określa kolejność, w jakiej klasy te będą przeszukiwane pod kątem atrybutów.

Ze względu na sposób działania wyszukiwania dziedziczenia obiekt, do którego dołączymy atrybut, okazuje się mieć kluczowe znaczenie — określa zakres zmiennej. Atrybuty dołączone do instancji odnoszą się tylko do tych instancji, jednak atrybuty dołączone do klas są współdzielone przez wszystkie ich podklasy oraz instancje. Później szczegółowo przestudujemy kod załączający atrybuty do takich obiektów. Dowiemy się, że:

- atrybuty są zazwyczaj dołączane do klas przez przypisanie wykonane wewnątrz instrukcji `class` i nie zagnieżdżone dodatkowo wewnątrz instrukcji `def` funkcji,
- atrybuty są zazwyczaj dołączane do instancji przez przypisanie do specjalnego argumentu przekazanego do funkcji wewnątrz klasy i noszącego nazwę `self`.

Klasy udostępniają na przykład zachowanie swoim instancjom za pomocą funkcji utworzonych przez instrukcje `def` umieszczone w instrukcjach `class`. Ponieważ takie zagnieżdżone instrukcje `def` przypisują nazwy wewnątrz klasy, w rezultacie dołączają do obiektu klasy atrybuty, które zostają odziedziczone przez wszystkie instancje i klasę podrzędne.

```
class C1(C2, C3):
    def setname(self, who):
        self.name = who

I1 = C1()                                     # Utworzenie i połączenie klasy C1
I2 = C1()                                     # Przypisanie nazwy: C1.setname
                                                # self to albo I1, albo I2

I1.setname('amadeusz')                         # Utworzenie dwóch instancji
I2.setname('aleksander')                       # Ustawia I1.name na 'amadeusz'
                                                # Ustawia I2.name na 'aleksander'
print(I1.name)                                 # Wyświetla 'amadeusz'
```

W instrukcjach `def` w tym kontekście nie ma nic unikalnego z punktu widzenia składni. Z punktu widzenia działania, kiedy instrukcja `def` pojawia się wewnątrz `class`, zazwyczaj znana jest jako *metoda* i automatycznie otrzymuje specjalny pierwszy argument — nazywany zgodnie z konwencją `self` — udostępniający uchwyt z powrotem do przetwarzanej instancji.²

Ponieważ klasy są fabrykami tworzącymi wiele instancji, ich metody zazwyczaj przechodzą automatycznie przekazany argument `self` za każdym razem, gdy potrzebują pobrać lub ustawić atrybuty określonej instancji przetwarzanej przez wywołanie metody. W poprzednim kodzie `self` wykorzystano do przechowania nazwy w jednej z dwóch instancji.

Podobnie do prostych zmiennych, atrybuty klas oraz instancji nie są deklarowane z wyprzedzeniem, a zamiast tego zaczynają istnieć w momencie pierwszego przypisania do nich wartości. Kiedy metoda przypisuje coś do atrybutu `self`, tworzy bądź modyfikuje atrybut w instancji na dole drzewa klas (to znaczy w jednym z prostokątów), ponieważ `self` automatycznie odnosi się do przetwarzanej instancji.

Tak naprawdę, ponieważ wszystkie obiekty drzewa klas są po prostu przestrzeniami nazw obiektów, możemy pobrać lub ustawić każdy z ich atrybutów, przechodząc odpowiednie nazwy. Kod `C1.setname` jest tak samo poprawny jak kod `I1.setname`, o ile zmienne `C1` i `I1` są w zakresach naszego kodu.

W obecnej formie nasza klasa `C1` nie dodaje atrybutu `name` do instancji, dopóki nie zostanie wywołana metoda `setname`. Tak naprawdę odniesienie się do `I1.name` przed wywołaniem metody `I1.setname` powoduje zwrócenie błędu niezdefiniowanej zmiennej. Jeśli chcemy zagwarantować w klasie, że atrybut taki, jak `name`, jest zawsze ustawiony w instancjach, zazwyczaj wypełniamy ten atrybut w czasie konstrukcji, jak w poniższym przykładzie.

```
class C1(C2, C3):
    def __init__(self, who):
        self.name = who
    # Ustawia name przy utworzeniu
    # self to albo I1, albo I2
I1 = C1('amadeusz')
I2 = C1('aleksander')
print(I1.name)
# Ustawia I1.name na 'amadeusz'
# Ustawia I2.name na 'aleksander'
# Wyświetla 'amadeusz'
```

Po zapisaniu tego w kodzie i odziedziczeniu Python automatycznie wywołuje metodę o nazwie `__init__` za każdym razem, gdy generowana jest instancja klasy. Nowa instancja przekazywana jest do argumentu `self` metody `__init__` tak, jak zawsze, a wartości podane w nawiasach w wywołaniu klasy trafiają do argumentów od drugiego w góre. W rezultacie inicjalizujemy instancję w czasie, gdy jest ona tworzona, bez wymagania dodatkowych wywołań metod.

Metoda `__init__` znana jest jako *konstruktor* ze względu na czas wykonywania. To najczęściej wykorzystywany reprezentant większej klasy metod znanych jako *metody przeciążania operatorów*, które omówimy bardziej szczegółowo w kolejnych rozdziałach. Takie metody są dziedziczone w drzewach klas tak, jak zawsze i na początku oraz końcu mają podwójne znaki `_`, co pozwala je odróżnić od innych metod. Python wykonuje je automatycznie, kiedy instancje je obsługujące pojawiają się w odpowiadających im operacjach. Są w większości alternatywą dla wykonywania prostych wywołań metod. Są również opcjonalne — jeśli się je pominie, operacje te nie są obsługiwane.

² Osoby, które kiedykolwiek używały języków C++ lub Java, rozpoznają, że `self` w Pythonie jest tym samym, co wskaźnik `this`, jednak `self` z Pythona jest zawsze jawnie, tak by dostęp do atrybutów był bardziej oczywisty.

Żeby na przykład zaimplementować część wspólną zbiorów, klasa może albo udostępniać metodę o nazwie `intersect`, albo przeciągać operator wyrażenia `&` w taki sposób, by udostępnić wymaganą logikę, zapisując metodę o nazwie `_and_`. Ponieważ rozwiązywanie z operatorem sprawia, że instancje wyglądają i zachowują się bardziej jak typy wbudowane, pozwala to klasom udostępniać konsekwentny i naturalny interfejs, a także zapewnia spójność z kodem oczekującym typu wbudowanego.

Programowanie zorientowane obiektywne oparte jest na ponownym wykorzystaniu kodu

I to, wraz z kilkoma szczegółami składniowymi, składa się na większość zagadnień związanych z programowaniem zorientowanym obiektywne w Pythonie. Oczywiście w grę wchodzi coś więcej niż tylko dziedziczenie. Przeciążanie operatorów jest o wiele bardziej ogólne, niż to dotychczas opisałem — klasy mogą również udostępniać swoje własne implementacje operacji takich, jak indeksowanie, pobieranie atrybutów czy wyświetlanie. Przede wszystkim jednak programowanie zorientowane obiektywne polega na wyszukiwaniu atrybutów w drzewach.

Dlaczego mielibyśmy interesować się budowaniem i przeszukiwaniem drzew obiektów? Choć zrozumienie tego wymaga nieco praktyki, to dobrze wykorzystywane klasy wspomagają *ponowne użycie kodu* w sposób niemożliwy do uzyskania za pomocą innych komponentów programu Pythona. Dzięki klasom kod tworzy się, dostosowując istniejące oprogramowanie do własnych potrzeb, zamiast albo modyfikować je w miejscu, albo rozpoczynać każdy nowy projekt od podstaw.

Na poziomie podstawowym klasy są tak naprawdę pakietami funkcji i innych zmiennych, podobnie jak moduły. Automatyczne wyszukiwanie dziedziczenia atrybutów będące częścią klas umożliwia dostosowanie oprogramowania do własnych potrzeb wykraczające poza to, co możemy uzyskać za pomocą modułów oraz funkcji. Co więcej, klasy stanowią naturalną strukturę dla kodu, lokalizując logikę oraz zmienne, co pomaga w debugowaniu kodu.

Przykładowo, ponieważ metody są po prostu funkcjami ze specjalnym pierwszym argumentem, możemy naśladować ich zachowanie, ręcznie przekazując obiekty, jakie mają być przetwarzane, do prostych funkcji. Udział metod w dziedziczeniu klas pozwala nam jednak w naturalny sposób dostosowywać istniejące oprogramowanie do własnej sytuacji za pomocą tworzenia klas podrzędnych z nowymi definicjami metod zamiast modyfikowania istniejącego kodu w miejscu. Takie coś w przypadku modułów oraz funkcji nie jest możliwe.

Jako przykład rozważmy sytuację, w której otrzymaliśmy zadanie zaimplementowania aplikacji działającej na bazie danych pracowników. Jako programista Pythona zorientowany obiektywne możemy rozpocząć od napisania ogólnej klasy nadzędnej definiującej domyślne zachowania wspólne dla wszystkich typów pracowników naszej organizacji.

```
class Employee:                                # Ogólna klasa nadzędna
    def computeSalary(self): ...
    def giveRaise(self): ...
    def promote(self): ...
    def retire(self): ...                         # Wspólne lub domyślne zachowanie
```

Po zapisaniu tego ogólnego zachowania w kodzie możemy je teraz wyspecjalizować dla każdego konkretnego typu pracownika w celu odzwierciedlenia tego, jak poszczególne typy odbiegają od normy. Możemy zatem utworzyć klasy podrzędne dostosowujące te fragmenty zachowania,

które są różne dla różnych typów pracowników. Pozostałe wzorce zachowania pracowników zostaną odziedziczone po klasie ogólnej. Jeśli na przykład inżynierowie mają unikalne reguły obliczania wynagrodzenia (na przykład nieopierające się na stawce godzinowej), wystarczy zastąpić jedną metodę w klasie podzielonej.

```
class Engineer(Employee):  
    def computeSalary(self): ...
```

Wyspecjalizowana klasa podzielona
Coś własnego

Ponieważ ta wersja metody `computeSalary` pojawia się niżej w drzewie klas, zastąpi (przesłoni) ogólną wersję z klasy `Employee`. Możemy następnie utworzyć podobne instancje dla klas pracowników, do których należą rzeczywiste osoby, tak by otrzymać poprawne zachowanie.

```
amadeusz = Employee()  
aleksander = Engineer()
```

Zachowanie domyślne
Własne obliczenie wynagrodzenia

Warto zauważyć, że możemy utworzyć instancje dowolnej klasy drzewa — nie tylko klas znajdujących się na dole. Klasa, której instancję tworzymy, określa poziom, na jakim rozpocznie się wyszukiwanie atrybutów. W rezultacie dwa obiekty instancji mogą zostać osadzone w jakimś większym obiekcie-pojemniku (na przykład liście lub instancji innej klasy) reprezentującym dział czy firmę, wykorzystując do tego koncepcję kompozycji wspomnianą na początku rozdziału.

Kiedy później będziemy pytać o zarobki tych pracowników, zostaną one obliczone zgodnie z klasami, w których utworzone zostały obiekty — ze względu na zasadę wyszukiwania dziedziczenia.³

```
company = [amadeusz, aleksander]  
for emp in company:  
    print(emp.computeSalary())
```

Obiekt kompozytowy
Wykonanie wersji tego obiektu

Jest to kolejny przykład koncepcji *polimorfizmu* wprowadzonej w rozdziale 4. i wspomnianej raz jeszcze w rozdziale 16. Warto przypomnieć, iż polimorfizm oznacza, że znaczenie operacji uzależnione jest od obiektu, na którym się ona odbywa. Tutaj metoda `computeSalary` przed wywołaniem lokalizowana jest za pomocą wyszukiwania dziedziczenia w każdym obiekcie. W innych aplikacjach polimorfizm może również służyć do ukrywania (*hermetyzacji*) różnic między interfejsami. Przykładowo program przetwarzający strumienie danych może zostać zapisany w kodzie w taki sposób, by oczekiwali obiektów z metodami wejścia i wyjścia, bez troszczenia się o to, co te metody tak naprawdę robią.

```
def processor(reader, converter, writer):  
    while 1:  
        data = reader.read()  
        if not data: break  
        data = converter(data)  
        writer.write(data)
```

Przekazując instancje klas podzielonych specjalizujących wymagane interfejsy metod `read` oraz `write` dla różnych źródeł danych, możemy wykorzystać funkcję `processor` ponownie dla dowolnego źródła danych, z jakiego będziemy musieli korzystać — teraz i w przyszłości.

³ Warto zauważyć, że lista `company` w tym przykładzie mogłyby zostać przechowana w pliku za pomocą serializacji z użyciem modułu `pickle` Pythona, wprowadzonego w rozdziale 9., w którym zapoznawaliśmy się z plikami. W ten sposób moglibyśmy uzyskać trwałą bazę danych pracowników. Python zawiera również moduł `shelve`, który pozwoliłby nam przechować poddaną serializacji reprezentację instancji klas w systemie plików dostępnych po kluczu. System zewnętrzny ZODB dostępny na licencji open source robi to samo, jednak ma lepszą obsługę zorientowanych obiektowo baz danych jakości produkcyjnej.

```

class Reader:
    def read(self): ...
    def other(self): ...
class FileReader(Reader):
    def read(self): ... # Domyślne zachowanie oraz narzędzia
class SocketReader(Reader):
    def read(self): ... # Odczytanie z lokalnego pliku
...
processor(FileReader(...), Converter, FileWriter(...))
processor(SocketReader(...), Converter, TapeWriter(...))
processor(FtpReader(...), Converter, XmlWriter(...))

```

Co więcej, ponieważ wewnętrzne implementacje tych metod `read` oraz `write` zostały umieszczone w osobnych lokalizacjach, można je modyfikować bez wpływu na kod je wykorzystujący. Tak naprawdę sama funkcja `processor` może być klasą, co pozwoli na wypełnienie logiki konwersji w `converter` za pomocą dziedziczenia, a także umożliwi osadzenie kodu odczytującego i zapisującego za pomocą kompozycji (w dalszej części książki zobaczymy, jak to działa).

Kiedy przyzwyczajmy się do programowania w ten sposób (czyli dostosowywania oprogramowania do własnych potrzeb), zobaczymy, że gdy przyjdzie nam napisać nowy program, okaże się, iż duża część pracy została już wykonana — nasze zadanie będzie się w dużej mierze sprowadzało do połączenia ze sobą istniejących klas nadzędnych implementujących zachowanie wymagane przez program. Przykładowo ktoś mógł już napisać klasy `Employee`, `Reader` oraz `Writer` z tego przykładu w celu zastosowania ich w zupełnie innym programie. Jeśli tak, cały kod utworzony przez tę osobę otrzymujemy „za darmo”.

W wielu dziedzinach zastosowania można pobrać lub zakupić kolekcje klas nadzędnych, znanych jako *platformy* (ang. *framework*) i implementujących często wykonywane zadania programistyczne w postaci klas gotowych do dołączenia do naszych aplikacji. Takie platformy mogą udostępniać interfejsy do baz danych, protokoły testowania czy zestawy narzędzi graficznego interfejsu użytkownika. Dzięki takim platformom wystarczy często utworzyć klasę podrzędną dodającą oczekiwany metodę czy nawet kilka metod, natomiast większość pracy zostanie wykonana przez klasy nadzędne znajdujące się wyżej w drzewie. Programowanie w świecie zorientowanym obiektowo polega na łączeniu i specjalizacji gotowego i sprawdzonego kodu za pomocą pisania własnych klas podrzędnych.

Oczywiście nauczenie się, jak można wykorzystywać klasy w taki sposób w celu uzyskania podobnej, zorientowanej obiektowo utopii, zajmuje trochę czasu. W praktyce programowanie zorientowane obiektowo obejmuje również sporą dawkę projektowania wymaganą do pełnego zrozumienia zalet ponownego wykorzystywania kodu dzięki klasom. W tym celu programiści zaczęli katalogować najczęściej wykorzystywane struktury programowania zorientowanego obiektowo, znane jako *wzorce projektowe* (ang. *design patterns*), które mogą pomóc nam w projektowaniu. Sam kod, jaki piszemy w Pythonie w programowaniu zorientowanym obiektowo, jest tak prosty, że nie powinien stanowić dla nas żadnej dodatkowej przeszkody. Żeby to jednak zobaczyć, będziemy musieli przejść do kolejnego rozdziału.

Podsumowanie rozdziału

W niniejszym rozdziale przyjrzaliśmy się klasom oraz programowaniu zorientowanemu obiektowo na poziomie abstrakcji, próbując zyskać szerszą perspektywę, zanim przejdziemy do szczegółów składni. Jak widzieliśmy, programowanie zorientowane obiektowo w dużej mierze polega

na wyszukiwaniu atrybutów w drzewach połączonych obiektów — nazywamy to wyszukiwaniem dziedziczenia. Obiekty znajdujące się na dole drzewa dziedziczą atrybuty po obiektach znajdujących się wyżej — pozwala nam to programować za pomocą dostosowywania kodu do własnych potrzeb, a nie modyfikowania go czy rozpoczęnięcia od nowa. Kiedy się go dobrze wykorzystuje, taki model programowania może radykalnie skrócić czas poświęcony na tworzenie kodu.

W kolejnym rozdziale zaczniemy wypełniać brakujące fragmenty szczegółów dotyczących kodu, stojących za zarysowaną tutaj perspektywą. W miarę zagłębiania się w klasy Pythona należy pamiętać o tym, że model programowania zorientowanego obiektowo w Pythonie jest bardzo prosty — jak powiedzieliśmy wcześniej, tak naprawdę sprowadza się do wyszukiwania atrybutów w drzewach obiektów. Zanim jednak przejdziemy dalej, czas na krótki quiz sprawdzający przedstawione dotychczas informacje.

Sprawdź swoją wiedzę — quiz

1. Co jest głównym celem programowania zorientowanego obiektowo w Pythonie?
2. Gdzie wyszukiwanie dziedziczenia szuka atrybutów?
3. Jaka jest różnica pomiędzy obiektem klasy a obiektem instancji?
4. Dlaczego pierwszy argument funkcji metody klasy jest specjalny?
5. Do czego wykorzystywana jest metoda `__init__`?
6. W jaki sposób tworzy sięinstancję klasy?
7. W jaki sposób tworzy się klasę?
8. W jaki sposób określa się klasę nadzczną jakiejś klasy?

Sprawdź swoją wiedzę — odpowiedzi

1. Programowanie zorientowane obiektowo polega na ponownym wykorzystywaniu kodu — kod poddawany jest faktoryzacji w celu zminimalizowania jego powtarzalności, a programuje się, dostosowując istniejący kod do własnych potrzeb, zamiast modyfikować go w miejscu czy pisać od nowa.
2. Wyszukiwanie dziedziczenia szuka atrybutu najpierw w obiekcie instancji, później w klasie, w której powstała ta instancja, a następnie we wszystkich klasach nadzędnych, przechodząc domyślnie od dołu do góry drzewa obiektów i od lewej strony do prawej. Wyszukiwanie zatrzymuje się w pierwszym miejscu, w jakim odnaleziony zostaje atrybut. Ponieważ najniższa wersja zmiennej znaleziona po drodze wygrywa, hierarchie klas w naturalny sposób obsługują dostosowywanie kodu do własnych potrzeb za pomocą rozszerzania go.
3. Obiekty klas oraz instancji są przestrzeniami nazw (pakietami zmiennych, które dostępne są jako atrybuty). Podstawowa różnica między nimi polega na tym, że klasy są rodzajem fabryki służącej do tworzenia wielu instancji. Klasa obsługuje również dziedziczone przez instancje metody przeciążania operatorów i traktują funkcje zagnieżdżone wewnętrz nich jako specjalne metody służące do przetwarzania instancji.

4. Pierwszy argument w funkcji metody klasy jest specjalny, ponieważ zawsze otrzymuje on obiekt instancji będący domniemanym podmiotem wywołania metody. Zazwyczaj nosi on nazwę `self`, zgodnie z przyjętą konwencją. Ponieważ funkcje metod zawsze zawierają ten kontekst domniemanego podmiotu, mówimy, że są „zorientowane obiektywnie”, czyli zaprojektowane tak, by przetwarzać lub modyfikować obiekty.
5. Jeśli metoda `__init__` jest zapisana w kodzie lub dziedziczona w klasie, Python wywołuje ją automatycznie za każdym razem, gdy tworzona jest instancja tej klasy. Jest ona znana jako metoda konstruktora. Jest przekazywana do każdej nowej instancji w sposób niejawny, wraz z dowolnymi argumentami przekazanymi do nazwy klasy w sposób jawny. Metoda `__init__` jest również najczęściej wykorzystywana metodą przeciążania operatorów. Jeśli nie występuje ona w klasie, instancje zaczynają swoje istnienie jako puste przestrzenie nazw.
6. Instancję klasy tworzy się, wywołując nazwę klasy, tak jakby była ona funkcją. Argumenty przekazane do nazwy klasy pojawiają się jako argumenty na pozycjach drugiej i kolejnych w metodzie konstruktora `__init__`. Nowa instancja pamięta klasę, z której została utworzona, ze względu na dziedziczenie.
7. Klasę tworzy się, wykonując instrukcję `class`. Podobnie do definicji funkcji, instrukcje te normalnie wykonywane są, kiedy importowany jest zawierający je plik modułu (więcej informacji na ten temat w kolejnym rozdziale).
8. Klasy nadzędne określa się, wymieniając je w nawiasach w instrukcji `class`, zaraz po nazwie nowej klasy. Uporządkowanie klas w nawiasach od lewej do prawej strony określa kolejność wyszukiwania dziedziczenia w drzewie klas.

Podstawy tworzenia klas

Skoro omówiliśmy już abstrakcyjną część programowania zorientowanego obiektowo, czas zaprezentować, jak przekłada się ona na prawdziwy kod. W niniejszym rozdziale zaczniemy omawiać szczegóły składni stojącej za modelem klas w Pythonie.

Osobom, które nigdy w przeszłości nie spotkały się z programowaniem zorientowanym obiektowo, klasy mogą się wydawać nieco skomplikowane, jeśli się je próbuje przyswoić w jednym kawałku. By tworzenie klas było łatwiejsze do zrozumienia, rozpoczęniemy nasze szczegółowe omówienie programowania zorientowanego obiektowo od przyjrzenia się działaniu podstawowych klas w tym rozdziale. Przedstawione tutaj podstawy rozszerzymy w kolejnych rozdziałach tej części książki. W swojej podstawowej postaci klasy Pythona są łatwe do zrozumienia.

Tak naprawdę klasy mają trzy podstawowe cechy wyróżniające. Na najbardziej podstawowym poziomie są po prostu przestrzeniami nazw, podobnie jak moduły omówione w piątej części książki. Jednak w przeciwieństwie do modułów klasy obsługują również generowanie wielu obiektów, dziedziczenie przestrzeni nazw oraz przeciążanie operatorów. Zaczniemy nasze omawianie klas od przyjrzenia się każdej z tych trzech cech wyróżniających.

Klasy generują większą liczbę obiektów instancji

By zrozumieć, jak działa koncepcja większej liczby obiektów, musimy najpierw pojąć, że w modelu programowania zorientowanego obiektowo w Pythonie istnieją dwa rodzaje obiektów — obiekty *klas* oraz obiekty *instancji*. Obiekty klas udostępniają domyślne zachowanie i służą jako fabryki obiektów instancji. Obiekty instancji są prawdziwymi obiektami产生的 przez programy — każdy jest osobną przestrzenią nazw, jednak jednocześnie dziedziczy (to znaczy automatycznie ma dostęp) zmienne z klasy, w której został utworzony. Obiekty klas pochodzą z instrukcji, a obiekty instancji — z wywołań. Za każdym razem gdy wywołujemy klasę, otrzymujemy nową instancję tej klasy.

Taka koncepcja generowania obiektów jest całkowicie różna od wszystkich konstrukcji programu, które dotychczas widzieliśmy. W rezultacie klasy są tak naprawdę *fabrykami* generującymi wiele instancji. W przeciwieństwie do tego tylko jedna kopia modułu jest importowana do jednego programu (jednym z powodów wywoływanego funkcji `imp.reload` jest uaktualnienie pojedynczego obiektu modułu, tak by po wprowadzeniu modyfikacji zmiany te były odzwierciedlane w obiekcie modułu).

Poniżej znajduje się krótkie streszczenie podstaw programowania zorientowanego obiektowo w Pythonie. Jak zobaczymy, klasy Pythona w wielu aspektach są podobne do instrukcji `def` oraz modułów, jednak mogą się dość znacznie różnić od tego, co możemy znać z innych języków programowania.

Obiekty klas udostępniają zachowania domyślne

Kiedy wykonujemy instrukcję `class`, otrzymujemy obiekt klasy. Poniżej znajduje się lista najważniejszych właściwości klas Pythona.

- **Instrukcja `class` tworzy obiekt klasy i przypisuje go do nazwy.** Podobnie jak instrukcja funkcji `def`, `class` jest w Pythonie instrukcją *wykonywalną*. Kiedy interpreter Pythona dotrze do niej i wykona ją, generuje nowy obiekt klasy i przypisuje go do nazwy podanej w nagłówku instrukcji. Podobnie jak `def`, instrukcja `class` zazwyczaj wykonywana jest przy pierwszym importowaniu pliku ją zawierającego.
- **Przypisania wewnętrz instrukcji `class` tworzą atrybuty klasy.** Podobnie jak w plikach modułów, przypisania na najwyższym poziomie wewnętrz instrukcji `class` (nieosadzone wewnętrz `def`) generują atrybuty w obiekcie klasy. Z technicznego punktu widzenia zakres instrukcji `class` staje się przestrzenią nazw atrybutów obiektu, podobnie jak ma to miejsce w przypadku zakresu globalnego modułu. Po wykonaniu instrukcji `class` dostęp do atrybutów można uzyskać za pomocą składni kwalifikującej `obiekt.atrybut`.
- **Atrybuty klasy udostępniają stan obiektu oraz jego zachowanie.** Atrybuty obiektu klasy rejestrują informacje o stanie oraz zachowanie współdzielone przez wszystkie instancje utworzone z tej klasy. Instrukcje `def` funkcji zagnieżdżone wewnętrz instrukcji `class` generują *metody*, które przetwarzają instancje.

Obiekty instancji są rzeczywistymi elementami

Kiedy wywołujemy obiekt klasy, otrzymujemy obiekt instancji. Poniżej znajduje się przegląd najważniejszych właściwości instancji klas.

- **Wywołanie obiektu klasy w sposób podobny do wywołania funkcji tworzy nowy obiekt instancji.** Za każdym wywołaniem klasy tworzony i zwracany jest nowy obiekt instancji. Instancje reprezentują rzeczywiste elementy z domeną naszego programu.
- **Każdy obiekt instancji dziedziczy atrybuty klasy oraz otrzymuje własną przestrzeń nazw.** Obiekty instancji utworzone przez klasy są nowymi przestrzeniami nazw. Na początku są puste, ale dziedziczą atrybuty znajdujące się w obiektach klas, z których zostały wygenerowane.
- **Przypisania do atrybutów `self` w metodach tworzą atrybuty instancji.** Wewnętrz funkcji metod klasy pierwszy argument (o zgodnej z konwencją nazwie `self`) odnosi się do przetwarzanego obiektu instancji. Przypisania do atrybutów `self` tworzą lub modyfikują dane w instancji, a nie w klasie.

Pierwszy przykład

Zajmijmy się teraz pierwszym prawdziwym przykładem, który pokaże nam, jak koncepcje te działają w praktyce. Na początek zdefiniujmy klasę o nazwie `FirstClass`, wykonując instrukcję `class` Pythona w sesji interaktywnej.

```
>>> class FirstClass:
...     def setdata(self, value):
...         self.data = value
...     def display(self):
...         print(self.data)
```

Zdefiniowanie obiektu klasy
Zdefiniowanie metod klasy
self to instancja
self.data: dla instancji

Pracujemy tutaj w sesji interaktywnej, jednak zazwyczaj taką instrukcję wykonuje się, kiedy importowany jest plik, w którym została ona zapisana. Podobnie do funkcji utworzonych za pomocą instrukcji `def`, klasa ta nie istnieje, dopóki Python nie dotrze do tej instrukcji i nie wykona jej.

Tak jak wszystkie instrukcje złożone, `class` rozpoczyna się od wiersza nagłówka podającego jej nazwę, po której umieszczone jest ciało składające się z jednej lub większej liczby zagnieżdżonych i w zwykły sposób wciętych instrukcji. Tutaj zagnieżdżone instrukcje to `def`; definiują one funkcje implementujące zachowanie, jakie klasa chce eksportować.

Jak wiemy z czwartej części książki, instrukcja `def` jest tak naprawdę przypisaniem. W kodzie powyżej przypisuje obiekty funkcji do nazw `setdata` oraz `display` w zakresie instrukcji `class`, przez co generuje atrybuty dołączone do klas, noszące nazwy `FirstClass.setdata` oraz `FirstClass.display`. Tak naprawdę każda zmienna przypisana na najwyższym poziomie zagnieżdżonego bloku klasy staje się atrybutem tej klasy.

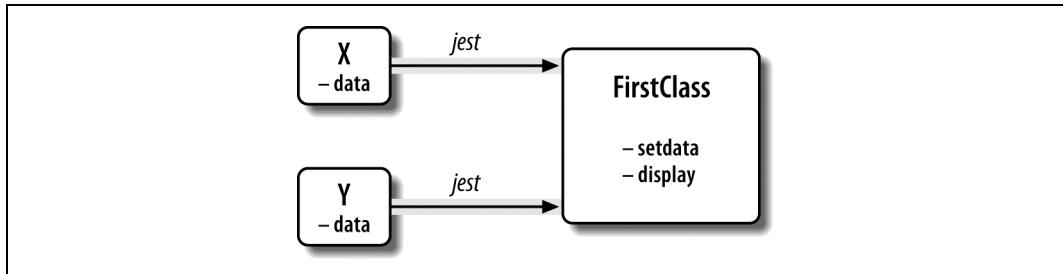
Funkcje znajdujące się wewnętrz klasy zazwyczaj nazywane są *metodami*. W kodzie zapisywane są za pomocą normalnych instrukcji `def` i obsługują wszystko, czego dotychczas nauczyliśmy się o funkcjach (mogą na przykład mieć wartości domyślne czy zwracać wartości). W funkcji metody pierwszy argument automatycznie po wywołaniu otrzymuje domniemany obiekt instancji — podmiot wywołania. By zobaczyć, jak to działa, musimy utworzyć kilka instancji.

```
>>> x = FirstClass()                                # Utworzenie dwóch instancji
>>> y = FirstClass()                                # Każda jest nową przestrzenią nazw
```

Wywołując klasę w ten sposób (należy zwrócić uwagę na nawiasy), generujemy obiekty instancji, które tak naprawdę są po prostu przestrzeniami nazw z dostępem do atrybutów klasy. Mówiąc dokładnie, w tym momencie mamy do dyspozycji trzy obiekty — dwie instancje oraz klasę. Tak naprawdę mamy trzy połączone przestrzenie nazw naszkicowane na rysunku 26.1. W terminologii programowania zorientowanego obiektowo mówimy, że `x` „jest” (ang. *is-a*) `FirstClass`, podobnie jak `y`.

Dwie instancje są na początku puste, jednak mają łączę z powrotem do klasy, z której zostały wygenerowane. Jeśli zapiszemy instancję wraz z nazwą atrybutu znajdującego się w obiekcie klasy w składni kwalifikującej (z kropką), Python pobierze tę nazwę z klasy za pomocą wyszukiwania dziedziczenia (o ile nie znajduje się ona również w instancji).

```
>>> x.setdata("Król Artur")                         # Wywołanie metod: self to x
>>> y.setdata(3.14159)                             # Wykonuje FirstClass.setdata(y, 3.14159)
```



Rysunek 26.1. Klassy oraz instancje są połączonymi obiektami przestrzeni nazw w drzewie klas, które jest przeszukiwane przez dziedziczenie. Atrybut „data” zostaje tutaj odnaleziony w instancjach, natomiast „setdata” oraz „display” w klasie znajdującej się ponad nimi

Ani x, ani y nie mają własnego atrybutu `setdata`, dlatego w celu odnalezienia go Python podąża łączem z instancji do klasy. I to mniej więcej wszystko, co można powiedzieć o dziedziczeniu — odbywa się ono w czasie kwalifikowania atrybutów i obejmuje jedynie wyszukiwanie nazw w połączonych obiektach (poprzez podążanie łączami widocznymi na rysunku 26.1 jako „jest”).

W funkcji `setdata` wewnętrz klasy `FirstClass` przekazana wartość jest przypisywana do `self.data`. Wewnątrz metody argument `self` — nazwa konwencjonalnie nadawana argumentowi znajdującemu się najbardziej na lewo — automatycznie odnosi się do przetwarzanej instancji (`x` lub `y`), dzięki czemu przypisania przechowują wartości w przestrzeni nazw instancji, a nie klas (w taki sposób utworzone zostały zmienne `data` z rysunku 26.1).

Ponieważ klasy mogą generować wiele instancji, metody muszą przechodzić argument `self` w celu otrzymania przekazanej instancji. Kiedy wywołujemy metodę klasy `display` w celu wyświetlenia `self.data`, widzimy, że dla każdej instancji wynik będzie inny. Z drugiej strony, sama nazwa `display` jest taka sama w `x` oraz `y`, ponieważ przychodzi ona (jest dziedziczona) z klasą.

```

>>> x.display()                                # self.data różni się w każdej instancji
Król Artur
>>> y.display()
3.14159

```

Warto zauważyć, że w każdej instancji w składowej `data` przechowaliśmy różne typy obiektów (łańcuch znaków oraz liczbę zmiennoprzecinkową). Tak jak w każdym innym przypadku w Pythonie, deklaracje atrybutów instancji (czasami nazywanych *składowymi*, ang. *member*) nie istnieją. Atrybuty pojawiają się, gdy za pierwszym razem przypiszemy do nich wartości, podobnie jak proste zmienne. Tak naprawdę gdybyśmy wywołali metodę `display` na jednej z instancji przed wywołaniem metody `setdata`, wywołalibyśmy błąd niezdefiniowanej zmiennej. Atrybut `data` nie istnieje nawet w pamięci, dopóki nie zostanie on przypisany wewnątrz metody `setdata`.

By przekonać się, jak bardzo dynamiczny jest ten model, zobaczymy, że możemy zmodyfikować atrybuty instancji w samej klasie, przypisując wartość do `self` w metodach, lub poza klasą, przypisując ją do jawnego obiektu instancji.

```

>>> x.data = "Nowa wartość"                  # Pobiera lub ustawia atrybuty
>>> x.display()                            # Również poza klasą
Nowa wartość

```

Choć jest to mniej popularne, możemy nawet wygenerować nowy atrybut w przestrzeni nazw, przypisując wartość do jego nazwy poza funkcjami metod klasy.

```

>>> x.anothername = "mielonka"            # Można tutaj również ustawiać nowe atrybuty!

```

Powyższy kod powoduje dołączenie do obiektu instancji `x` nowego atrybutu o nazwie `anot` → `hername`, który mógł być wykorzystywany przez ktoś z metod klasy, ale nie musiał. Klasy zazwyczaj tworzą wszystkie atrybuty instancji, przypisując do argumentu `self`, jednak nie musi tak być. Programy mogą pobierać, modyfikować lub tworzyć atrybuty dowolnych obiektów, do których mają referencje.

Klasy dostosowuje się do własnych potrzeb przez dziedziczenie

Poza służeniem jako fabryki generujące wiele obiektów instancji klasy pozwalają nam na modyfikacje za pomocą wprowadzania nowych komponentów (o nazwie *klas podrzędnych*) w miejsce modyfikowania istniejących komponentów w miejscu. Obiekty instancji wygenerowane z klasy dziedziczą atrybuty klasy. Python pozwala również klasom na dziedziczenie po innych klasach, umożliwiając tworzenie *hierarchii* klas specjalizujących jakieś zachowanie za pomocą redefinowania atrybutów w klasach podrzędnych występujących niżej w hierarchii i przesłaniając tym samym bardziej ogólne definicje tych atrybutów znajdujące się wyżej w drzewie klas. W rezultacie im niżej w hierarchii schodzimy, tym bardziej wyspecjalizowany staje się program. I także tutaj nie można tego porównać z modułami — ich atrybuty istnieją w jednej, płaskiej przestrzeni nazw, która nie jest tak podatna na dostosowywanie do własnych potrzeb.

W Pythonie instancje dziedziczą po klasach, natomiast klasy dziedziczą po klasach nadrzędnych. Poniżej opisano najważniejsze koncepcje leżące u podstaw maszynierii dziedziczenia atrybutów.

- **Klasy nadrzędne są wymieniane w nawiasach w nagłówku instrukcji `class`.** By odziedziczyć atrybuty po innej klasie, wystarczy wymieścić tę klasę w nawiasach w nagłówku instrukcji `class`. Klasa dziedzicząca zazwyczaj nazywana jest *klasą podrzędną* (podklassą), natomiast klasa, z której się dziedziczy — *klasą nadzędną* (nadklassą).
- **Klasy dziedziczą atrybuty po swoich klasach nadrzędnych.** Podobnie jak instancje dziedziczą nazwy atrybutów zdefiniowane w klasach, tak samo klasy dziedziczą wszystkie nazwy atrybutów zdefiniowane w swoich klasach nadrzędnych. Python odnajduje je automatycznie, kiedy próbuje się uzyskać do nich dostęp, o ile nie istnieją one w klasach podrzędnych.
- **Instancje dziedziczą atrybuty po wszystkich dostępnych klasach.** Każda instancja otrzymuje zmienne z klasy, z której została wygenerowana, a także ze wszystkich klas nadrzędnych tej klasy. Przy szukaniu jakiejś zmiennej Python sprawdza instancję, później klasę, a następnie wszystkie klasy nadrzędne.
- **Każda referencja `obiekt.atrybut` wywołuje nowe, niezależne wyszukiwanie.** Python wykonuje niezależne wyszukiwanie w drzewie klas dla każdego wyrażenia pobierającego atrybut. Obejmuje to referencje do instancji i klas wykonane poza instrukcjami `class` (na przykład `X.atrybut`), a także referencje do atrybutów argumentu instancji `self` w funkcjach metod klasy. Każde wyrażenie `self.atrybut` w metodzie wywołuje nowe wyszukiwanie tego atrybutu w `self` oraz wyżej.
- **Zmiany logiki wykonuje się przez tworzenie klas podrzędnych, a nie modyfikację klas nadrzędnych.** Redefiniując zmienne z klas nadrzędnych w klasach znajdujących się niżej w hierarchii (drzewie klas), klasy podrzędne zastępują i tym samym dostosowują do własnych potrzeb odziedziczone zachowanie.

Rezultat i główny cel wyszukiwania jest taki, że klasy obsługują faktoryzację i dostosowywanie kodu do własnych potrzeb o wiele lepiej niż jakiekolwiek inne narzędzia języka, z jakimi się spotkaliśmy. Z jednej strony, pozwala nam to na minimalizację powtarzalności kodu (i tym samym zredukowanie kosztów jego utrzymania) ze względu na faktoryzację operacji na jedną, współdzieloną implementację. Z drugiej strony, pozwala nam to na programowanie za pomocą dostosowywania do własnych potrzeb czegoś, co już istnieje, zamiast modyfikowania kodu w miejscu lub pisania go od początku.

Drugi przykład

W celu zilustrowania roli dziedziczenia w kolejnym przykładzie rozwijamy kod zaprezentowany wcześniej. Najpierw zdefiniujemy nową klasę `SecondClass`, dziedziczącą wszystkie zmienne klasy `FirstClass` i udostępniającą jedną własną.

```
>>> class SecondClass(FirstClass):          # Dziedziczy metodę setdata
...     def display(self):                  # Modyfikuje metodę display
...         print('Aktualna wartość = "%s"' % self.data)
...
```

Klasa `SecondClass` definiuje metodę `display` w taki sposób, by wyświetlała ona dane w innym formacie. Definiując atrybut z taką samą nazwą jak atrybut klasy `FirstClass`, `SecondClass` w rezultacie zastępuje atrybut `display` ze swojej klasy nadzędnej.

Warto przypomnieć, że wyszukiwanie dziedziczenia postępuje w góre — od instancji, przez klasy podrzędne, do nadrzędnych, zatrzymując się na pierwszym wystąpieniu nazwy atrybutu, jaką znajduje. W tym przypadku, ponieważ nazwa `display` z klasy `SecondClass` zostanie odnalezione przed tą samą nazwą z klasy `FirstClass`, mówimy, że `SecondClass` przesłania (zastępuje, nadpisuje, ang. *override*) metodę `display` z `FirstClass`. Czasami takie zastępowanie atrybutów przez ich redefiniowanie niżej nazywamy *przeciążaniem* drzewa.

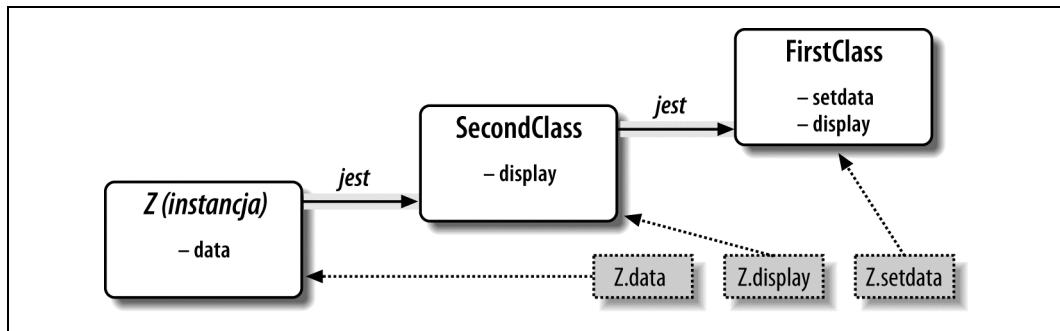
W rezultacie klasa `SecondClass` specjalizuje `FirstClass`, modyfikując zachowanie metody `display`. Z drugiej strony, `SecondClass` (i wszystkie instancje utworzone z tej klasy) nadal dziedziczy metodę `setdata` w takiej postaci, w jakiej występuje ona w klasie `FirstClass`. Utwórzmy nową instancję, żeby to zademonstrować.

```
>>> z = SecondClass()
>>> z.setdata(42)                      # Odnajduje metodę setdata w FirstClass
>>> z.display()                        # Odnajduje przesłoniętą metodę w SecondClass
Aktualna wartość = "42"
```

Tak jak wcześniej, tworzymy obiekt instancji `SecondClass`, wywołując tę klasę. Wywołanie metody `setdata` nadal wykonuje wersję z `FirstClass`, jednak tym razem atrybut `display` pochodzi z `SecondClass` i wyświetla zmodyfikowany komunikat. Na rysunku 26.2 przedstawiono przestrzenie nazw, jakie uczestniczą w tych działaniach.

A oto bardzo istotna kwestia związana z programowaniem zorientowanym obiektowo. Specjalizacja wprowadzona w klasie `SecondClass` jest całkowicie *zewnętrzna* dla `FirstClass`. Oznacza to, że nie wpływa ona na istniejące lub przyszłe obiekty `FirstClass`, takie jak `x` z poprzedniego przykładu.

```
>>> x.display()                         # x nadal jest instancją FirstClass (stary komunikat)
Nowa wartość
```



Rysunek 26.2. Specjalizacja poprzez przesłonięcie odziedziczonych nazw przez zredefiniowanie ich w rozszerzeniach niżej w drzewie klas. Na rysunku klasa SecondClass redefiniuje i dostosowuje metodę „display” do własnych potrzeb jej instancji

Zamiast modyfikować klasę FirstClass, dostosowujemy ją do własnych potrzeb. Jest to oczywiście sztuczny przykład, jednak z zasady — ponieważ dziedziczenie pozwala nam na wprowadzanie takich zmian w komponentach zewnętrznych (to znaczy klasach podanych) — klasy często obsługują rozszerzanie i ponowne wykorzystanie kodu o wiele lepiej, niż mogą to robić funkcje oraz moduły.

Klasy są atrybutami w modułach

Zanim przejdziemy dalej, należy pamiętać, że w nazwach klas nie ma nic magicznego. Jest to po prostu zmienna przypisywana do obiektu, kiedy wykonywana jest instrukcja `class`, a do samego obiektu możemy odnosić się za pomocą każdego normalnego wyrażenia. Gdyby na przykład nasza klasa FirstClass została utworzona w pliku modułu, a nie wpisana w sesji interaktywnej, moglibyśmy ją zimportować i wykorzystać w normalny sposób w wierszu nagłówka innej klasy.

```
from modulename import FirstClass
class SecondClass(FirstClass):
    def display(self): ...
```

Skopiowanie nazwy do mojego zakresu
Bezpośrednie wykorzystanie nazwy klasy

Poniższe rozwiązanie dałoby taki sam efekt.

```
import modulename
class SecondClass(modulename.FirstClass):
    def display(self): ...
```

Dostęp do całego modułu
Składnia kwalifikująca tworzy referencję

Jak wszystko inne, nazwy klas istnieją wewnątrz modułów, dlatego muszą przestrzegać wszystkich reguł omawianych w piątej części książki. W jednym pliku modułu można na przykład zapisać więcej niż jedną klasę — tak jak inne instrukcje modułów, instrukcje `class` wykonywane są w czasie importowania w celu zdefiniowania nazw, a nazwy te stają się osobnymi atrybutami modułu. Mówiąc bardziej ogólnie, każdy moduł może w dowolny sposób mieszać dowolną liczbę zmiennych, funkcji oraz klas, a wszystkie nazwy w module zachowują się w ten sam sposób. Demonstруje to plik `food.py`.

```
# Plik food.py

var = 1
def func():
    ...
class spam:
```

food.var
food.func
food.spam

```
...  
class ham:                                # food.ham  
...  
class eggs:                                # food.eggs  
...
```

Tak samo będzie nawet wtedy, gdy moduł i klasa noszą tę samą nazwę. Kiedy na przykład mamy poniższy plik *person.py*:

```
class person:  
...
```

by pobrać klasę, jak zawsze musimy przejść przez moduł:

```
import person                                # Zimportowanie modułu  
x = person.person()                          # Klasa wewnętrz modułu
```

Choć taka ścieżka może się wydawać zbędna, jest konieczna — *person.person* odnosi się do klasy *person* wewnątrz modułu *person*. Samo *person* powoduje pobranie modułu, a nie klasy, o ile oczywiście wcześniej nie użyjemy instrukcji *from*.

```
from person import person                      # Pobranie klasy z modułu  
x = person()                                  # Wykorzystanie nazwy klasy
```

Tak jak w przypadku każdej innej zmiennej, nigdy nie zobaczymy klas z pliku bez uprzedniego zainportowania i pobrania jej w jakiś sposób z pliku, który ją zawiera. Jeśli wydaje się to mylące, nie należy używać tej samej nazwy dla modułu oraz znajdującej się w nim klasy. Tak naprawdę powszechnie stosowana w Pythonie konwencja zaleca rozpoczęwanie nazw klas od wielkiej litery, tak by lepiej się one odróżniały.

```
import person                                # Mała litera w modułach  
x = person.Person()                          # Wielka litera w klasach
```

Należy również pamiętać, że choć klasa i moduły są przestrzeniami nazw służącymi do dołączania atrybutów, odpowiadają różnym strukturom w kodzie źródłowym. Moduł odzwierciedla cały plik, natomiast klasa — instrukcję wewnątrz tego pliku. Więcej informacji na temat tych różnic znajdziesz później w tej części książki.

Klasy mogą przechytywać operatory Pythona

Przyjrzyjmy się teraz trzeciej różnicy między klasami a modułami — *przeciążaniu operatorów*. W uproszczeniu przeciążanie operatorów pozwala obiektom zapisanym za pomocą klas przechytywać i odpowiadać na operacje, które działają na typach wbudowanych — takich jak dodawanie, tworzenie wycinków, wyświetlanie czy kwalifikacja. W dużej mierze jest to mechanizm automatyczny — wyrażenia oraz inne operacje wbudowane kierują sterowanie do implementacji w klasach. I w tym przypadku nie ma czegoś podobnego w modułach — moduły mogą implementować wywołania funkcji, ale nie zachowanie wyrażeń.

Choć moglibyśmy zaimplementować zachowanie klas jako funkcje metod, przeciążanie operatorów pozwala obiektom na ściszęszą integrację z modelem obiektów Pythona. Co więcej, ponieważ przeciążanie operatorów sprawia, że nasze własne obiekty zachowują się jak obiekty wbudowane, powoduje to powstawanie bardziej spójnych i łatwiejszych do zrozumienia interfejsów. Pozwala także na przetwarzanie obiektów opartych na klasach za pomocą kodu napisanego pod kątem interfejsów typów wbudowanych. Poniżej znajduje się krótkie streszczenie najważniejszych koncepcji związanych z przeciążaniem operatorów.

- **Metody zawierające w nazwie podwójne znaki _ (jak `_x_`) są specjalnymi punktami zaczepienia.** Przeciążanie operatorów jest w Pythonie zaimplementowane za pomocą udostępniania metod o specjalnych nazwach, które przechwytyują operacje. Język Python definiuje stałe i niezmienne odwzorowanie każdej z tych operacji na metodę o specjalnej nazwie.
- **Takie metody wywoływane są automatycznie, kiedy instancje pojawiają się w operacjach wbudowanych.** Jeśli na przykład obiekt instancji dziedziczy metodę `_add_`, metoda ta wywoływana jest za każdym razem, gdy obiekt ten pojawi się w wyrażeniu ze znakiem `+`. Wartość zwracana przez metodę staje się wynikiem odpowiadającego jej wyrażenia.
- **Klasy mogą nadpisywać większość operacji na typach wbudowanych.** Istnieją dziesiątki specjalnych nazw metod przeciążania operatorów, które służą do przechwytywania i implementowania prawie każdej operacji dostępnej dla typów wbudowanych. Obejmuje to wyrażenia, ale również podstawowe operacje, takie jak wyświetlanie czy tworzenie obiektów.
- **Nie istnieją wartości domyślne dla metod przeciążania operatorów i nie są one potrzebne.** Jeśli klasa nie definiuje lub nie dziedziczy metody przeciążania operatorów, oznacza to po prostu, że odpowiadająca jej operacja nie jest obsługiwana na instancjach klasy. Jeśli na przykład nie ma metody `_add_`, wyrażenia z operatorem `+` powodują zgłoszenie błędu.
- **Operatory pozwalają klasom na integrację z modelem obiektów Pythona.** Przeciążając operacje na typach, obiekty zdefiniowane przez użytkowników zaimplementowane za pomocą klas mogą się zachowywać tak samo jak obiekty wbudowane, co daje nam spójność, a także zgodność z oczekiwanyymi interfejsami.

Przeciążanie operatorów jest opcjonalne. Wykorzystywane jest przede wszystkim przez osoby tworzące narzędzia przeznaczone dla innych programistów Pythona, a nie przez samych twórców aplikacji. I szczerze mówiąc, nie powinno się próbować go używać tylko dla tego, że wydaje się fajne. Dopóki klasa naprawdę nie potrzebuje naśladować wbudowanych interfejsów, powinniśmy się ograniczyć do prostszych nazwanych metod. Po co aplikacja przeznaczona dla bazy danych pracowników miałaby na przykład obsługiwać rozszerzenia takie, jak `*` oraz `?` Nazwane metody, takie jak `giveRaise` oraz `promote`, zazwyczaj mają większy sens.

Z tego powodu nie będziemy się w tej książce zagłębiać w szczegóły każdej metody przeciążania operatorów dostępnej w Pythonie. Istnieje jednak jedna metoda przeciążania operatorów, która występuje w prawie każdej prawdziwej klasie Pythona — metoda `_init_`, znana jako metoda *konstruktora*, wykorzystywana do inicjalizacji stanu obiektów. Powinniśmy zwrócić na nią szczególną uwagę, ponieważ `_init_` wraz z argumentem `self` okazują się kluczami do zrozumienia większości kodu zorientowanego obiektowo w Pythonie.

Trzeci przykład

Przejdzmy do kolejnego przykładu. Tym razem zdefiniujemy klasę podzielną dla `SecondClass`, implementującą trzy atrybuty o specjalnych nazwach, które Python wywołuje automatycznie:

- Metoda `_init_` wykonywana jest, kiedy konstruowany jest nowy obiekt instancji (`self` jest nowym obiektem klasy `ThirdClass`)¹.

¹ Nie należy tego mylić z plikami `__init__.py` w pakietach modułów! Więcej informacji na ten temat znajduje się w rozdziale 23.

- Metoda `__add__` wykonywana jest, kiedy instancja tej klasy pojawi się w wyrażeniu z operatorem `+`.
- Metoda `__str__` wykonywana jest, kiedy obiekt jest wyświetlany (z technicznego punktu widzenia, kiedy przekształcana jest na łańcuch znaków wyświetlania za pomocą wbudowanej funkcji `str` lub jej wewnętrznego odpowiednika w Pythonie).

Nasza nowa klasa podzielona definiuje również normalnie nazwaną metodę `mul`, modyfikującą obiekt w miejscu. Poniżej widać kod nowej podklasy.

```
>>> class ThirdClass(SecondClass):           # Dziedziczy po SecondClass
...     def __init__(self, value):            # Przy "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):              # Przy "self + other"
...         return ThirdClass(self.data + other)
...     def __str__(self):                   # Przy "print(self)", "str()"
...         return '[ThirdClass: %s]' % self.data
...     def mul(self, other):                # Modyfikacja w miejscu: nazwana metoda
...         self.data *= other
...
>>> a = ThirdClass('abc')                  # Wywołanie metody __init__
>>> a.display()                          # Wywołanie dziedziczonej metody
Aktualna wartość = "abc"
>>> print(a)                            # __str__ zwraca wyświetlany łańcuch znaków
[ThirdClass: abc]

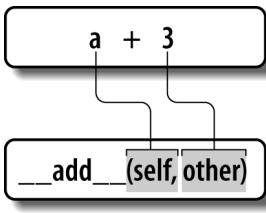
>>> b = a + 'xyz'                      # __add__: utworzenie nowej instancji
>>> b.display()                        # b ma wszystkie metody klasy ThirdClass
Aktualna wartość = "abcxyz"
>>> print(b)                           # __str__ zwraca wyświetlany łańcuch znaków
[ThirdClass: abcxyz]

>>> a.mul(3)                           # mul: modyfikuje instancję w miejscu
>>> print(a)
[ThirdClass: abcabcabc]
```

Klasa `ThirdClass` jest podklassą `SecondClass`, dlatego jej instancje dziedziczą dostosowaną do własnych potrzeb metodę `display` po tej klasie nadzędnej. Wywołania tworzące obiekty klasy `ThirdClass` przekazują tym razem jednak argument (na przykład `abc`). Jest on przekazywany do argumentu `value` konstruktora `__init__` i przypisywany do atrybutu `self.data`. W rezultacie `ThirdClass` sprawia, że atrybut `data` ustawiany jest automatycznie w czasie tworzenia, nie wymagając już wywoływanego metody `setdata` po tym fakcie.

Co więcej, obiekty klasy `ThirdClass` mogą się teraz pojawiać w wyrażeniach z operatorem `+` oraz wywołaniach funkcji `print`. W przypadku operatora `+` Python przekazuje obiekt instancji znajdujący się po lewej stronie do argumentu `self` w metodzie `__add__`, natomiast wartość z prawej strony do zmiennej `other`, zgodnie z rysunkiem 26.3. Wartość zwracana przez metodę `__add__` staje się wynikiem wyrażenia z operatorem `+`. W przypadku funkcji `print` Python przekazuje wyświetlany obiekt do `self` w metodzie `__str__`. łańcuch znaków zwracany przez tę metodę przyjmowany jest za wyświetlany łańcuch znaków obiektu. Dzięki `__str__` możemy wykorzystać zwykłe wywołanie `print` do wyświetlania obiektów tej klasy, zamiast wywoływać specjalną metodę `display`.

Metody o specjalnych nazwach, takie jak `__init__`, `__add__` oraz `__str__` są dziedziczone przez klasy podzielone oraz instancje, podobnie do pozostałych nazw przypisanych w instrukcji `class`. Jeśli metody nie zostaną zapisane w klasie, Python jak zwykle szuka nazw we wszystkich



Rysunek 26.3. W przeciążaniu operatorów operatory wyrażeń oraz inne operacje wbudowane wykonywane na instancjach klas są odwzorowywane na metody o specjalnych nazwach w klasach. Te metody specjalne są opcjonalne i mogą być dziedziczone w zwykły sposób. Na rysunku wyrażenie ze znakiem + wywołuje metodę `__add__`.

klasach nadrzędnych. Nazwy metod przeciążania operatorów nie są również słowami wbudowanymi lub zastrzeżonymi — są to tylko atrybuty, których Python szuka, kiedy obiekty pojawiają się w różnych kontekstach. Python zazwyczaj wywołuje je automatycznie, jednak czasami mogą one również zostać wywołane przez nasz kod; metoda `__init__` jest na przykład często wywoływana ręcznie w celu uruchomienia konstruktorów klas nadrzędnych (więcej informacji na ten temat później).

Warto zauważyć, że metoda `__add__` tworzy oraz zwraca *nowy* obiekt instancji klasy (wywołując `ThirdClass` z wartością wynikową). Metoda `mul` z kolei modyfikuje bieżący obiekt instancji w miejscu, przypisując go z powrotem do atrybutu `self`. Moglibyśmy przeciążyć także wyrażenie z operatorem `*`, by robiło to samo, jednak różniłoby się to od działania operatora `*` dla typów wbudowanych, takich jak liczby oraz łańcuchy znaków, dla których zawsze tworzone są nowe obiekty. Przyjęta praktyka zaleca, by przeciążone operatory działały w ten sam sposób, jak działają implementacje operatorów dla typów wbudowanych. Ponieważ przeciążanie operatorów jest tak naprawdę mechanizmem odwzorowania z wyrażenia na metodę, możemy we własnych obiektach klas interpretować operatory w dowolny sposób.

Po co przeciąża się operatory?

Jako projektanci klas możemy wybierać, czy chcemy korzystać z przeciążania operatorów, czy też nie. Nasz wybór uzależniony jest od tego, w jak dużym stopniu nasze obiekty mają wyglądać i zachowywać się jak typy wbudowane. Jak wspomniano wcześniej, jeśli pominiemy metodę przeciążającą operator i nie odziedziczymy jej po klasie nadrzędnej, odpowiadająca jej operacja nie będzie obsługiwana w instancjach klasy. Jeśli spróbujemy ją wywołać, otrzymamy wyjątek (lub zostaną użyte standardowe wartości domyślne).

Szczerze mówiąc, wiele metod przeciążania operatorów wykorzystywanych jest tylko wtedy, gdy implementuje się obiekty o matematycznej naturze. Klasa wektora lub macierzy może na przykład przeciązać operator dodawania, natomiast klasa pracownika najprawdopodobniej nie będzie tego robić. W przypadku prostszych klas możemy w ogóle nie korzystać z przeciążania operatorów i zamiast tego polegać na jawnym wywoływaniu metod, które będą implementować zachowanie naszych obiektów.

Z drugiej strony, możemy się zdecydować na wykorzystywanie przeciążania operatorów, jeśli musimy przekazać obiekt zdefiniowany przez użytkownika do funkcji oczekującej zastosowania operatorów dostępnych dla typów wbudowanych, takich jak lista czy słownik. Implementacja

tego samego zbioru operatorów w klasie pozwoli zapewnić, że obiekty obsługują te same oczekiwane interfejsy i tym samym są zgodne z funkcją. Choć w niniejszej książce nie omówimy wszystkich metod przeciążania operatorów, kilka dodatkowych technik przeciążania zobaczymy w praktyce w rozdziale 29.

Jedna omówiona tutaj metoda przeciążania wydaje się pojawiać w każdej prawdziwej klasie — chodzi tu o metodę konstruktora `__init__`. Ponieważ pozwala ona klasom na natychmiastowe wypełnienie atrybutów w nowo utworzonych instancjach, konstruktor przydaje się w prawie każdym rodzaju klasy, jaki możemy utworzyć. Tak naprawdę nawet jeśli w Pythonie nie deklaruje się atrybutów instancji, zazwyczaj możemy się dowiedzieć, jakie atrybuty będzie miała instancja, sprawdzając kod metody `__init__` jej klasy.

Najprostsza klasa Pythona na świecie

W tym rozdziale zaczęliśmy szczegółowo omawiać składnię instrukcji `class`, jednak raz jeszcze chciałbym przypomnieć, że podstawowy model dziedziczenia tworzony przez klasy jest bardzo prosty. Tak naprawdę obejmuje on tylko wyszukiwanie atrybutów w drzewach połączonych obiektów. Możemy na przykład utworzyć klasę, w której nie ma niczego. Poniższa instrukcja tworzy klasę bez dołączonych atrybutów (pusty obiekt przestrzeni nazw).

```
>>> class rec: pass # Pusty obiekt przestrzeni nazw
```

Potrzebna nam jest tutaj omówiona w rozdziale 13. instrukcja `pass`, ponieważ nie mamy żadnych metod do zapisania w kodzie. Po utworzeniu klasy za pomocą wpisania tej instrukcji do sesji interaktywnej możemy zacząć dołączać atrybuty do klasy, przypisując do niej zmienne całkowicie poza oryginalną instrukcją `class`.

```
>>> rec.name = 'Edward' # Obiekty z atrybutami  
>>> rec.age = 40
```

Po utworzeniu tych atrybutów przez przypisanie możemy je pobrać za pomocą normalnej składni. Kiedy wykorzystujemy klasę w taki sposób, przypomina ona strukturę z języka C lub rekord z Pascala. Jest to tak naprawdę obiekt z dołączonymi nazwami pól (podobną pracę możemy wykonać z kluczami słowników, jednak wymaga to dodatkowych znaków).

```
>>> print(rec.name) # Jak struktura z języka C lub rekord  
Edward
```

Warto zauważyć, że kod ten działa, nawet jeśli nie istnieją jeszcze żadne instancje klasy. Klasy same w sobie są obiektami, nawet bez instancji. Tak naprawdę są one po prostu samodzielnyimi przestrzeniami nazw, dlatego dopóki mamy referencję do klasy, możemy ustawiać lub modyfikować jej atrybuty w dowolnym momencie. Zobaczmy jednak, co się dzieje, kiedy utworzymy dwie instancje.

```
>>> x = rec() # Instancje dziedziczą nazwy klas  
>>> y = rec()
```

Te instancje rozpoczynają swoje istnienie jako całkowicie puste obiekty przestrzeni nazw. Ponieważ jednak pamiętają klasę, z jakiej zostały utworzone, otrzymają dołączone do klasy atrybuty za pomocą dziedziczenia.

```
>>> x.name, y.name # Zmienna jest przechowywana tylko w klasie  
('Edward', 'Edward')
```

Tak naprawdę instancje nie mają własnych atrybutów — pobierają po prostu atrybut `name` z obiektu klasy, w którym jest on przechowywany. Jeśli jednak przypiszemy atrybut do instancji, tworzy on (lub modyfikuje) atrybut w tym obiekcie i żadnym innym. Referencje do atrybutów uruchamiają wyszukiwanie dziedziczenia, ale przypisania atrybutów wpływają jedynie na obiekty, w których wykonywane są przypisania. W poniższym kodzie obiekt instancji `x` otrzymuje własną wartość atrybutu `name`, natomiast `y` nadal dziedziczy atrybut `name` dołączony do klasy znajdującej się nad nim.

```
>>> x.name = 'Amadeusz'                                # Przypisanie modyfikuje jedynie x
>>> rec.name, x.name, y.name
('Edward', 'Amadeusz', 'Edward')
```

Tak naprawdę, jak zobaczymy w rozdziale 28., atrybuty obiektu przestrzeni nazw są zazwyczaj implementowane jako słowniki, a drzewa dziedziczenia klas są (mówiąc ogólnie) po prostu słownikami z łączami do innych słowników. Jeśli wiemy, gdzie szukać, możemy to zobaczyć.

Przykładowo atrybut `__dict__` jest słownikiem przestrzeni nazw dla większości obiektów opartych na klasach (niektóre klasy mogą także definiować atrybuty w `__slots__`, zaawansowanej i rzadko wykorzystywanej opcji, którą będziemy omawiać w rozdziałach 30. oraz 31.) Poniższy kod został wykonany w Pythonie 3.0. Kolejność nazw i zbiór obecnych zmiennych wewnętrznych dla `x` mogą się różnić w poszczególnych wydaniach Pythona, jednak obecne są wszystkie przypisane przez nas nazwy.

```
>>> rec.__dict__.keys()
['__module__', 'name', 'age', '__dict__', '__weakref__', '__doc__']

>>> list(x.__dict__.keys())
['name']

>>> list(y.__dict__.keys())                           # list() nie jest wymagane w Pythonie 2.6
[]
```

W kodzie powyżej w słowniku przestrzeni nazw klasy widoczne są przypisane do niego atrybuty `name` oraz `age`, obiekt `x` ma własny atrybut `name`, a `y` nadal jest pusty. Każda instancja ma jednak łącze do swojej klasy ze względu na dziedziczenie — nosi ono nazwę `__class__`, gdyby ktoś miał ochotę je zbadać.

```
>>> x.__class__
<class '__main__.rec'>
```

Klasy mają również atrybut `__bases__` będący krotką ich klas nadzujących.

```
>>> rec.__bases__                                     # () — pusta krotka w Pythonie 2.6
(<class 'object'>,)
```

Dzięki tym dwóm atrybutom widać, jak drzewa klas są reprezentowane w pamięci przez Pythona.

Najważniejszą informacją, jaką należy zapamiętać, jest to, że model klas Pythona jest bardzo dynamiczny. Klasy oraz instancje są po prostu obiektami przestrzeni nazw z atrybutami tworzonymi w locie przez przypisanie. Takie przypisania zazwyczaj mają w kodzie miejsce wewnętrz ciała instrukcji `class`, jednak mogą się pojawić w dowolnym miejscu, w którym mamy referencję do jednego z obiektów drzewa.

Nawet metody — normalnie tworzone za pomocą instrukcji `def` osadzonej w instrukcji `class` — można tworzyć całkowicie niezależnie od obiektu klasy. Poniżej znajduje się na przykład kod definiujący poza klasą prostą funkcję przyjmującą jeden argument.

```
>>> def upperName(self):
...     return self.name.upper() # Nadal potrzebuje self
```

W kodzie tym nie ma nic dotyczącego klas — to po prostu funkcja, która może w tym momencie być wywołana po prostu, pod warunkiem że przekażemy jej obiekt z atrybutem name (nazwa self nie sprawia, że funkcja staje się specjalna).

```
>>> upperName(x) # Wywołanie jak prostej funkcji
'AMADEUSZ'
```

Jeśli jednak przypiszemy tę prostą funkcję do atrybutu klasy, staje się ona jej metodą, którą można wywołać za pomocą dowolnej instancji (a także przez samą nazwę klasy, o ile instancję przekażemy ręcznie).²

```
>>> rec.method = upperName

>>> x.method() # Wykonanie metody w celu przetworzenia x
'AMADEUSZ'

>>> y.method() # To samo, jednak przekazuje y do self
'EDWARD'

>>> rec.method(x) # Można wywołać przez instancję lub klasę
'AMADEUSZ'
```

Normalnie klasy są wypełniane przez instrukcje class, a atrybuty instancji tworzone są przez przypisania do atrybutów self w funkcjach metod. Tak naprawdę jednak wcale nie musi tak być — programowanie zorientowane obiektowo w Pythonie polega przede wszystkim na wyszukiwaniu atrybutów w połączonych obiektach przestrzeni nazw.

Klasy a słowniki

Choć proste klasy z poprzedniego podrozdziału służą do zilustrowania podstaw modelu klas, stosowane przez nie techniki można wykorzystać do prawdziwej pracy. Przykładowo w rozdziale 8. pokazaliśmy, w jaki sposób można tworzyć słowniki zapisujące właściwości elementów naszego programu. Okazuje się, że również klasy mogą spełniać tę rolę — pakują one informacje w sposób podobny do słowników, jednak mogą także dołączać logikę przetwarzania w postaci metod. Dla odniesienia poniżej znajduje się przykład wykorzystanych wcześniej w książce informacji zapisanych w słowniku:

```
>>> rec = {}
>>> rec['name'] = 'mel' # Informacje zapisane w słowniku
>>> rec['age'] = 45
>>> rec['job'] = 'instruktor'
>>>
>>> print(rec['name'])
mel
```

² Tak naprawdę jest to jeden z powodów, dla których argument self musi w metodach Pythona być zawsze jawnym. Ponieważ metody mogą być tworzone jako proste funkcje niezależne od klasy, muszą w sposób jawnym odwoływać się do argumentu domniemanej instancji. Mogą być wywoływane albo jako funkcje, albo jako metody, a Python nie może w inny sposób zgadnąć lub założyć, że prosta funkcja może się w rezultacie stać metodą klasy. Najważniejszym powodem jawnego podawania argumentu self jest jednak to, że dzięki niemu znaczenie nazw staje się bardziej oczywiste. Nazwy, do których nie odnosimy się przez self, są zwykłymi zmiennymi, natomiast nazwy, do których referencje odbywają się przez self, są w sposób oczywisty atrybutami instancji.

Powyższy kod emuluje narzędzia takie jak rekordy z innych języków programowania. Jak jednak widzieliśmy, istnieje kilka sposobów uzyskania tego samego za pomocą klas. Chyba najprostsze będzie poniższe rozwiązanie, zamieniające klucze na atrybuty:

```
>>> class rec: pass
...
>>> rec.name = 'mel'                                # Informacje zapisane w klasie
>>> rec.age = 45
>>> rec.job = 'instruktor'
>>>
>>> print(rec.age)
40
```

Powyższy kod zawiera znacznie mniej składni niż jego odpowiednik oparty na słowniku. Wykorzystuje pustą instrukcję `class` do wygenerowania pustego obiektu przestrzeni nazw. Po utworzeniu pustej klasy wypełniamy ją, przypisując z czasem atrybuty klasy (tak jak wcześniej).

Takie rozwiązanie działa, jednak nowa instrukcja `class` będzie wymagana dla każdego odrębnego rekordu, jaki będzie nam potrzebny. Być może lepiej będzie zamiast tego generować *instancje* pustej klasy reprezentujące każdy z odrębnych elementów:

```
>>> class rec: pass
...
>>> pers1 = rec()                                  # Informacje zapisane w instancjach
>>> pers1.name = 'mel'
>>> pers1.job = 'instruktor'
>>> pers1.age = 40
>>>
>>> pers2 = rec()
>>> pers2.name = 'vls'
>>> pers2.job = 'programista'
>>>
>>> pers1.name, pers2.name
('mel', 'vls')
```

W powyższym przykładzie tworzymy dwa rekordy z tej samej klasy. Instancje rozpoczynają swoje istnienie jako puste, podobnie jak klasy. Następnie wypełniamy rekordy za pomocą przypisania do atrybutów. Tym razem jednak istnieją dwa odrębne obiekty, stąd dwa odrębne atrybuty `name`. Tak naprawdę instancje tej samej klasy nie muszą nawet mieć tego samego zbioru nazw atrybutów. W tym przykładzie jedna z nich ma unikalny atrybut `age`. Instancje są tak naprawdę odrębnymi przestrzeniami nazw, dlatego każda z nich ma odrębny słownik atrybutów. Choć normalnie są one wypełniane w sposób spójny za pomocą metod klas, instancje są o wiele bardziej elastyczne, niż można by się tego spodziewać.

Wreszcie możemy zamiast tego utworzyć w pełni rozbudowaną klasę implementującą rekord oraz jego przetwarzanie:

```
>>> class Person:
...     def __init__(self, name, job):           # Klasa = Dane + Logika
...         self.name = name
...         self.job = job
...     def info(self):
...         return (self.name, self.job)
...
>>> rec1 = Person('mel', 'instruktor')
>>> rec2 = Person('vls', 'programista')
>>>
>>> rec1.job, rec2.info()
('instruktor', ('vls', 'programista'))
```

Powyższe rozwiązywanie także tworzy większą liczbę instancji, jednak klasa tym razem nie jest pusta — dodaliśmy *logikę* (metody) inicjalizującą instancje w czasie utworzenia i zbierającą atrybuty w krotkę. Konstruktor narzuca tutaj spójność instancji, zawsze ustawiając atrybuty `name` oraz `job`. Metody klasy i atrybuty instancji tworzą razem *pakiet* łączący dane i logikę.

Moglibyśmy dalej rozszerzać ten kod, dodając do niego logikę obliczającą pensje czy przetwarzającą zmienne. Możemy wreszcie połączyć tę klasę w większą hierarchię w celu odziedziczenia istniejącego zbioru metod za pomocą automatycznego wyszukiwania atrybutów w klasach czy wręcz przechować instancje klasy w pliku za pomocą serializacji z modułu `pickle` Pythona w celu uczynienia ich trwałymi. Tak naprawdę zrobimy to w kolejnym rozdziale, gdzie rozwinieśmy analogię między klasami a rekordami w bardziej realistyczny, działający przykład demonstrujący podstawy klas.

Choć typy takie jak słowniki są elastyczne, klasy pozwalają nam dodawać do obiektów działania, których typy wbudowane oraz proste funkcje nie obsługują w sposób bezpośredni. Możemy przechować funkcje także w słownikach, jednak użycie ich w celu przetworzenia domniemanych instancji nie jest aż tak naturalne, jak ma to miejsce w przypadku klas.

Podsumowanie rozdziału

W niniejszym rozdziale wprowadziliśmy podstawy tworzenia klas w Pythonie. Omówiliśmy składnię instrukcji `class` i zobaczyliśmy, jak wykorzystuje się ją w celu utworzenia drzewa dziedziczenia klas. Widzieliśmy również, jak Python automatycznie wypełnia pierwszy argument w funkcjach metod, jak atrybuty dołączane są do obiektów w drzewie klas za pomocą zwykłego przypisania, a także jak noszące specjalne nazwy metody przeciążania operatorów przechwytyują oraz implementują wbudowane operacje dla naszych instancji (na przykład wyrażenia czy wyświetlanie).

Skoro wiemy już wszystko o mechanizmach tworzenia kodu klas w Pythonie, w kolejnym rozdziale przejdziemy do bardziej rozbudowanego i realistycznego przykładu łączącego większość z tego, czego dowiedzieliśmy się dotychczas na temat programowania zorientowanego obiektowo. Później będziemy kontynuować przyglądanie się tworzeniu kodu klas, raz jeszcze powracając do modelu klas w Pythonie i uzupełniając pominięte tutaj dla uproszczenia szczegóły. Najpierw jednak przejdźmy do quizu, który pozwoli powtórzyć omówione dotychczas zagadnienia.

Sprawdź swoją wiedzę — quiz

1. W jaki sposób klasy powiązane są z modułami?
2. W jaki sposób tworzone są klasy oraz instancje?
3. Gdzie i w jaki sposób tworzone są atrybuty klas?
4. Gdzie i w jaki sposób tworzone są atrybuty instancji?
5. Co w klasie Pythona oznacza `self`?
6. W jaki sposób w kodzie klasy Pythona zapisuje się przeciążanie operatorów?

7. Kiedy możemy chcieć obsługiwać w klasach przeciążanie operatorów?
8. Któża metoda przeciążania operatorów wykorzystywana jest najczęściej?
9. Jakie są dwie kluczowe koncepcje niezbędne do zrozumienia kodu zorientowanego obiektowo napisanego w Pythonie?

Sprawdź swoją wiedzę — odpowiedzi

1. Klasy są zawsze zagnieżdżane w modułach; są one atrybutami obiektu modułu. Klasy i moduły są przestrzeniami nazw, jednak klasy odpowiadają instrukcjom (nie całym plikom) i obsługują koncepcje programowania zorientowanego obiektowo takie, jak wiele instancji, dziedziczenie oraz przeciążanie operatorów. W pewnym sensie moduł przypomina klasę z jedną instancją i bez dziedziczenia, odpowiadającą całemu plikowi kodu.
2. Klasy są tworzone poprzez wykonanie instrukcji `class`. Instancje tworzy się, wywołując klasę tak samo, jakby była ona funkcją.
3. Atrybuty klas tworzy się, przypisując atrybuty do obiektu klasy. Normalnie są one generowane przez przypisania najwyższego poziomu zagnieżdżone wewnętrz instrukcji `class`. Każda nazwa przypisana w bloku instrukcji `class` staje się atrybutem obiektu klasy (z technicznego punktu widzenia zakres instrukcji `class` staje się przestrzenią nazw atrybutów obiektu klasy). Atrybuty klas można jednak również tworzyć, przypisując je do klasy w dowolnym miejscu, w którym istnieje obiekt klasy — to znaczy nawet poza instrukcją `class`.
4. Atrybuty instancji tworzy się, przypisując atrybuty do obiektów instancji. Normalnie tworzone są wewnątrz funkcji metod klasy, wewnątrz instrukcji `class`, poprzez przypisanie atrybutów do argumentu `self` (który zawsze jest domniemaną instancją). Mogą one jednak również być tworzone przez przypisanie w dowolnym miejscu, w którym pojawia się referencja do instancji, nawet poza instrukcją `class`. Normalnie wszystkie atrybuty instancji są inicjalizowane w metodzie konstruktora `__init__`. W ten sposób późniejsze wywołania metod mogą zakładać, że atrybuty już istnieją.
5. `self` to nazwa, zgodnie z przyjętą konwencją, nadawaną pierwszemu (znajdującemu się najbardziej na lewo) argumentowi funkcji metody klasy. Python automatycznie wypełnia ten argument obiektem instancji, który jest domniemanym podmiotem wywołania metody. Argument ten nie musi nosić nazwy `self` (choć ta konwencja nazewnictwa jest bardzo silna), znaczenie ma tu jego pozycja. Być może dawni programiści języków C++ czy Java wolą nazywać go `this`, ponieważ w tych językach nazwa ta odzwierciedla tę samą koncepcję. W Pythonie jednak argument ten musi być podany w jawnym sposobie.
6. Przeciążanie operatorów w klasach Pythona zapisuje się za pomocą metod o specjalnych nazwach. Wszystkie one zaczynają się i kończą podwójnymi znakami `_`, tak by ich wygląd był unikalny. Nie są to nazwy wbudowane ani zarezerwowane; Python wykonuje je automatycznie, kiedy instancja pojawia się w odpowiadającej im operacji. Sam Python definiuje odwzorowania z operacji na specjalne nazwy metod.
7. Przeciążanie operatorów przydaje się w implementacji obiektów, które przypominają typy wbudowane (na przykład sekwencji czy obiektów numerycznych, takich jak macierze), a także do naśladowania wbudowanych interfejsów typów oczekiwanych przez fragment kodu. Naśladowanie wbudowanych interfejsów typów pozwala nam przekazywać instancje

klas mające jednocześnie informacje o stanie — to znaczy atrybuty pamiętające dane pomieśdz wywołaniami operacji. Nie powinniśmy jednak stosować przeciążania operatorów, kiedy wystarczy prosta nazwana metoda.

8. Najczęściej używana jest metoda konstruktora `__init__`. Prawie każda klasa wykorzystuje ją do ustawienia początkowych wartości atrybutów instancji i wykonania innych zadań startowych.
9. Dwoma podstawami kodu zorientowanego obiektowo w Pythonie są specjalny argument `self` w funkcjach metod oraz metoda konstruktora `__init__`.

Bardziej realistyczny przykład

W szczególności składni klas zagłębimy się bardziej w kolejnym rozdziale. Najpierw jednak chciałbym zaprezentować bardziej realistyczny przykład działania klas, który będzie o wiele bardziej praktyczny od tego, co widzieliśmy dotychczas. W niniejszym rozdziale zbudujemy zbiór klas wykonujących coś o wiele bardziej konkretnego — zapisujących i przetwarzających informacje o ludziach. Jak zobaczymy, to, co w programowaniu w Pythonie nazywamy *klasami* i *instancjami*, często może pełnić te same role, jakie co w bardziej tradycyjnym rozumieniu pełnią *rekordy* i *programy*.

W niniejszym rozdziale utworzymy kod dwóch klas:

- `Person` — klasy tworzącej i przetwarzającej informacje o osobach,
- `Manager` — dostosowanie klasy `Person` do własnych potrzeb, modyfikujące odziedziczone działanie.

Przy okazji będziemy tworzyli instancje obu klas i sprawdzali ich działanie. Po zakończeniu pokażę interesujący przykładowy przypadek użycia dla klas — przechowamy je w zorientowanej obiektywnie bazie danych typu *shelve*, tak by uczynić je trwałymi. W ten sposób można wykorzystać ten kod jako szablon dla utworzenia pełnej bazy danych osób, napisanej w całości w Pythonie.

Poza próbą pokazania użyteczności tych rozwiązań naszym celem jest także *edukacja*, gdyż rozdział ten stanowi również prezentację programowania zorientowanego obiektywnie w Pythonie. Często zdarza się, że ludzie rozumieją składnię klas z poprzedniego rozdziału na papierze, ale nie wiedzą, od czego zacząć, gdy muszą sami od podstaw napisać kod nowej klasy. W tym celu podzielimy pracę na etapy, tak by móc opanować podstawy. Klassy będziemy budować stopniowo, aby można było zobaczyć, w jaki sposób ich możliwości łączą się ze sobą w pełne programy.

Pod koniec rozdziału nasze klasy będą nadal stosunkowo skromne, jeśli chodzi o ilość kodu, jednak będą demonstrować *wszystkie* najważniejsze koncepcje modelu programowania zorientowanego obiektywnie w Pythonie. Obok szczegółów składni system klas Pythona sprowadza się w dużej mierze do szukania atrybutu w drzewie obiektów wraz ze specjalnym pierwszym argumentem dla funkcji.

Krok 1. — tworzenie instancji

Tyle na temat fazy projektowania — przejdźmy teraz do implementacji. Nasze pierwsze zadanie będzie polegało na rozpoczęciu tworzenia kodu głównej klasy — Person. W oknie ulubionego edytora tekstu należy utworzyć nowy plik dla pisanej klasy. W Pythonie silną konwencją jest rozpoczynanie nazw modułów od małych liter, natomiast nazw klas od wielkich liter. Podobnie jak nazwa argumentów `self` w metodach, nie jest to wymagane przez język, jednak na tyle popularne, że zmiana tej konwencji może być myląca dla osób, które później będą odczytywać nasz kod. W celu dostosowania się do tego zwyczaju nazwiemy nasz nowy plik modułu `person.py`, natomiast klasie nadamy nazwę Person, jak poniżej:

```
# File person.py (początek)
```

```
class Person:
```

Aż do pewnego momentu w dalszej części rozdziału wszystkie działania będziemy wykonywać w tym właśnie pliku. W jednym pliku modułu w Pythonie możemy zapisywać kod dowolnej liczby funkcji i klas, jednak nazwa `person.py` może stracić sens, jeśli później będziemy do tego pliku dodawać niezwiązane z nim komponenty. Na razie jednak zakładamy, że cały umieszczony w nim kod będzie powiązany z klasą Person. Tak zresztą powinno być — jak już wiemy, moduły działają najlepiej, jeśli mają jeden, spójny cel.

Tworzenie konstruktorów

Pierwszą czynnością, jaką chcemy wykonywać za pomocą klasy Person, jest zapisywanie podstawowych informacji o osobach — na przykład w celu wypełnienia pól rekordu. Oczywiście w Pythonie informacje te znane są pod nazwą *atrybutów* obiektów instancji i tworzone są zazwyczaj za pomocą przypisania do atrybutów `self` w funkcjach metod klas. Normalnym sposobem nadania atrybutom instancji ich pierwszych wartości jest przypisanie ich do `self` w *metodzie konstruktora* `__init__`, która zawiera kod wykonywany przez Pythona — automatycznie za każdym razem, gdy tworzona jest instancja. Dodajmy taką metodę do naszej klasy:

```
# Dodanie inicjalizacji pola rekordu
```

```
class Person:  
    def __init__(self, name, job, pay):  
        self.name = name  
        self.job = job  
        self.pay = pay
```

Konstruktor przyjmuje 3 argumenty
Wypełnienie pól przy tworzeniu
self to obiekt nowej instancji

Jest to bardzo popularny wzorzec kodu — przekazujemy dane, które będą dołączone do instancji jako argumenty, do metody konstruktora, i przypisujemy je do `self` w celu trwałego ich zachowania. W terminologii programowania zorientowanego obiektem `self` jest nowo utworzonym obiektem instancji, natomiast `name` (imię i nazwisko), `job` (stanowisko) oraz `pay` (płaca) stają się *informacjami o stanie* — opisowymi danymi zapisanymi w obiekcie w celu późniejszego wykorzystania. Choć inne techniki (takie jak referencje z zakresów zawierających) także są w stanie zapisywać szczegóły, atrybuty instancji sprawiają, że odbywa się to w sposób jawnym i łatwy do zrozumienia.

Warto zwrócić uwagę na to, że nazwy argumentów pojawiają się w kodzie *dwukrotnie*. Kod ten na pierwszy rzut oka wydaje się powtarzać, jednak w rzeczywistości tak nie jest. Argument

`job` jest na przykład zmienną lokalną w zakresie funkcji `__init__`, natomiast `self.job` jest atrybutem instancji będącej sugerowanym podmiotem wywołania metody. Są to dwie różne zmienne, które — tak się składa — noszą tę samą nazwę. Przypisując zmienną lokalną `job` do atrybutu `self.job` za pomocą kodu `self.job = job`, zapisujemy przekazaną `job` w instancji w celu późniejszego wykorzystania. Jak zawsze w Pythonie miejsce przypisania zmiennej (lub obiekt, do którego jest ona przypisana) determinuje jej znaczenie.

A skoro już mowa o argumentach, w metodzie `__init__` nie ma nic magicznego poza faktem, że jest ona wywoływana automatycznie, kiedy tworzona jest instancja, i ma specjalny pierwszy argument. Pomimo dziwnej nazwy jest to normalna funkcja, która obsługuje wszystkie omówione przez nas możliwości funkcji. Możemy na przykład podać *wartości domyślne* dla niektórych z jej argumentów, tak by nie trzeba ich było podawać w sytuacjach, gdy ich wartości nie są dostępne bądź przydatne.

W celu zademonstrowania tej możliwości uczynimy argument `job` opcjonalnym. Jego wartością domyślną będzie `None`, co oznacza, że tworzona osoba nie jest (aktualnie) zatrudniona. Jeśli `job` ma wartość domyślną `None`, prawdopodobnie dla porządku będziemy chcieli, by `pay` było równe `0` (o ile oczywiście nie znamy kogoś, kto otrzymuje płacę, nie pracując!). Tak naprawdę musimy dla `pay` podać wartość domyślną, ponieważ zgodnie z regułami składni Pythona wszystkie argumenty w nagłówku funkcji po pierwszej wartości domyślnej muszą także mieć wartości domyślne.

Dodanie wartości domyślnych do argumentów konstruktora

```
class Person:  
    def __init__(self, name, job=None, pay=0):  
        self.name = name  
        self.job = job  
        self.pay = pay  
  
# Normalne argumenty funkcji
```

Kod ten oznacza, że przy tworzeniu obiektów `Person` musimy przekazać imiona i nazwiska (`name`), jednak stanowisko (`job`) oraz płaca (`pay`) są teraz opcjonalne — jeśli je pominiemy, będą miały wartości domyślne, odpowiednio, `None` oraz `0`. Argument `self` jest jak zwykle wypełniany przez Pythona automatycznie, tak by odwoływał się do obiektu instancji. Przypisanie wartości do atrybutów `self` powoduje dołączenie ich do nowej instancji.

Testowanie w miarę pracy

Powyższa klasa niewiele jeszcze robi — na razie wypełnia tylko pola nowego rekordu — jednak jest to prawdziwa, działająca klasa. W tym momencie możemy dodać do niej więcej kodu, by rozszerzyć nieco jej możliwości, jednak nie będziemy tego jeszcze robić. Co każdy już zapewne nauczył się doceniać, programowanie w Pythonie jest tak naprawdę *inkrementalnym prototypowaniem* — piszemy jakiś kod, testujemy go, piszemy więcej kodu, znowu testujemy... Ponieważ Python udostępnia sesję interaktywną i praktycznie natychmiastową reakcję po modyfikacji kodu, testowanie w miarę postępu prac jest o wiele bardziej naturalne od opisania ogromnej ilości kodu w celu wykonania wszystkich testów naraz.

Zanim zatem dodamy do kodu kolejne opcje, przetestujmy to, co już mamy, tworząc kilka instancji naszej klasy i wyświetlając ich atrybuty utworzone przez konstruktor. Moglibyśmy to robić interaktywnie, ale, co już wiadomo na tym etapie książki, testowanie interaktywne ma swoje ograniczenia — ponowne importowanie modułów i wpisywanie przypadków testowych

z każdym rozpoczęciem sesji testowania na nowo jest dość żmudne. Częściej programiści Pythona wykorzystują sesję interaktywną w przypadku jednorazowych testów, jednak większe testy wykonują, wpisując kod na dole pliku zawierającego obiekty do testowania, jak poniżej:

Dodanie inkrementalnego kodu testów samosprawdzających

```
class Person:  
    def __init__(self, name, job=None, pay=0):  
        self.name = name  
        self.job = job  
        self.pay = pay  
  
bob = Person('Robert Zielony')          # Test klasy  
anna = Person('Anna Czerwona', job='programista', pay=100000) # Automatycznie wykonuje __init__  
print(bob.name, bob.pay)                # Pobranie dodatkowych atrybutów  
print(anna.name, anna.pay)              # Atrybuty dla obiektów „bob” i „anna” różnią się
```

Warto zauważyć, że obiekt bob przyjmuje wartości domyślne dla atrybutów job oraz pay, natomiast obiekt anna udostępnia własne wartości. Należy także zwrócić uwagę na użycie argumentów ze słowami kluczowymi przy tworzeniu obiektu anna. Moglibyśmy zamiast tego przekazywać wartości przez pozycję, jednak słowa kluczowe pozwolą nam później sobie przypomnieć, czym są dane (a także pozwalają nam przekazywać argumenty w dowolnej kolejności od lewej do prawej strony). I znów, pomimo niezwykłej nazwy, `__init__` jest normalną funkcją, obsługującą wszystko, co już wiemy na temat funkcji — w tym wartości domyślne i przekazywanie argumentów ze słowami kluczowymi.

Kiedy plik ten zostanie wykonany jako skrypt, kod testu znajdujący się na dole tworzy dwie instancje naszej klasy i wyświetla dwa atrybuty każdej z nich (name oraz pay).

```
C:\misc> person.py  
Robert Zielony 0  
Anna Czerwona 100000
```

Kod testu z tego pliku można także wpisać w sesji interaktywnej Pythona (zakładając, że najpierw zaimportujemy tam klasę Person), jednak tworzenie gotowych testów wewnątrz modułu, jak powyżej, sprawia, że o wiele łatwiej jest je ponownie wykonać w przyszłości.

Choć kod ten jest stosunkowo prosty, demonstruje już coś istotnego. Jak widać, atrybut `name` obiektu bob nie jest tym samym co atrybut `name` obiektu anna; podobnie jest w przypadku atrybutu `pay`. Każdy z nich jest niezależnym rekordem informacji. Obiekty bob i anna są obiektami przestrzeni nazw — jak wszystkie instancje klasy, każdy z nich ma własną, niezależną kopię informacji o stanie utworzonych za pomocą klasy. Ponieważ każda instancja klasy ma swój własny zbiór atrybutów `self`, klasy są naturalnym rozwiązaniem służącym do zapisywania w ten sposób informacji dla większej liczby obiektów. Podobnie jak typy wbudowane, klasy służą jako rodzaj fabryki obiektów. Inne struktury programów Pythona, takie jak funkcje i moduły, nie mają takich możliwości.

Wykorzystywanie kodu na dwa sposoby

W obecnej postaci kod testu znajdujący się na dole pliku działa, jednak jest tu pewien haczyk — instrukcje `print` najwyższego poziomu wykonywane są zarówno wtedy, gdy plik jest wykonywany jako skrypt, jak i gdy importowany jest jako moduł. Oznacza to, że jeśli kiedykolwiek zdecydujemy się zimportować klasę z tego pliku w celu wykorzystania jej gdzieś indziej (co faktycznie zrobimy w dalszej części rozdziału), będziemy oglądali dane wyjściowe z kodu testu

za każdym razem, gdy plik jest importowany. Nie jest to zbyt dobre rozwiązanie w dziedzinie programowania — dla programów klientów nasze wewnętrzne testy nie mają znaczenia i nie chcą one oglądać naszych danych wyjściowych wymieszanych z własnymi.

Choć moglibyśmy wydzielić kod testu do osobnego pliku, często o wiele wygodniejsze jest wpisywanie testów do tego samego pliku co elementy, które są testowane. Lepiej byłoby zorganizować to tak, by wykonywać testy z dołu pliku *tylko* wtedy, gdy plik wykonywany jest w celach testowych, a nie gdy jest importowany. Właśnie do tego służy sprawdzanie atrybutu `__name__` modułu, co wiemy już z poprzedniej części niniejszej książki. Oto jak będzie wyglądało dodanie go do naszego kodu:

```
# Pozwala na importowanie pliku oraz wykonywanie i testowanie go

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

if __name__ == '__main__':          # Przy wykonywaniu jedynie w celu przetestowania
    # Kod testu samosprawdzającego
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob.name, bob.pay)
    print(anna.name, anna.pay)
```

Teraz otrzymujemy dokładnie takie zachowanie, o jakie nam chodzi. Wykonanie tego pliku w postaci skryptu najwyższego poziomu testuje go, ponieważ jego atrybut `__name__` ma wartość `__main__`, natomiast zaimportowanie pliku w postaci biblioteki klas nie wykonuje go.

```
C:\misc> person.py
Robert Zielony 0
Anna Czerwona 100000

c:\misc> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) ...
>>> import person
>>>
```

Po zaimportowaniu plik definiuje teraz klasę, jednak z niej nie korzysta. Po bezpośrednim wykonaniu plik tworzy dwie instancje naszej klasy, jak wcześniej, i znów wyświetla po dwa atrybuty każdej z instancji, a ponieważ każda instancja jest niezależnym obiektem przestrzeni nazw, wartości ich atrybutów różnią się od siebie.

Krok 2. — dodawanie metod

Na razie wszystko wygląda świetnie. W tym punkcie nasza klasa jest *fabryką* rekordów, tworzy i wypełnia pola rekordów (w terminologii Pythona — atrybuty instancji). Choć jest dość ograniczona, możemy na jej obiektach wykonywać pewne działania. Klasy dodają dodatkowy poziom struktury, jednak większość swojej pracy wykonują, osadzając i przetwarzając *podstawowe typy danych*, takie jak listy oraz łańcuchy znaków. Innymi słowy, jeśli wiemy już, jak korzystać z podstawowych typów danych Pythona, wiemy także sporo o klasach Pythona — klasy są tak naprawdę niewielkim rozszerzeniem strukturalnym.

Uwaga na temat zgodności z wersjami

Cały kod z niniejszego rozdziału wykonuję w Pythonie 3.0, wykorzystując składnię wywołania funkcji `print` z wersji 3.0. Jeśli pracujemy w Pythonie 2.6, kod będzie działał w obecnej postaci, jednak będzie można zauważać nawiasy wokół niektórych ze zwracanych wierszy, ponieważ dodatkowe nawiasy w instrukcjach `print` zamieniają kilka elementów w krotkę:

```
c:\misc> c:\python26\python person.py
('Robert Zielony', 0)
('Anna Czerwona', 100000)
```

Jeśli ta różnica ma powodować u kogoś bezsenne noce, wystarczy usunąć nawiasy, by móc korzystać z instrukcji `print` z wersji 2.6. Dodatkowych nawiasów można także uniknąć, wykorzystując formatowanie w celu zwrócenia do `print` pojedynczego obiektu. Oba rozwiązania działają w wersjach 2.6 oraz 3.0, choć forma metody jest nowsza:

```
print('{0} {1}'.format(bob.name, bob.pay))      # Nowa metoda formatująca
print('%s %s' % (bob.name, bob.pay))            # Wyrażenie formatujące
```

Przykładowo pole `name` z naszego obiektu jest prostym łańcuchem znaków, dzięki czemu możemy pobierać nazwiska z naszych obiektów, dzieląc łańcuchy w miejscu wystąpienia spacji i indeksując. Wszystko to są operacje na podstawowych typach danych, które działają bez względu na to, czy cele ich działania osadzone są w instancjach klas, czy też nie.

```
>>> name = 'Robert Zielony'          # Prosty łańcuch znaków, poza klasą
>>> name.split()                   # Ekstrakcja nazwiska
['Robert', 'Zielony']
>>> name.split()[-1]                # Lub [1], jeśli zawsze składa się z 2 części
'Zielony'
```

W podobny sposób możemy dać obiektyowi podwyżkę pensji, aktualniając jego pole `pay` — czyli zmieniając jego informacje o stanie w miejscu za pomocą przypisania. To zadanie obejmuje również proste operacje, które działają na obiektach podstawowych Pythona bez względu na to, czy są one samodzielne, czy osadzone są w strukturze klasy:

```
>>> pay = 100000                  # Prosta zmienna, poza klasą
>>> pay *= 1.10                   # 10% podwyżki
>>> print(pay)                   # Lub: pay = pay * 1.10, jeśli ktoś lubi pisać
110000.0                           # Lub: pay = pay + (pay * .10), jeśli ktoś _naprawdę_ lubi pisać!
```

W celu zastosowania tych działań do obiektów `Person` utworzonych za pomocą naszego skryptu wystarczy zrobić z `bob.name` i `anna.pay` to samo, co zrobiliśmy przed chwilą z `name` oraz `pay`. Operacje te będą te same, jednak obiekty docelowe dołączone są do atrybutów w naszej strukturze klasy:

```
# Przetworzenie osadzonych typów wbudowanych — łańcuchów znaków; zmienność

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    if __name__ == '__main__':
        bob = Person('Robert Zielony')
        anna = Person('Anna Czerwona', job='programista', pay=100000)
        print(bob.name, bob.pay)
```

```
print(anna.name, anna.pay)
print(bob.name.split()[-1])      # Pobranie nazwiska obiektu
anna.pay *= 1.10                 # Danie temu obiektowi podwyżki
print(anna.pay)
```

Dodaliśmy tutaj trzy ostatnie wiersze. Po wykonaniu pobierają one nazwisko obiektu bob za pomocą prostych działań na łańcuchach znaków i listach, a także dają obiektowi anna podwyżkę, modyfikując jego atrybut pay w miejscu za pomocą prostej operacji na liczbach. W pewnym sensie obiekt anna również jest obiektem *zmiennym* — jego stan zmienia się w miejscu podobnie jak lista po wywołaniu append.

```
Robert Zielony 0
Anna Czerwona 100000
Zielony
110000.0
```

Powyższy kod działa zgodnie z planem, jednak gdybyśmy pokazali go jakiemuś weteranowi programowania, najprawdopodobniej powiedziałby on nam, że takie ogólne podejście w praktyce nie jest najlepszym pomysłem. Wpisywanie działań takich jak powyższe na stałe, *poza klasą*, może prowadzić do problemów z utrzymaniem kodu w przyszłości.

Co na przykład stanie się, jeśli na stałe zapiszemy wzór kodu pobierającego nazwisko w wielu różnych miejscach programu? Jeśli kiedyś będziemy musieli zmodyfikować jego sposób działania (na przykład by móc obsługiwać nową strukturę danych osobowych), będąmy musieli odszukać i uaktualnić *każde* jego wystąpienie. W podobny sposób, jeśli kod obliczający podwyżkę kiedykolwiek się zmieni (na przykład wymagając zgody lub uaktualnienia bazy danych), będąmy musieli zmodyfikować większą liczbę jego kopii. Samo odnalezienie wystąpień takiego kodu może być w większych programach problematyczne — mogą one być rozsiane po wielu plikach czy podzielone na poszczególne kroki.

Tworzenie kodu metod

To, co tak naprawdę chcemy zrobić, to wykorzystać koncepcję z dziedziny programowania znaną pod nazwą *hermetyzowania* (kapsułkowania, ang. *encapsulation*). Hermetyzowanie polega na opakowaniu logiki operacji za interfejsami w taki sposób, by każda operacja była w naszym programie zapisana w kodzie tylko raz. W ten sposób, jeśli nasze potrzeby w przyszłości się zmieniają, trzeba będzie uaktualnić tylko jedną kopię. Co więcej, możemy prawie dowolnie modyfikować zawartość tej pojedynczej kopii bez psucia korzystającego z niej kodu.

W terminologii Pythona chcemy zapisać w kodzie nasze operacje na obiektach w *metodach* klasy, zamiast rozsiewać je po całym programie. Tak naprawdę jest to jedna z rzeczy, do których klasy świetnie się nadają — *faktoryzacja* kodu w celu usunięcia jego powtarzalności i tym samym zoptymalizowania utrzymywania. Dodatkową zaletą jest to, że zmiana działań w metody pozwala na zastosowanie ich do dowolnych instancji klasy, nie tylko do tych, których przetwarzanie zostało zapisane w kodzie na stałe.

Wszystko to w kodzie jest o wiele łatwiejsze, niż brzmi w teorii. Poniżej wykonujemy hermetyzację kodu, przesuwając dwie operacje z kodu poza klasą do metod klasy. A skoro już przy tym jesteśmy, zmodyfikujemy także kod testu samosprawdzającego na dole pliku w taki sposób, by wykorzystywał tworzone przez nas nowe metody zamiast zapisanych w kodzie na stałe operacji:

```

# Dodanie metod w celu hermetyzacji operacji i łatwiejszego utrzymania kodu

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                      # Metody zachowania
        return self.name.split()[-1]          # self to sugerowany podmiot
    def giveRaise(self, percent):             # Wystarczy zmienić tutaj
        self.pay = int(self.pay * (1 + percent))

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob.name, bob.pay)
    print(anna.name, anna.pay)
    print(bob.lastName(), anna.lastName())      # Użycie nowych metod
    anna.giveRaise(.10)                         # zamiast kodu zapisanego na stałe
    print(anna.pay)

```

Jak już wiemy, *metody* są po prostu normalnymi funkcjami dołączonymi do klas i zaprojektowanymi w taki sposób, by przetwarzać instancje zamiast tych klas. Instancja jest podmiotem wywołania metody i automatycznie przekazywana jest do argumentu `self` metody.

Przekształcenie kodu na metody w tej wersji jest dość łatwe do zrozumienia. Nowa metoda `lastName` robi na przykład z `self` to samo, co poprzednia, zapisana na stałe w kodzie wersja robiliła z obiektem `bob`, ponieważ `self` jest sugerowanym podmiotem wywołania metody. Metoda `lastName` zwraca również wynik, ponieważ operacja ta nazywana jest teraz funkcją. Oblicza ona wartość, której może użyć kod wywołujący, nawet jeśli ma ona posłużyć tylko do wyświetlenia. W podobny sposób nowa metoda `giveRaise` robi z `self` to samo, co wcześniej robiliśmy z obiektem `anna`.

Po wykonaniu kodu dane wyjściowe z pliku będą podobne do poprzednich — tak naprawdę w większości dokonaliśmy *refaktoryzacji* kodu w celu ułatwienia zmian w przyszłości, jednak nie modyfikowaliśmy jego sposobu działania:

```

Robert Zielony 0
Anna Czerwona 100000
Zielony Czerwona
110000

```

Warto zwrócić tutaj uwagę na kilka kwestii dotyczących kodu. Po pierwsze, atrybut `pay` obiektu `anna` po podwyżce nadal jest *liczbą całkowitą* — przekształcamy wynik obliczeń z powrotem na liczbę całkowitą, wywołując funkcję wbudowaną `int` wewnętrz metody. Zmiana wartości na `int` (liczbę całkowitą) lub `float` (liczbę zmienoprzecinkową) w większości sytuacji nie ma dużego znaczenia (obiekty liczb całkowitych i zmienoprzecinkowych mają te same interfejsy i można je ze sobą mieszać w wyrażeniach), jednak być może w prawdziwym systemie będziemy musieli poradzić sobie z problemem zaokrąglania (dla obiektów `Person` pieniądze z pewnością mają znaczenie!).

Jak wiemy z rozdziału 5., możemy sobie z tym poradzić za pomocą funkcji wbudowanej `round(N, 2)`, która pozwoli zaokrąglić i zachować grosze, bądź za pomocą typu `decimal` naprawiającego precyzję. Można także przechować wartość pieniężną w postaci pełnej liczby zmienoprzecinkowej i wyświetlić ją za pomocą lańcucha formatującego `% .2f` lub `{0: .2f}` w celu pokazania groszy. W przypadku tego przykładu grosze odetniemy za pomocą funkcji `int`.

Po drugie, warto zwrócić uwagę na to, że tym razem wyświetlamy także nazwisko obiektu anna — ponieważ logika z nazwiskiem została poddana hermetyzacji w metodzie, możemy ją wykorzystać na *dowolnej instancji klasy*. Jak widzieliśmy, Python mówi metodzie, którą instancję należy przetworzyć, automatycznie przekazując ją do pierwszego argumentu, zazwyczaj noszącego nazwę `self`. A dokładniej:

- W pierwszym wywołaniu — `bob.lastName()` — bob jest sugerowanym podmiotem przekazywanym do `self`.
- W drugim wywołaniu — `anna.lastName()` — do `self` trafia zamiast tego anna.

Warto prześledzić te wywołania, by zobaczyć, jak instancja trafia do `self`. W rezultacie metoda pobiera za każdym razem nazwę sugerowanego podmiotu. To samo dzieje się w przypadku `giveRaise`. Moglibyśmy na przykład dać podwyżkę obiekowi bob, wywołując w ten sposób metodę `giveRaise` dla obu instancji. Niestety, pensja obiektu bob w wysokości 0 uniemożliwia mu otrzymanie podwyżki w obecnej wersji programu (jest to coś, czym możemy się zająć w wersji 2.0 naszego oprogramowania).

Wreszcie warto zauważyc, że metoda `giveRaise` zakłada przekazanie zmiennej `percent` w postaci liczby zmienoprzecinkowej o wartości pomiędzy zero a jeden. W prawdziwym świecie takie założenie może być zbyt radykalne (podwyżka o 1000% zostałaby chyba przez większość z nas uznana za błęd!). Na potrzeby tego prototypu będzie to wystarczające, ale być może będziemy chcieli sprawdzić (a przynajmniej udokumentować) tę kwestię w przyszłej wersji naszego kodu. Powrócimy do tego pomysłu w późniejszym rozdziale książki, w którym zajmiemy się kodem czegoś o nazwie *dekoratory funkcji* i będziemy badać instrukcję `assert`. Oba rozwiązania alternatywne pozwolą nam na automatyczne wykonywanie testów poprawności w trakcie programowania.

Krok 3. — przeciążanie operatorów

W tym momencie mamy do dyspozycji klasę o pełnych możliwościach, generującą i inicjalizującą instancję, a także dwa nowe elementy działania służące do przetwarzania instancji (w formie metod). Jak na razie wszystko jest w porządku.

W takiej postaci testowanie jest jednak nieco mniej wygodne, niż być powinno — w celu prześledzenia naszych obiektów musimy ręcznie pobierać i wyświetlać *poszczególne atrybuty* (na przykład `bob.name`, `anna.pay`). Byłoby miło, gdyby wyświetlanie instancji w całości od razu dawało nam jakieś przydatne informacje. Niestety, domyślona forma wyświetlania obiektu instancji nie jest zbyt dobra — wyświetla ona nazwę klasy obiektu oraz jego adres w pamięci (co w Pythonie jest właściwie całkowicie zbędne, z wyjątkiem konieczności posiadania unikalnego identyfikatora).

By to zobaczyć, wystarczy zmodyfikować ostatni wiersz w skrypcie na `print(anna)`, tak by wyświetlał on obiekt jako całość. Oto, co otrzymamy — dane wyjaśnione mówią, że anna jest w 3.0 obiektem („object”), natomiast w 2.6 instancją („instance”):

```
Robert Zielony 0
Anna Czerwona 100000
Zielony Czerwona
<__main__.Person object at 0x02614430>
```

Udostępnienie wyświetlania

Na szczęście łatwo możemy poprawić tę sytuację, wykorzystując *przeciążanie operatorów* — tworzenie w klasie metod przechwytyujących i przetwarzających wbudowane działania po wykonaniu na instancjach klasy. W tej sytuacji możemy skorzystać z czegoś, co jest chyba najczęściej używaną metodą przeciążania operatorów w Pythonie po metodzie `__init__` — przedstawionej w poprzednim rozdziale `__str__`. Metoda `__str__` wykonywana jest automatycznie za każdym razem, gdy instancja przekształcana jest na swój łańcuch wyświetlania. Ponieważ to właśnie robi wyświetlenie obiektu za pomocą `print`, w rezultacie wyświetlenie obiektu pokazuje cokolwiek, co zwróci jego metoda `__str__`, o ile obiekt ten albo sam ją definiuje, albo dziedziczy po klasie nadzędnej (nazwy z podwójnymi znakami `_` dziedziczone są tak samo jak wszystkie pozostałe).

Zapisana przez nas już w kodzie metoda konstruktora `__init__` także jest przeciążaniem operatorów — wykonywana jest automatycznie w czasie tworzenia w celu zainicjalizowania nowo utworzonej instancji. Konstruktory są jednak tak często spotykane, że wydają się właściwie przypadkiem specjalnym. Bardziej ograniczone metody, takie jak `__str__`, pozwalają nam wejść w określone operacje i udostępnić *wyspecjalizowane działania*, kiedy nasze obiekty wykorzystywane są w tych kontekstach.

Umieścmy to teraz w kodzie. Poniższy przykład rozszerza naszą klasę w celu uzyskania własnego sposobu wyświetlania, wymieniającego atrybuty, gdy instancje klas wyświetlane są jako całość — w miejsce polegania na mniej przydatnym domyślnym sposobie wyświetlania:

```
# Dodanie metody przeciążania operatorów __str__ w celu wyświetlania obiektów

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)
# Dodana metoda
# Wyświetlany łańcuch znaków

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob)
    print(anna)
    print(bob.lastName(), anna.lastName())
    anna.giveRaise(.10)
    print(anna)
```

Warto zwrócić uwagę, że w metodzie `__str__` w celu zbudowania łańcucha wyświetlania wykorzystujemy łańcuch formatujący `%`. Klasy wykorzystują wbudowane obiekty typów oraz operacje takie jak ta w celu wykonania swoich zadań. I znów wszystko, czego nauczyliśmy się już o typach wbudowanych oraz funkcjach, ma zastosowanie do kodu opartego na klasach. Klasa w dużej mierze dodają po prostu dodatkową warstwę *struktury*, która opakowuje funkcje i dane razem oraz obsługuje rozszerzenia.

Zmodyfikowaliśmy również nasz kod testu samosprawdzającego, tak by wyświetlał obiekty w sposób bezpośredni, zamiast pokazywać poszczególne atrybuty. Po wykonaniu kodu dane wyjściowe są teraz o wiele bardziej spójne i zrozumiałe. Wiersze ze znakami „[. . .]” zwracane są przez naszą nową metodę `__str__`, wykonywaną automatycznie przez operacje wyświetlania.

```
[Person: Robert Zielony, 0]  
[Person: Anna Czerwona, 100000]  
Zielony Czerwona  
[Person: Anna Czerwona, 110000]
```

Oto subtelna wskazówka. Jak dowiemy się z następnego rozdziału, podobna metoda przeciążania operatorów `__repr__`, jeśli jest obecna, udostępnia niskopoziomowe wyświetlanie obiektu jak w kodzie. Czasami klasy udostępniają zarówno metodę `__str__`, dla celów wyświetlania przyjaznych dla użytkownika, jak i metodę `__repr__` z dodatkowymi szczegółami, które mogą zobaczyć programiści. Ponieważ wyświetlanie za pomocą `print` wykonuje `__str__`, natomiast sesja interaktywna zwraca wyniki za pomocą `__repr__`, takie rozwiązanie może udostępnić obu typom odbiorców docelowych odpowiedni dla nich rodzaj wyświetlania. Ponieważ nie interesuje nas tym razem wyświetlanie w formacie jak w kodzie, dla naszej klasy wystarczająca będzie metoda `__str__`.

Krok 4. — dostosowanie zachowania do własnych potrzeb za pomocą klas podrzędnych

W tym momencie nasza klasa zawiera większość możliwości programowania zorientowanego obiektowo z Pythona — tworzy instancje, udostępnia działanie w metodach, a nawet wykonuje przeciążanie operatorów w celu przechwytywania operacji wyświetlania za pomocą metody `__str__`. W rezultacie pakuję nasze dane oraz logikę w jeden samodzielny *komponent oprogramowania*, ułatwiając zlokalizowanie kodu, a także sprawiając, że jego modyfikacja w przyszłości będzie prosta. Pozwalając na hermetyzację zachowania, pozwala nam także na faktoryzację kodu w celu uniknięcia jego powtarzalności i związanych z tym problemów z utrzymywaniem kodu w przyszłości.

Jedyną ważną koncepcją z dziedziny programowania zorientowanego obiektowo, której nie implementuje jeszcze nasz kod, jest *dostosowanie do własnych potrzeb za pomocą dziedziczenia*. W pewnym sensie już korzystamy z dziedziczenia, gdyż instancje dziedziczą metody po klasie. By jednak zaprezentować pełne możliwości programowania zorientowanego obiektowo, musimy zdefiniować związek klasa nadziedziona – klasa podrzędna, który pozwoli nam rozszerzać nasze oprogramowanie i zastępować fragmenty odziedziczonego działania. Na tym w końcu polega główna idea programowania zorientowanego obiektowo — dzięki promowaniu modelu opartego na dostosowywaniu do własnych potrzeb już wykonanej pracy pozwala ono drastycznie skrócić czas pisania programów.

Tworzenie klas podrzędnych

W kolejnym kroku zastosujemy nieco metodologii programowania zorientowanego obiektowo w celu wykorzystania naszej klasy `Person` i dostosowania jej do naszych potrzeb za pomocą rozszerzenia naszej hierarchii oprogramowania. Na potrzeby tego przykładu zdefiniujemy klasę podrzędną `Person` o nazwie `Manager`, zastępującą odziedziczoną metodę `giveRaise` jej bardziej wyspecjalizowaną wersją. Nasza nowa klasa rozpoczyna się w następujący sposób:

```
class Manager(Person): # Definiuje klasę podzielną Person
```

Powyższy kod oznacza, że definiujemy nową klasę o nazwie `Manager`, która dziedziczy po klasie nadzielnnej `Person` i może ją dostosowywać do własnych potrzeb. Upraszczając, klasa `Manager` jest prawie tym samym co `Person`, jednak `Manager` posiada własny sposób przyznawania podwyżek.

Na potrzeby przykładu założymy, że kiedy obiekt klasy `Manager` otrzymuje podwyżkę, otrzymuje jak zwykle przekazaną wartość procentową, ale także dodatkowy bonus w domyślnej wysokości 10%. Jeśli zatem podwyżka obiektu `Manager` ustalona została na 10%, tak naprawdę obiekt ten otrzyma 20% (wszelkie podobieństwo do jakichkolwiek osób żyjących lub martwych jest oczywiście przypadkowe). Nasza nowa metoda rozpoczyna się w następujący sposób. Ponieważ ta nowa definicja metody `giveRaise` będzie w drzewie klas bliższa instancjom klasy `Manager` niż oryginalna wersja z klasy `Person`, w rezultacie zastąpi, i tym samym dostosuje do własnych potrzeb, to działanie. Przypomnijmy, że zgodnie z regułami wyszukiwania dziedziczenia, wygrywa wersja nazwy znajdująca się *najniżej* w drzewie:

```
class Manager(Person): # Dziedziczy atrybuty klasy Person
    def giveRaise(self, percent, bonus=.10): # Redefiniuje w celu dostosowania do własnych potrzeb
```

Rozszerzanie metod — niewłaściwy sposób

Istnieją dwa sposoby zapisania tego dostosowania do potrzeb klasy `Manager` w kodzie — właściwy i niewłaściwy. Zaczniemy od *niewłaściwego*, gdyż może on być nieco łatwiejszy do zrozumienia. Niewłaściwy sposób polega na wycięciu i wklejeniu kodu z metody `giveRaise` klasy `Person`, a następnie zmodyfikowaniu go w klasie `Manager`, jak poniżej:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        self.pay = int(self.pay * (1 + percent + bonus)) # Źle — wytnij i wklej
```

Powyższy kod działa zgodnie z planem — kiedy później wywołamy metodę `giveRaise` instancji klasy `Manager`, wykonana zostanie jej własna wersja, dorzucająca dodatkowy bonus. Co zatem jest nie tak z kodem, który działa poprawnie?

Problem jest natury ogólnej — za każdym razem, gdy kod kopujemy za pomocą wycinania i wklejania, w rezultacie *podwajamy* wyciątek, jaki będzie w przyszłości niezbędny do jego utrzymywania. Zastanówmy się: ponieważ skopiowaliśmy oryginalną wersję kodu, gdybyśmy kiedyś musieli zmienić sposób przyznawania podwyżek (co raczej na pewno nastąpi), będziemy musieli zmodyfikować kod w dwóch miejscach, a nie tylko w jednym. Choć jest to niewielki i sztuczny przykład, dobrze reprezentuje on problem uniwersalny. Za każdym razem, gdy kusi nas programowanie za pomocą takiego kopowania kodu, najprawdopodobniej powinniśmy poszukać lepszego rozwiązania.

Rozszerzanie metod — właściwy sposób

To, co tak naprawdę chcemy uzyskać, to *rozszerzenie* w jakiś sposób oryginalnej metody `giveRaise` w miejsce całkowitego jej zastąpienia. *Właściwym* sposobem wykonania tego w Pythonie jest bezpośrednie wywołanie oryginalnej wersji z rozszerzonymi argumentami, jak poniżej:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus) # Dobrze — rozszerzenie oryginału
```

Powyższy kod wykorzystuje fakt, że metodę klasy można zawsze wywołać albo za pomocą *instancji* (zwykły sposób, gdy Python automatycznie przesyła instancję do argumentu `self`), albo za pomocą *klasy* (mniej popularne rozwiązanie, gdzie instancję trzeba przekazać ręcznie). W bardziej symbolicznej terminologii, przypomnijmy, normalne wywołanie metody w postaci:

```
instancja.metoda(argumenty...)
```

jest automatycznie tłumaczone przez Pythona na ten odpowiednik:

```
klasa.metoda(instancja, argumenty...)
```

gdzie klasa zawierająca metodę, która ma być wykonana, ustalana jest za pomocą zastosowania do nazwy metody reguły wyszukiwania dziedziczenia. W kodzie skryptu można zapisać *dowolną* z tych form, jednak jest pomiędzy nimi niewielka różnica — jeśli chcemy wywoływać metodę bezpośrednio za pomocą klasy, musimy pamiętać o ręcznym przekazaniu instancji. Metoda zawsze potrzebuje podmiotu instancji, przekazanego w ten czy inny sposób, a Python udostępnia go automatycznie jedynie w przypadku wywołań wykonywanych za pośrednictwem instancji. W przypadku wywołań wykonywanych za pomocą nazwy klasy musimy sami przesłać instancję do `self`. W przypadku kodu wewnętrz metody, takiej jak `giveRaise`, `self` jest już podmiotem wywołania i tym samym instancją do przekazania.

Wywołanie bezpośrednio za pomocą klasy w rezultacie odwraca dziedziczenie i wysyła wywołanie w górę drzewa klas w celu wykonania określonej wersji. W naszym przypadku możemy wykorzystać tę technikę do wywołania domyślnej metody `giveRaise` klasy `Person`, nawet jeśli została ona ponownie zdefiniowana na poziomie klasy `Manager`. W pewnym sensie *musimy* w ten sposób wywoływać metodę za pośrednictwem `Person`, gdyż `self.giveRaise()` wewnątrz kodu metody `giveRaise` klasy `Manager` wykonałby pętlę — ponieważ `self` jest już obiektem `Manager`, `self.giveRaise()` oznaczałoby `Manager.giveRaise` — i tak aż do wyczerpania dostępnej pamięci.

„Właściwa” wersja może się wydawać niewielką różnicą w kodzie, jednak różnica w zakresie późniejszego *utrzymywania kodu* może być ogromna. Ponieważ logika metody `giveRaise` znajduje się teraz tylko w jednym miejscu (metodzie klasy `Person`), w przyszłości, w miarę ewoluowania potrzeb, będziemy musieli zmodyfikować tylko jedną wersję. I tak naprawdę ta forma w bardziej bezpośredni sposób odpowiada naszym celom — chcemy wykonać standardowe działanie `giveRaise` i po prostu dorzucić dodatkowy bonus. Oto nasz pełny plik modułu z zastosowanym nowym krokiem:

```
# Dodanie dostosowania jednego działania do naszych potrzeb w klasie podrzędnej

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def giveRaise(self, percent, bonus=.10):          # Redefiniowanie na tym poziomie
        Person.giveRaise(self, percent + bonus)         # Wywołanie wersji klasy Person
```

```

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob)
    print(anna)
    print(bob.lastName(), anna.lastName())
    anna.giveRaise(.10)
    print(anna)
    tom = Manager('Tomasz Czarny', 'manager', 50000) # Utworzenie obiektu Manager: __init__
    tom.giveRaise(.10)                                # Wykonanie własnej wersji
    print(tom.lastName())                            # Wykonanie dziedziczonej metody
    print(tom)                                      # Wykonanie dziedziczonej __str__

```

W celu przetestowania naszego dostosowania klasy podzielonej Manager do własnych potrzeb dodaliśmy także kod samosprawdzający tworzący instancję tej klasy, wywołującą jej metody i wyświetlającą ją. Oto dane wyjściowe nowej wersji kodu:

```

[Person: Robert Zielony, 0]
[Person: Anna Czerwona, 100000]
Zielony Czerwona
[Person: Anna Czerwona, 110000]
Czarny
[Person: Tomasz Czarny, 60000]

```

Wszystko wygląda tutaj świetnie. Obiekty bob i anna są takie same jak poprzednio, a kiedy nowy obiekt klasy Manager, tom, otrzymuje 10% podwyżki, tak naprawdę dostaje 20% (jego pensja zwiększa się z 50 000 do 60 000), ponieważ dostosowana do własnych potrzeb metoda giveRaise klasy Manager wykonywana jest tylko dla niego. Warto również zwrócić uwagę, jak wyświetlanie obiektu tom w całości na końcu kodu testu wykorzystuje ładny format zdefiniowany przez metodę `__str__` klasy Person. Obiekty klasy Manager otrzymują kod tej metody, atrybut lastName oraz metodę konstruktora `__init__` „gratis” od klasy Person, dzięki dziedziczeniu.

Polimorfizm w akcji

By zaprezentować nabycie dziedziczonego działania w jeszcze bardziej dobitny sposób, możemy na końcu naszego pliku dodać następujący kod:

```

if __name__ == '__main__':
    ...
    print('--Wszystkie trzy--')
    for object in (bob, anna, tom):
        object.giveRaise(.10)
        print(object)

```

Ogólne przetwarzanie obiektów
Wykonanie metody giveRaise tego obiektu
Wykonanie wspólnej metody __str__

Oto uzyskany rezultat:

```

[Person: Robert Zielony, 0]
[Person: Anna Czerwona, 100000]
Zielony Czerwona
[Person: Anna Czerwona, 110000]
Czarny
[Person: Tomasz Czarny, 60000]
--Wszystkie trzy--
[Person: Robert Zielony, 0]
[Person: Anna Czerwona, 121000]
[Person: Tomasz Czarny, 72000]

```

W dodanym kodzie obiekt jest *albo* instancją klasy Person, *albo* klasy Manager, a Python automatycznie wykonuje kod odpowiedniej metody giveRaise — oryginalną wersję z Person dla

obiektów bob oraz anna, a wersję zmodyfikowaną z Manager dla obiektu tom. Warto samodzielnie prześledzić te wywołania metod w celu zobaczenia, jak Python wybiera właściwą metodę giveRaise dla każdego obiektu.

Jest to po prostu przykład działania zjawiska *polimorfizmu* Pythona, z którym spotkaliśmy się wcześniej — to, co robi giveRaise, uzależnione jest od tego, na czym wykonujemy tę metodę. W powyższym kodzie jest to jeszcze bardziej oczywiste, kiedy metoda ta wybiera z kodu, który zapisaliśmy sami w klasach. Praktycznym efektem tego kodu jest to, iż obiekt anna otrzymuje następne 10% podwyżki, natomiast obiekt tom otrzymuje kolejne 20%, ponieważ metoda giveRaise wybierana jest w oparciu o typ obiektu. Jak już wiemy, polimorfizm jest sercem elastyczności Pythona. Przekazanie dowolnego z trzech naszych obiektów do funkcji wywołującej metodę giveRaise miałoby ten sam efekt — w zależności od typu przekazanego obiektu automatycznie wykonana zostałaby właściwa wersja.

Z drugiej strony, wyświetlanie za pomocą print wykonuje *tę samą* metodę `__str__` dla wszystkich trzech obiektów, ponieważ kod tej metody został napisany tylko raz w klasie Person. Klasa Manager specjalizuje i stosuje kod napisany oryginalnie w Person. Choć przykład ten jest niewielki, wykorzystuje możliwości programowania zorientowanego obiektowo w zakresie dostosowywania kodu do własnych potrzeb i ponownego wykorzystania go. W przypadku klas czasami wydaje się to działać całkowicie automatycznie.

Dziedziczenie, dostosowanie do własnych potrzeb i rozszerzenie

Tak naprawdę klasy mogą być jeszcze bardziej elastyczne, niż sugeruje to nasz przykład. Klasy mogą *dziedziczyć, dostosowywać do własnych potrzeb lub rozszerzać* istniejący kod z klas nadzędnych. Przykładowo, choć w tekście skupiliśmy się na aspekcie dostosowania do własnych potrzeb, do klasy Manager możemy także dodać unikalne metody, których nie ma w klasie Person, jeśli obiekty tej klasy wymagają czegoś zupełnie z innej beczki (odwołanie do *Latającego Cyrku Monty Pythona* jest zamierzzone). Dobra ilustracja to poniższy fragment kodu. Metoda giveRaise redefiniuje tutaj metodę z klasy nadzędnej w celu dostosowania jej do własnych potrzeb, natomiast someThingElse definiuje coś nowego w celu rozszerzenia klasy:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __str__(self): ...

class Manager(Person):
    def giveRaise(self, ...): ...
    def someThingElse(self, ...): ...

tom = Manager()
tom.lastName()                                # Dziedziczenie
tom.giveRaise()                                 # Dostosowanie do własnych potrzeb
tom.someThingElse()                           # Rozszerzenie

# Odziedziczona wprost
# Wersja dostosowana do własnych potrzeb
# Tutaj rozszerzenie
# Odziedziczona przeciążona metoda
```

Dodatkowe metody, jak someThingElse z powyższego kodu, *rozszerzają* istniejące oprogramowanie i dostępne są tylko dla obiektów klasy Manager, a nie dla obiektów klasy Person. Na cele tego przykładu ograniczymy nasze czynności do dostosowywania działania klasy Person do naszych potrzeb za pomocą ponownego zdefiniowania ich w klasie podrzędnej, a nie dodawania czegoś do niej.

Programowanie zorientowane obiektowo — idea

Nasz kod, choć niewielki, jest stosunkowo funkcjonalny. I tak naprawdę ilustruje już najważniejszą kwestię związaną ogólnie z programowaniem zorientowanym obiektowo: programujemy za pomocą *dostosowywania* tego, co zostało już wykonane, *do własnych potrzeb*, a nie za pomocą kopiowania czy modyfikowania istniejącego kodu. Nie jest to zawsze na pierwszy rzut oka oczywiste dla osób początkujących, zwłaszcza biorąc pod uwagę dodatkowe wymagania w zakresie kodu klas. Styl programowania powiązany z klasami może jednak zdecydowanie skrócić czas programowania w porównaniu z innymi rozwiązaniami.

W naszym przykładzie moglibyśmy teoretycznie zaimplementować własne działanie `giveRaise` bez tworzenia klasy podrzędnej, jednak żadna z poniższych opcji nie daje kodu tak optymalnego jak nasz:

- Choć moglibyśmy po prostu napisać kod dla obiektów `Manager` od podstaw, jako nowy, samodzielny kod klasy, musielibyśmy ponownie zaimplementować wszystkie działania klasy `Person`, które dla obiektów `Manager` są takie same.
- Choć moglibyśmy po prostu zmodyfikować istniejącą klasę `Person` w miejscu pod kątem wymagań metody `giveRaise` klasy `Manager`, takie działanie najprawdopodobniej zniszczyłoby miejsca, w których nadal potrzebne jest nam oryginalne działanie klasy `Person`.
- Choć moglibyśmy po prostu skopiować klasę `Person` w całości, zmienić nazwę jej kopii na `Manager` i zmodyfikować jej metodę `giveRaise`, takie działanie wprowadziłoby powtarzalność kodu i w przyszłości podwoiłoby naszą ilość pracy — modyfikacje wprowadzone do klasy `Person` nie zostałyby pobrane automatycznie i musielibyśmy je ręcznie przenieść do kodu klasy `Manager`. Jak zwykle podejście „wytnij i wklej” teraz może się wydawać szybkie, ale podwaja ilość pracy potrzebnej w przyszłości.

Hierarchia kodu, który możemy dostosować do własnych potrzeb, budowana za pomocą klas, jest o wiele lepszym rozwiązaniem w przypadku oprogramowania, które ewoluje z czasem. Żadne inne narzędzia Pythona nie obsługują tego trybu programowania. Ponieważ możemy przycinać i rozszerzać wykonaną już pracę, dodając kod nowych klas podrzędnych, możemy skorzystać z tego, co zostało już zrobione, zamiast za każdym razem rozpoczynać od podstaw, niszcząc to, co już działa, lub wprowadzać kilka kopii kodu, które w przyszłości będziemy musieli aktualnić. Wykonane w poprawny sposób programowanie zorientowane obiektowo jest wielkim sojusznikiem programisty.

Krok 5. — dostosowanie do własnych potrzeb także konstruktorów

Nasz kod w obecnej postaci działa, jednak jeśli przyjrzymy się bliżej jego aktualnej wersji, pewien element może wydać nam się nieco dziwny. Podanie nazwy stanowiska 'manager' dla obiektu `Manager` przy tworzeniu wydaje się bez sensu — nazwa stanowiska wynika przecież z samej klasy. Lepiej byłoby, gdybyśmy mogli w jakiś sposób wypełnić tę wartość automatycznie przy tworzeniu obiektu tej klasy.

Sztuczka, jaką wykorzystamy do poprawienia tego kodu, okazuje się *tą samą*, jaką wykorzystaliśmy w poprzednim podrozdziale. Chcemy dostosować logikę konstruktora klasy `Manager`

do własnych potrzeb w taki sposób, by nazwa stanowiska dodawana była automatycznie. Jeśli chodzi o kod, musimy zdefiniować metodę `__init__` klasy `Manager`, tak by przekazywała ona łańcuch znaków 'manager' za nas. I tak jak w przypadku dostosowania do naszych potrzeb metody `giveRaise`, chcemy wykonać oryginalną metodę `__init__` klasy `Person`, wywołując ją za pośrednictwem nazwy klasy, tak by nadal inicjalizowała ona atrybuty informacji o stanie.

Poniższe rozszerzenie kodu wykona wyznaczone zadanie. Utworzyliśmy nowy konstruktor klasy `Manager` i zmodyfikowaliśmy wywołanie tworzące obiekt `tom`, tak by nie przekazywało ono nazwy stanowiska 'manager'.

```
# Dodanie dostosowania konstruktora w klasie podrzędnej do własnych potrzeb

class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager(Person):
    def __init__(self, name, pay): # Zredefiniowanie konstruktora
        Person.__init__(self, name, 'manager', pay) # Wykonanie oryginalnej metody z łańcuchem
                                                       # 'manager'
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob)
    print(anna)
    print(bob.lastName(), anna.lastName())
    anna.giveRaise(.10)
    print(anna)
    tom = Manager('Tomasz Czarny', 50000) # Nazwa stanowiska nie jest potrzebna:
                                             # jest ona sugerowana (ustawiana) za pomocą klasy
    tom.giveRaise(.10)
    print(tom.lastName())
    print(tom)
```

Do rozszerzenia konstruktora `__init__` wykorzystujemy tutaj tę samą technikę, którą zastosowaliśmy wcześniej w przypadku metody `giveRaise` — wykonanie wersji z klasy nadzędnej za pomocą wywołania za pośrednictwem nazwy klasy i przekazania instancji `self` w sposób jawnny. Choć konstruktor ma dziwną nazwę, rezultat jest identyczny. Ponieważ potrzebowaliśmy wykonać także logikę konstruktora klasy `Person` (w celu zainicjalizowania atrybutów instancji), naprawdę musieliśmy wywołać metodę w ten sposób — w przeciwnym razie instancje nie miałyby dodatkowych żadnych atrybutów.

Wywołanie konstruktora klasy nadzędnej z jego zdefiniowanego na nowo odpowiednika w ten sposób okazuje się w Pythonie bardzo popularnym wzorcem kodu. Sam z siebie Python wykorzystuje dziedziczenie do wyszukania i wywołania tylko jednej metody `__init__` naraz w czasie tworzenia instancji — metody znajdującej się *najniżej* w drzewie klas. Jeśli chcemy, by w czasie konstrukcji obiektu wykonane zostały metody `__init__` znajdujące się wyżej w drzewie

(a zazwyczaj tak jest), musimy wywołać je ręcznie za pośrednictwem nazwy klasy nadrzędnej. Zaletą tego rozwiązania jest to, że możemy w jawnym sposobie zdecydować, który argument przekazać do konstruktora klasy nadrzędnej, a także możemy wybrać, by wcale go nie wywoływać. Niewywołanie konstruktora klasy nadrzędnej pozwala w całości zastąpić jego logikę, zamiast tylko ją rozszerzyć.

Wynik kodu testu samosprawdzającego pliku będzie taki sam jak ostatnio — nie zmieniliśmy działania kodu, a jedynie poddaliśmy go restrukturyzacji w celu pozbycia się powtarzalności.

```
[Person: Robert Zielony, 0]
[Person: Anna Czerwona, 100000]
Zielony Czerwona
[Person: Anna Czerwona, 110000]
Czarny
[Person: Tomasz Czarny, 60000]
```

Programowanie zorientowane obiektowo jest prostsze, niż się wydaje

W swojej obecnej postaci, pomimo rozmiaru, nasze klasy wykorzystują prawie wszystkie najważniejsze koncepcje z dziedziny programowania zorientowanego obiektowo w Pythonie:

- tworzenie instancji — wypełnianie atrybutów instancji,
- metody i działanie — hermetyzacja logiki w metodach klas,
- przeciążanie operatorów — udostępnianie działania operacjom wbudowanym, takim jak wyświetlanie,
- dostosowanie działania do własnych potrzeb — redefiniowanie metod w klasach podrzędnych w celu ich specjalizacji,
- dostosowanie konstruktorów do własnych potrzeb — dodanie logiki inicjalizującej do kroków z klasy nadrzędnej.

Większość z tych koncepcji oparta jest na trzech podstawowych ideach: wyszukiwaniu dziedziczenia atrybutów w drzewie obiektów, specjalnym argumencie `self` w metodach oraz automatycznym udostępnianiu przeciążania operatorów w metodach.

Przy okazji sprawiliśmy także, że nasz kod będzie łatwy do zmodyfikowania w przyszłości, wykorzystując skłonność klas do faktoryzowania kodu w celu zmniejszenia jego powtarzalności. Przykładowo opakowaliśmy logikę w metody i wywołujemy metody klas nadrzędnych z ich rozszerzeń w celu uniknięcia posiadania kilku kopii tego samego kodu. Większość z tych kroków była naturalną konsekwencją strukturyzujących możliwości klas.

Ogólnie rzecz biorąc, na tym właśnie polega programowanie zorientowane obiektowo w Pythonie. Klasy z pewnością mogą stać się większe od tych z przykładu. Istnieją również bardziej zaawansowane zagadnienia związane z klasami, takie jak dekoratory i metaklasy, z którymi spotkamy się w późniejszych rozdziałach. Jeśli jednak chodzi o podstawy, nasze klasy robią wszystko, co należy. Tak naprawdę dla każdej osoby, która pojęła sposób działania napisanych przed chwilą klas, większość kodu z dziedziny programowania zorientowanego obiektowo w Pythonie powinna być teraz na wyciągnięcie ręki.

Inne sposoby łączenia klas

To powiedziawszy, powiniensem również wspomnieć o tym, że choć podstawy programowania zorientowanego obiektowo są w Pythonie proste, w większych programach sztuka polega na tym, w jaki sposób te klasy ze sobą łączymy. W tym przykładzie skupialiśmy się na *dziedziczeniu*, ponieważ mechanizm ten udostępniany jest przez Pythona, jednak programiści czasami łączą klasy również na inne sposoby. Przykładowo popularny wzorzec programowania obejmuje zagnieżdżanie obiektów wewnętrz siebie w celu tworzenia tak zwanych *kompozytów* (ang. *composite*). Wzorzec ten omówimy bardziej szczegółowo w rozdziale 30., który poświęcony jest bardziej projektowaniu niż samemu Pythonowi. Moglibyśmy jednak wykorzystać ideę kompozycji w naszym rozszerzeniu Manager, osadzając klasę Person, zamiast po niej dziedziczyć.

Poniższe rozwiązywanie alternatywne robi to za pomocą wykorzystania metody przeciążania operatorów `__getattr__`, z którą spotkamy się w rozdziale 29., do przechwycenia prób pobierania niezdefiniowanych atrybutów i delegowania ich do osadzonego obiektu za pomocą wbudowanego `getattr`. Metoda `giveRaise` nadal służy do dostosowania do własnych potrzeb, modyfikując argument przekazywany do osadzonego obiektu. W rezultacie klasa Manager staje się warstwą kontrolera przekazującą wywołania *w dół*, do osadzonego obiektu, zamiast *w górę*, do metod klasy nadzędnej:

```
# Alternatywa klasy Manager z osadzaniem

class Person:
    ...to samo...

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay)      # Osadzenie obiektu Person
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Przechwycenie i delegowanie
    def __getattr__(self, attr):
        return getattr(self.person, attr)              # Delegowanie wszystkich pozostałych atrybutów
    def __str__(self):
        return str(self.person)                        # Musi znowu przeciążać operator (w 3.0)

if __name__ == '__main__':
    ...to samo...
```

Tak naprawdę powyższa alternatywa klasy Manager jest reprezentatywna dla ogólnego wzorca kodu zazwyczaj znanego pod nazwą *delegacji* — struktury opartej na kompozycie, która zarządza opakowanym obiektem i przesyła do niego wywołania metod. Wzorzec ten działa w naszym przykładzie, jednak wymaga mniej więcej dwa razy więcej kodu i jest gorzej od dziedziczenia przystosowany do takich typów bezpośredniego dostosowywania do własnych potrzeb, jakie chcieliśmy uzyskać (tak naprawdę żaden rozsądny programista Pythona w praktyce nigdy nie zapisałby tego przykładu w ten sposób, z wyjątkiem, być może, ogólnych celów demonstracyjnych). Obiekt Manager nie jest tutaj tak naprawdę obiektem Person, dlatego potrzebny jest nam dodatkowy kod w celu ręcznego przekazania wywołań metod do osadzonego obiektu. Metody przeciążania operatorów, takie jak `__str__`, musimy zdefiniować ponownie (przynajmniej w wersji 3.0, co wynika z ramki „Przechwytywanie wbudowanych atrybutów w wersji 3.0”), a dodanie nowych działań dla klasy Manager jest o wiele mniej oczywiste, ponieważ informacje o stanie są odległe o jeden poziom.

Mimo to *osadzanie obiektów* i oparte na nim wzorce projektowe mogą być dobrym rozwiązyaniem, kiedy osadzone obiekty wymagają bardziej ograniczonej interakcji z zawierającym je

pojemnikiem niż w przypadku bezpośredniego dostosowania do własnych potrzeb. Warstwa kontrolera, taka jak w tej alternatywnej klasie Manager, mogłaby się na przykład przydać, gdybyśmy chcieli śledzić wywołania do metod innego obiektu lub sprawdzać ich poprawność. I faktycznie, prawie identyczny wzorzec kodu wykorzystamy, kiedy będziemy omawiać *dekatory klas* w dalszej części książki. Co więcej, hipotetyczna klasa Department („dział”), podobna do poniższej, mogłaby *agregować* inne obiekty w celu potraktowania ich jako zbiór. Dodajmy na dole pliku person.py poniższy kod, by to wypróbować:

```
# Agregacja osadzonych obiektów w kompozyt  
...  
bob = Person(...)  
anna = Person(...)  
tom = Manager(...)  
  
class Department:  
    def __init__(self, *args):  
        self.members = list(args)  
    def addMember(self, person):  
        self.members.append(person)  
    def giveRaises(self, percent):  
        for person in self.members:  
            person.giveRaise(percent)  
    def showAll(self):  
        for person in self.members:  
            print(person)  
  
development = Department(bob, anna)      # Osadzenie obiektów w kompozycie  
development.addMember(tom)               # Wykonuje metodę giveRaise osadzonych obiektów  
development.giveRaises(.10)              # Wykonuje metody __str__ osadzonych obiektów  
development.showAll()
```

Co ciekawe, powyższy kod wykorzystuje zarówno dziedziczenie, jak i kompozycję. Department jest kompozytem osadzającym i kontrolującym inne obiekty do zagregowania, natomiast same osadzone obiekty Person oraz Manager wykorzystują dziedziczenie w celu dostosowania do własnych potrzeb. W innym przykładzie graficzny interfejs użytkownika może wykorzystywać dziedziczenie do dostosowania działania lub wyglądu etykiet i przycisków do własnych potrzeb, ale także kompozycję w celu utworzenia większych pakietów osadzonych widgetów, takich jak formularze do wprowadzania danych, kalkulatory czy edytory tekstu. Wykorzystywana struktura klas uzależniona jest od obiektów, których model próbujemy stworzyć.

Zagadnienia związane z projektowaniem, takie jak kompozycja, omówione są w rozdziale 30., dlatego odłożę dalsze omawianie ich na później. Jednak znów, w kontekście podstawowych opcji programowania zorientowanego obiektowo w Pythonie, klasy Person i Manager wykorzystują już wszystko. Po opanowaniu postaw programowania zorientowanego obiektowo tworzenie uniwersalnych narzędzi służących do zastosowania go w łatwiejszy sposób w skryptach jest często naturalnym kolejnym krokiem — a także tematem kolejnego podrozdziału.

Krok 6. — wykorzystywanie narzędzi do introspekcji

Wykonajmy jeszcze jedną, ostatnią zmianę, zanim wrzucimy nasze obiekty do bazy danych. W obecnej postaci nasze klasy są kompletnie i demonstrują większość podstaw programowania zorientowanego obiektowo w Pythonie. Nadal jednak wiążą się z nimi dwa problemy, które najprawdopodobniej powinniśmy rozwiązać, zanim się do nich niepotrzebnie przyzwyczaimy:

Przechwytywanie wbudowanych atrybutów w wersji 3.0

W Pythonie 3.0 (a także w wersji 2.6, jeśli korzystamy z klas w nowym stylu) utworzona przez nas przed chwilą alternatywna wersja klasy `Manager` oparta na delegacji nie będzie w stanie przechwytywać i delegować atrybutów metod przeciążania operatorów takich jak `__str__` bez ponownego ich zdefiniowania. Choć wiemy, że `__str__` jest jedyną taką metodą wykorzystaną w naszym przykładzie, jest to ogólny problem klas opartych na delegacji.

Warto przypomnieć, że operacje wbudowane, takie jak wyświetlanie i indeksowanie, w niejawnym sposób wywołują metody przeciążania operatorów, takie jak `__str__` czy `__getitem__`. W Pythonie 3.0 takie wbudowane operacje nie przekierowują niejawnego pobierania atrybutów za pośrednictwem ogólnych menedżerów atrybutów — nie zostanie wywołana ani metoda `__getattr__` (wykonywana w przypadku niezdefiniowanych atrybutów), ani jej kuzyn `__getattribute__` (wykonywana dla wszystkich atrybutów). Dlatego właśnie musimy ponownie zdefiniować `__str__` w alternatywnej klasie `Manager` — powtarzając się — by dopilnować tego, że wyświetlanie zostanie przekierowane do osadzonego obiektu `Person` po wykonaniu kodu w Pythonie 3.0.

Dzieje się tak dlatego, że klasyczne klasy normalnie wyszukują nazwy metod przeciążania operatorów w instancjach w czasie wykonywania, natomiast klasy w nowym stylu tego nie robią — całkowicie pomijają instancję i szukają takich metod w klasach. W klasach klasycznych z Pythona 2.6 funkcje wbudowane przekierowują atrybuty w sposób ogólny — wyświetlanie przekierowuje na przykład `__str__` za pośrednictwem `__getattr__`. Klasy w nowym stylu także dziedziczą domyślne zachowanie `__str__`, które przysłania `__getattr__`, jednak `__getattribute__` nie przechwytuje nazwy w wersji 3.0.

Jest to zmiana, jednak nie uniemożliwia ona działania kodu. Klasę opartą na delegacji mogą ogólnie redefiniować metody przeciążania operatorów w celu wydelegowania ich do opakowanych obiektów w Pythonie 3.0 — albo ręcznie, albo za pomocą narzędzi bądź klas nadzędnych. Zagadnienie to jest jednak zbyt zaawansowane, by poświęcać mu czas w tym przykładzie, dlatego nie będziemy się nim zajmować zbyt szczegółowo. Powrócimy do niego w omówieniu zarządzania atrybutami w rozdziale 37., a także ponownie w kontekście dekoratorów klas prywatnych z rozdziału 38.

- Po pierwsze, jeśli przyjrzymy się wyświetaniu obiektów w obecnej postaci, zauważymy, że kiedy wyświetlamy obiekt `tom`, klasa `Manager` podpisuje go jako `Person`. Nie jest to niepoprawne z technicznego punktu widzenia, ponieważ `Manager` jest rodzajem wyspecjalizowanego obiektu `Person`. Mimo to lepiej byłoby wyświetlać obiekty z najbardziej uszczerbioną (*najniższą* w drzewie) klasą.
- Po drugie, i chyba ważniejsze, obecny format wyświetlania pokazuje tylko atrybuty podane w naszej metodzie `__str__`, a to może nie obejmować naszych przyszłych zmian. Nie możemy na przykład jeszcze zweryfikować, czy nazwa stanowiska obiektu `tom` została poprawnie ustawiona na '`manager`' przez konstruktor klasy `Manager`, ponieważ metoda `__str__` klasy `Person` nie wyświetla tego pola. Co gorsza, jeśli kiedykolwiek rozszerzymy lub w inny sposób zmodyfikujemy zbiór atrybutów przypisanych do naszych obiektów w metodzie `__init__`, będziemy musieli pamiętać, by uaktualnić również metodę `__str__` nowymi nazwami do wyświetlenia, gdyż inaczej z czasem metody te się rozsynchonizują.

Ostatni punkt oznacza, że znowu zastawiliśmy na siebie pułapkę potencjalnej dodatkowej pracy w przeszłości, wprowadzając *powtarzalność* w kodzie. Ponieważ rozbieżność w metodzie

`str` zostanie odzwierciedlona w danych wyjściowych programu, ta powtarzalność kodu może być jeszcze bardziej oczywista niż inne jej postaci, którymi zajmowaliśmy się wcześniej. Ponadto uniknięcie dodatkowej pracy w przyszłości jest czymś pozytywnym.

Specjalne atrybuty klas

Oba problemy możemy rozwiązać za pomocą narzędzi introspekcji Pythona — specjalnych atrybutów oraz funkcji, które dają nam dostęp do pewnych mechanizmów wewnętrznych implementacji obiektów. Narzędzia te są nieco bardziej zaawansowane i zazwyczaj wykorzystywane są raczej przez osoby tworzące narzędzia dla innych programistów niż przez samych programistów piszących aplikacje. Mimo to podstawowa znajomość niektórych z tych narzędzi

będzie przydatna, ponieważ pozwalają nam one pisać kod przetwarzający klasy na uniwersalne sposoby. W naszym kodzie znajdują się na przykład dwa punkty zaczepienia, które mogą nam pomóc — wprowadzone pod koniec poprzedniego rozdziału:

- Wbudowany atrybut `instancja.__class__` udostępnia łącze z instancją do klasy, z której została ona utworzona. Klasy z kolei mają atrybut `_name_`, tak jak moduły, oraz sekwencję `_bases__` dającą dostęp do klas nadzędnych. Możemy z tego skorzystać tutaj w celu wyświetlenia nazwy klasy, z której utworzono instancję, zamiast nazwy wpisanej na stałe do kodu.
- Wbudowany atrybut `obiekt.__dict__` udostępnia słownik z parami klucz-wartość dla każdego atrybutu dołączonego do obiektu przestrzeni nazw (w tym modułów, klas oraz instancji). Ponieważ jest to słownik, możemy między innymi pobierać jego listy kluczów, indeksować go po kluczu czy wykonywać iterację po kluczach w celu ogólnego przetworzenia wszystkich atrybutów. Możemy z tego skorzystać tutaj, by wyświetlić każdy atrybut instancji, a nie tylko te zapisane na stałe w naszym kodzie wyświetlającym.

Oto jak narzędzia te wyglądają w praktyce w sesji interaktywnej Pythona. Warto zwrócić uwagę na załadowanie klasy `Person` w sesji interaktywnej za pomocą instrukcji `from`. Nazwy klas znajdują się w modułach, skąd są importowane — dokładnie tak jak nazwy funkcji i inne zmienne.

```
>>> from person import Person
>>> bob = Person('Robert Zielony')
>>> print(bob)                                     # Pokazanie __str__ obiektu bob [Person: Robert
Zielony, 0]

>>> bob.__class__                                 # Pokazanie klasy obiektu bob i jej nazwy <class
'person.Person'>
>>> bob.__class__.name__                         # Atrybuty są tak naprawdę kluczami słownika
'Person'                                         # By wymusić listę w wersji 3.0, należy użyć list

>>> list(bob.__dict__.keys())                      # By wymusić listę w wersji 3.0, należy użyć list
['pay', 'job', 'name']

>>> for key in bob.__dict__:
    print(key, '=>', bob.__dict__[key])          # Ręczne indeksowanie

pay => 0
job => None
name => Robert Zielony

>>> for key in bob.__dict__:
```

```
pay => 0
job => None
name => Robert Zielony
```

Jak wspomnieliśmy krótko w poprzednim rozdziale, niektóre atrybuty dostępne z instancji mogą nie być przechowywane w słowniku `__dict__`, jeśli klasa instancji definiuje `__slots__` — omówioną w rozdziałach 30. oraz 31. opcjonalną i dość rzadko spotykaną opcję klas w nowym stylu (i wszystkich klas z Pythona 3.0) przechowującą atrybuty w tablicy. Ponieważ `__slots__` przynależy tak naprawdę do klas, a nie instancji, a poza tym jest ogólnie bardzo rzadko wykorzystywana, możemy ją bezpiecznie zignorować i skupić się na zwykłym słowniku `__dict__`.

Uniwersalne narzędzie do wyświetletlania

Interfejsy te możemy wykorzystać w klasie nadzędnej wyświetlającej dokładne nazwy klas i formatującą wszystkie atrybuty instancji dowolnej klasy. W edytorze tekstu należy otworzyć nowy plik w celu zapisania w nim następującego kodu — będzie to nowy, niezależny moduł o nazwie `classestools.py`, implementujący taką właśnie klasę. Ponieważ jej metoda przeciążania wyświetletlania `__str__` będzie korzystała z uniwersalnych narzędzi do introspekcji, będzie ona działała na każdej *instancji*, bez względu na jej zbiór atrybutów. A ponieważ będzie klasą, automatycznie stanie się uniwersalnym narzędziem do formatowania — dzięki dziedziczeniu można ją wzmieszać w dowolną klasę, która chce skorzystać z jej formatu wyświetletlania. Jako dodatkowy bonus, jeśli kiedykolwiek będziemy chcieli zmodyfikować sposób wyświetletlania instancji, będziemy musieli zmienić jedynie tę klasę, gdyż każda klasa dziedzicząca jej metodę `__str__` automatycznie pobierze nowy format przy następnym wykonywaniu.

```
# Nowy plik classestools.py
# Wybrane narzędzia do obsługi klas

class AttrDisplay:
    """
    Udostępnia dziedziczoną metodę przeciążania wyświetletlania, która pokazuje instancje z ich nazwami klas, a także parę nazwa-wartość dla każdego atrybutu przechowanego w samej instancji (ale nie atrybutów odziedziczonych po klasach).
    Można ją wzmieszać w dowolną klasę i będzie działała na dowolnej instancji.
    """

    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__):
            attrs.append('%s=%s' % (key, getattr(self, key)))
        return ', '.join(attrs)
    def __str__(self):
        return '[%s: %s] % (self.__class___.name__, self.gatherAttrs())

if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self):
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count+1
            TopTest.count += 2
    class SubTest(TopTest):
        pass

    X, Y = TopTest(), SubTest()
    print(X)                                     # Pokazanie wszystkich atrybutów instancji
    print(Y)                                     # Pokazanie najwyższej nazwy klasy
```

Warto zwrócić tutaj uwagę na łańcuchy znaków dokumentacji. Ponieważ jest to narzędzie ogólnego przeznaczenia, chcemy dodać nieco dokumentacji, by potencjalni użytkownicy mogli ją przeczytać. Jak widzieliśmy w rozdziale 15., łańcuchy znaków dokumentacji można umieścić na górze prostych funkcji i modułów, a także na początku klas i ich metod. Funkcja `help` oraz narzędzie PyDoc pobierają i wyświetlają je automatycznie (łańcuchom znaków dokumentacji przyjrzymy się ponownie w rozdziale 28.).

Po wykonaniu kod samosprawdzający modułu tworzy dwie instancje i wyświetla je. Zdefiniowana tutaj metoda `__str__` pokazuje klasę instancji oraz wszystkie jej nazwy i wartości atrybutów w kolejności zgodnej z posortowanymi nazwami atrybutów.

```
C:\misc> classtools.py
[TopTest: attr1=0, attr2=1]
[SubTest: attr1=2, attr2=3]
```

Atrybuty instancji a atrybuty klas

Osoby, które wystarczająco długo przyglądały się kodowi testu samosprawdzającego, zauważły zapewne, że jego klasa wyświetla tylko *atrybuty instancji*, dołączone do obiektu `self` na dole drzewa dziedziczenia — właśnie to zawiera słownik `__dict__` dla `self`. Zamierzoną konsekwencją jest zatem to, że nie widzimy atrybutów odziedziczonych przez instancje po klasach znajdujących się wyżej w drzewie (w przypadku kodu powyższego testu samosprawdzającego na przykład atrybutu `count`). Odziedziczone atrybuty klas dołączane są jedynie do klas — nie są kopowane aż do instancji.

Jeśli jednak kiedykolwiek będziemy chcieli dołączyć wyświetlanie odziedziczonych atrybutów, możemy za pomocą łącza `__class__` wspiąć się do klasy instancji, tam użyć `__dict__` w celu pobrania atrybutów klasy, a następnie wykonać iterację po atrybutach `__bases__` klasy w celu wsparcia się do jeszcze wyższych klas nadzędnych (powtarzając to w miarę potrzeby). Dla fanów prostego kodu wykonanie wywołania wbudowanej funkcji `dir` będzie miało ten sam efekt, ponieważ `dir` uwzględnia w posortowanej liście wyników także zmienne odziedziczone.

```
>>> from person import Person
>>> bob = Person('Robert Zielony')

# W Pythonie 2.6:

>>> bob.__dict__.keys()                               # Tylko atrybuty instancji
['pay', 'job', 'name']

>>> dir(bob)                                         # + odziedziczone atrybuty z klas
['__doc__', '__init__', '__module__', '__str__', 'giveRaise', 'job', 'lastName',
 'name', 'pay']

# W Pythonie 3.0:

>>> list(bob.__dict__.keys())                         # W 3.0 keys jest widokiem, a nie listą
['pay', 'job', 'name']

>>> dir(bob)                                         # 3.0 obejmuje metody typu klas
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', ...pominęto kilka
 ↴wierszy... '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', 'giveRaise', 'job', 'lastName', 'name', 'pay']
```

Wynik jest różny dla Pythona w wersji 2.6 i 3.0, ponieważ `dict.keys` z 3.0 nie jest listą, a funkcja `dir` w tej wersji zwraca dodatkowe atrybuty implementacyjne typu klasy. Funkcja ta zwraca w Pythonie 3.0 więcej, ponieważ wszystkie klasy są klasami w nowym stylu i dziedziczą duży zbiór nazw przeciążania operatorów po typie klasy. Tak naprawdę w wyniku funkcji `dir` będziemy najprawdopodobniej chcieli odfiltrować większość nazw z `__X__` z wyników, ponieważ są to wewnętrzne szczegóły implementacyjne, a nie coś, co normalnie chcielibyśmy wyświetlić.

Chcąc zaoszczędzić nieco miejsca, zostawimy na razie opcjonalne wyświetlanie atrybutów klas za pomocą wspinania się w górę drzewa klas lub `dir` jako sugerowane eksperymenty. Więcej wskazówek dotyczących tych kwestii znajdziesz w programie wspinającym się w górę drzewa dziedziczenia `classtree.py` z rozdziału 28., a także w programach wymieniających atrybuty z pliku `lister.py` z rozdziału 30.

Rozważania na temat nazw w klasach narzędzi

Ostatnia drobnostka: ponieważ nasza klasa `AttrDisplay` z modułu `clastools` jest uniwersalnym narzędziem zaprojektowanym z myślą o umieszaniu w dowolne klasy, musimy być świadomi ryzyka potencjalnych i niezamierzonych konfliktów nazw zmiennych z klasami klienta. W obecnej formie kodu założyłem, że klasy klienta mogą chcieć skorzystać z metod `__str__` oraz `gatherAttrs`, jednak druga z nich może dawać więcej, niż oczekuje tego klasa podzielona. Jeśli klasa podzielona nieświadomie zdefiniuje własną zmienną `gatherAttrs`, zepsuje to działanie naszej klasy, ponieważ znajdująca się niżej wersja z klasy podzielonej zostanie użyta w miejscu naszej.

By się o tym przekonać, wystarczy dodać `gatherAttrs` do klasy `TopTest` w kodzie samosprawdzającym pliku. O ile nowa metoda nie będzie identyczna lub celowo nie będzie dostosowywać oryginału do własnych potrzeb, nasza klasa narzędzi nie będzie już działała zgodnie z planem:

```
class TopTest(AttrDisplay):
    ...
    def gatherAttrs(self): # Zastępuje metodę z klasy AttrDisplay!
        return 'Mielonka'
```

Niekoniecznie jest to czymś złym — czasami chcemy, by inne metody były dostępne dla klas podzielonych w celu bezpośredniego wywołania lub dostosowania do własnych potrzeb. Jeśli jednak naprawdę chcielibyśmy tylko udostępnić metodę `__str__`, takie rozwiązanie nie jest idealne.

By zminimalizować szanse wystąpienia takich konfliktów między nazwami, programiści Pythona często poprzedzają metodę, która nie jest przeznaczona do użycia na zewnątrz, *pojedynczym znakiem „_”*. W naszym przypadku da nam to metodę `_gatherAttrs`. Nie jest to rozwiązanie idealne (co, jeśli inna klasa także definiuje `_gatherAttrs`?), ale zazwyczaj wystarczające i jest to popularna w Pythonie konwencja nazewnictwa metod wewnętrznych dla klas.

Lepszym i rzadziej wykorzystywanym rozwiązaniem byłoby użycie *dwóch znaków „_”* przed nazwą metody, jak w `_gatherAttrs`. Python automatycznie rozszerza takie nazwy dla nas, tak by zawierały nazwę klasy zawierającej, przez co stają się one prawdziwie unikalne. Opcja ta znana jest pod nazwą *atrybutów pseudoprivałnych klas* i omówimy ją w rozdziale 30. Na razie obie nasze metody będziemy udostępniać.

Ostateczna postać naszych klas

By teraz użyć tego uniwersalnego narzędzia w naszych klasach, wystarczy zaimportować je z modułu, wzmieszać do naszej klasy najwyższego poziomu za pomocą dziedziczenia i pozbyć się bardziej uszczegółowionego kodu utworzonej wcześniej metody `__str__`. Nowa metoda przeciążająca wyświetlanie będzie dziedziczona przez instancje klasy Person, a także klasy Manager. Manager otrzymuje metodę `__str__` z klasy Person, która teraz uzyskuje ją z klasy AttrDisplay zapisanej w innym module. Oto ostateczna wersja naszego pliku `person.py` po zastosowaniu tych zmian:

```
# Plik person.py (wersja ostateczna)

from classtools import AttrDisplay          # Użycie uniwersalnego narzędzia do wyświetlania

class Person(AttrDisplay):
    """
    Tworzy i przetwarza rekordy osób
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):                      # Zakłada, że nazwisko jest na końcu
        return self.name.split()[-1]
    def giveRaise(self, percent):             # Procent musi się mieścić między 0 a 1
        self.pay = int(self.pay * (1 + percent))

class Manager(Person):
    """
    Dostosowana do własnych potrzeb klasa Person ze specjalnymi wymaganiami
    """
    def __init__(self, name, pay):
        Person.__init__(self, name, 'manager', pay)
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Robert Zielony')
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(bob)
    print(anna)
    print(bob.lastName(), anna.lastName())
    anna.giveRaise(.10)
    print(anna)
    tom = Manager('Tomasz Czarny', 50000)
    tom.giveRaise(.10)
    print(tom.lastName())
    print(tom)
```

Ponieważ jest to ostateczna wersja kodu, dodaliśmy kilka *komentarzy* dokumentujących naszą pracę — łańcuchy znaków dokumentacji w przypadku opisów funkcjonalności i komentarze ze znakami `#` z mniejszymi uwagami, zgodnie z zalecanymi praktykami. Po wykonaniu kodu widzimy teraz wszystkie atrybuty naszych obiektów, a nie tylko te zapisane na stałe w oryginalnej metodzie `__str__`. Rozwiążany został także nasz ostatni problem — AttrDisplay pobiera nazwy klas bezpośrednio z instancji `self`, a każdy obiekt pokazany jest z nazwą najbliższej (znajdującej się najniżej w drzewie) klasy — obiekt `tom` jest teraz wyświetlany jako Manager, a nie Person i możemy wreszcie zweryfikować, że jego nazwa stanowiska została poprawnie wypełniona przez konstruktor klasy Manager.

```
C:\misc> person.py
[Person: job=None, name=Robert Zielony, pay=0]
[Person: job=programista, name=Anna Czerwona, pay=100000]
Zielony Czerwona
[Person: job=programista, name=Anna Czerwona, pay=110000]
Czarny
[Manager: job=manager, name=Tomasz Czarny, pay=60000]
```

Oto bardziej przydatny sposób wyświetlania, o jaki nam chodziło. Z odleglejszej perspektywy nasza klasa wyświetlająca atrybuty stała się *uniwersalnym narzędziem*, które możemy wzmieścić w dowolną klasę za pomocą dziedziczenia w celu wykorzystania zdefiniowanego przez nią formatu wyświetlania. Co więcej, wszystkie klienci automatycznie pobiorą przyszłe modyfikacje naszego narzędzia. W dalszej części książki spotkamy się z koncepcjami związanymi z klasami, które mają jeszcze większe możliwości — takimi jak dekoratory oraz metaklasy. Obok narzędzi do introspekcji Pythona pozwalają nam one pisać kod rozszerzający klasy i zarządzający nimi w ustrukturyzowany, łatwy do utrzymania w przyszłości sposób.

Krok 7. i ostatni — przechowanie obiektów w bazie danych

W tym momencie nasza praca jest prawie gotowa. Mamy teraz *system składający się z dwóch modułów*, który nie tylko implementuje nasze oryginalne cele projektowe, czyli reprezentowanie osób, ale także udostępnia uniwersalne narzędzie do wyświetlania atrybutów, które możemy w przyszłości wykorzystać w innych programach. Umieszczając kod funkcji oraz klas w plikach modułów, uzyskaliśmy to, że w naturalny sposób obsługują one ponowne wykorzystanie kodu. A zapisując program w postaci klas, uzyskaliśmy naturalną obsługę rozszerzania.

Choć jednak nasze klasy działają zgodnie z planem, tworzone przez nie obiekty nie są prawdziwymi rekordami bazy danych. Oznacza to, że jeśli wyłączymy Pythona, nasze instancje znikną — są one tymczasowymi obiektami w pamięci i nie są przechowywane w bardziej trwałym medium, takim jak plik, dlatego nie będą dostępne dla przyszłych wykonień programu. Okazuje się, że uczynienie obiektów instancji bardziej trwałymi jest bardzo proste — dzięki opcji Pythona znanej pod nazwą *trwałość obiektów* (ang. *object persistence*). W ten sposób obiekty będą istniały po zakończeniu działania programu, który je utworzył. W ostatnim kroku niniejszego przykładu sprawimy, że nasze obiekty będą prawdziwie trwałe.

Obiekty pickle i shelve

Trwałość obiektów zaimplementowana została za pomocą trzech modułów biblioteki standardej, dostępnych w każdej wersji Pythona:

pickle

Serializuje dowolne obiekty Pythona na łańcuchy bajtowe i odwrotnie.

dbm (w Pythonie 2.6 nosi nazwę anydbm)

Implementuje system plików dostępu według klucza, służący do przechowywania łańcuchów znaków.

shelve

Wykorzystuje dwa inne moduły w celu przechowania w pliku obiektów Pythona — dostępnych po kluczach.

Z modułami tymi spotkaliśmy się krótko w rozdziale 9., kiedy omawialiśmy podstawy plików. Udostępniają one opcje przechowywania danych o sporych możliwościach. Choć nie jesteśmy w stanie w pełni przedstawić ich zastosowań w tym przykładzie czy książce, są one na tyle proste, że krótkie wprowadzenie wystarczy, by zacząć pracę z nimi.

Moduł `pickle` jest rodzajem bardzo uniwersalnego narzędzia do formatowania i deformatowania obiektów. Po podaniu prawie dowolnego obiektu Pythona z pamięci jest na tyle sprytny, by potrafić przekształcić ten obiekt na łańcuch bajtów, który później można wykorzystać do odtworzenia oryginalnego obiektu w pamięci. Moduł `pickle` obsługuje prawie każdy obiekt, jaki możemy utworzyć — listy, słowniki, ich zagnieżdżone kombinacje i instancje klasy. Szczególnie użytecznymi obiektami dla serializacji są zwłaszcza te ostatnie, ponieważ udostępniają one zarówno dane (atrybuty), jak i działania (metody). Tak naprawdę kombinacja ta jest mniej więcej odpowiednikiem „rekordów” i „programów”. Ponieważ moduł `pickle` jest tak uniwersalny, może zastąpić dodatkowy kod, który w innym przypadku musielibyśmy napisać w celu utworzenia i przeanalizowania składniowo własnych reprezentacji pliku tekstowego dla naszych obiektów. Przechowując łańcuch serializacji w pliku, w rezultacie czynimy go trwałym. Później wystarczy załadować obiekt i zdeserializować go w celu jego odtworzenia.

Choć użycie samego modułu `pickle` w celu przechowania obiektów w zwykłych, płaskich plikach i późniejszego ich załadowania jest proste, moduł `shelve` udostępnia dodatkową warstwę struktury, która pozwala przechowywać zserializowane obiekty po *kluczu*. Moduł `shelve` przekłada obiekt na jego zserializowany za pomocą `pickle` łańcuch, a następnie przechowuje ten łańcuch pod kluczem w pliku `dbm`. Kiedy później go ładujemy, `shelve` pobiera zserializowany łańcuch po kluczu i odtwarza oryginalny obiekt w pamięci za pomocą `pickle`. To wszystko jest niezłą sztuczką, jednak dla naszego skryptu zserializowane obiekty wyglądają tak jak *słownik* — indeksuje się je po kluczu w celu pobrania, przypisuje do kluczy w celu przechowania i korzysta z narzędzi słowników, takich jak `len`, `in` czy `dict.keys`, w celu otrzymania informacji. Zserializowane za pomocą `shelve` obiekty automatycznie odwzorowują operacje na słownikach na obiekty przechowane w pliku.

Tak naprawdę dla naszego skryptu jedyna różnica w kodzie pomiędzy `shelve` a normalnym słownikiem polega na tym, że zserializowane za pomocą `shelve` obiekty musimy początkowo otwierać, a po wprowadzeniu zmian zamazywać. Rezultat jest taki, że moduł `shelve` udostępnia prostą bazę danych służącą do przechowywania i pobierania własnych obiektów Pythona po kluczu, tym samym czyniąc je trwałymi pomiędzy wywołaniami programu. Nie obsługuje narzędzi tworzenia zapytań, takich jak SQL, i brakuje mu pewnych zaawansowanych opcji dostępnych w profesjonalnych bazach danych (takich jak prawdziwe przetwarzanie transakcji), jednak własne obiekty Pythona przechowane za pomocą `shelve` można przetwarzać z pełną mocą języka Python po uprzednim pobraniu ich z powrotem za pomocą klucza.

Przechowywanie obiektów w bazie danych za pomocą `shelve`

Korzystanie z modułów `pickle` oraz `shelve` jest zagadnieniem stosunkowo zaawansowanym, dlatego nie będziemy się tutaj zagłębiać we wszystkie szczegóły. Więcej informacji na ich temat można znaleźć w dokumentacji biblioteki standardowej, a także książkach skupiających się na aplikacjach, takich jak *Programming Python*. Wszystko to jednak jest łatwiejsze w kodzie Pythona niż tekstowym opisie, dlatego zabierzmy się za kod.

Napiszemy nowy skrypt, który serializuje obiekty naszych klas za pomocą modułu `shelve`. W edytorze tekstu należy otworzyć nowy plik, który nazwiemy `makedb.py`. Ponieważ jest to nowy plik, będziemy musieli zaimportować nasze klasy w celu utworzenia kilku instancji do przechowania. Wcześniej, w sesji interaktywnej, importowaliśmy klasę za pomocą instrukcji `from`, jednak tak jak w przypadku funkcji i innych zmiennych, zawsze istnieją dwa sposoby załadowania z pliku (nazwy klas są takimi samymi zmiennymi jak wszystkie pozostałe i w tym kontekście nie ma nic magicznego):

```
import person  
bob = person.Person(...)  
  
from person import Person  
bob = Person(...)
```

Załadowanie klasy za pomocą instrukcji import
Przejście nazwy modułu

Załadowanie klasy za pomocą instrukcji from
Bezpośrednie użycie zmiennej

W celu załadowania klasy w naszym skrypcie skorzystamy z instrukcji `from`, jednak zrobimy tak tylko dla tego, że wymaga to mniej pisania. Należy skopiować lub ponownie wpisać ten kod w celu utworzenia w nowym skrypcie instancji naszych klas, byśmy mieli co przechowywać (to prosta demonstracja, więc nie będziemy się tu przejmować powtarzalnością kodu testowego). Kiedy mamy już instancje, przechowanie ich za pomocą `shelve` jest prawie trywialne. Po prostu importujemy moduł `shelve`, otwieramy nowy obiekt `shelve` z nazwą pliku zewnętrznego, przypisujemy w nim obiekty do kluczy, a następnie po zakończeniu zamkamy obiekt `shelve`, ponieważ wprowadziliśmy zmiany.

Plik makedb.py: przechowanie obiektów klasy Person w bazie danych modułu shelve

```
from person import Person, Manager  
bob = Person('Robert Zielony')  
anna = Person('Anna Czerwona', job='programista', pay=100000)  
tom = Manager('Tomasz Czarny', 50000)  
  
import shelve  
db = shelve.open('persondb')  
for object in (bob, anna, tom):  
    db[object.name] = object  
db.close()
```

Załadowanie naszych klas
Ponowne utworzenie obiektów do przechowania

Nazwa pliku, w którym przechowywane są obiekty
Użycie atrybutu name obiektu jako klucza
Przechowanie obiektu w pliku shelve po kluczowi
Zamknięcie po wprowadzeniu zmian

Warto zwrócić uwagę na to, w jaki sposób przypisujemy obiekty do obiektu `shelve`, wykorzystując do tego ich własne atrybuty `name` w postaci kluczy. Robimy to dla wygody — w obiekcie `shelve` kluczem może być dowolny łańcuch znaków, w tym taki, który utworzymy jako unikalny za pomocą narzędzi takich, jak identyfikator procesu czy data i godzina (dostępnych w modułach biblioteki standardowej `os` oraz `time`). Jedyna zasada jest taka, że klucze muszą być łańcuchami znaków i powinny być unikalne, ponieważ możemy przechować tylko jeden obiekt na klucz (choć obiekt ten może być listą czy słownikiem zawierającym wiele obiektów). Wartości przechowywane pod kluczami mogą jednak być prawie dowolnego rodzaju obiektami Pythona — typami wbudowanymi, jak łańcuchy znaków, listy czy słowniki, a także instancjami klas zdefiniowanych przez użytkownika i zagnieżdżonymi kombinacjami wszystkich tych kategorii.

I to tyle — jeśli skrypt nie zwraca żadnych danych wyjściowych po wykonaniu, oznacza to najprawdopodobniej, że zadziałał. Nic nie wyświetlamy, a jedynie tworzymy i przechowujemy obiekty.

```
C:\misc> makedb.py
```

Interaktywne badanie obiektów `shelve`

W tym momencie w katalogu bieżącym mamy jeden lub kilka prawdziwych plików, których nazwy rozpoczynają się od „`persondb`”. Same tworzone pliki mogą się różnić dla poszczególnych platform i, tak jak funkcja wbudowana `open`, nazwa pliku w `shelve.open()` podawana jest względem aktualnego katalogu roboczego, o ile nie zawiera ścieżki do katalogu. Bez względu na miejsce przechowania pliki te implementują plik z dostępem po kluczu, który zawiera zserializowaną reprezentację naszych trzech obiektów Pythona. Nie należy usuwać tych plików — są one naszą bazą danych i będziemy je musieli skopiować lub przenieść, kiedy będziemy tworzyć kopię zapasową bazy.

Można sprawdzić zawartość plików `shelve`, jeśli ktoś ma ochotę — albo z Eksploratora Windows, albo powłoki Pythona, jednak są to pliki binarne, a większa część ich zawartości ma niewielki sens poza kontekstem modułu `shelve`. W Pythonie 3.0 i bez zainstalowanego żadnego dodatkowego oprogramowania nasza baza danych przechowana zostaje w trzech plikach. W wersji 2.6 będzie to tylko jeden plik, `persondb`, ponieważ moduł rozszerzenia `bsddb` jest w Pythonie instalowany automatycznie dla plików `shelve`; w wersji 3.0 `bsddb` jest dodatkiem zewnętrznym na licencji open source.

```
# Moduł wymieniający zawartość katalogu — sprawdzenie, czy pliki są obecne

>>> import glob
>>> glob.glob('person*')
['person.py', 'person.pyc', 'persondb.bak', 'persondb.dat', 'persondb.dir']

# Wpisanie pliku — tryb tekstowy dla łańcucha znaków, tryb binarny dla bajtów

>>> print(open('persondb.dir').read())
'Anna Czerwona', (512, 104)
...reszta zawartości pominięta...

>>> print(open('persondb.dat', 'rb').read())
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01)q\x02(X\x03\x00\x00\x00payq\x03K...
...reszta zawartości pominięta...
```

Zawartość tę da się odszyfrować, jednak może ona być różna dla różnych platform i trudno to uznać za przyjazny dla użytkownika interfejs bazy danych! W celu lepszego zweryfikowania naszej pracy możemy napisać kolejny skrypt albo pobawić się plikiem `shelve` w sesji interaktywnej. Ponieważ obiekty `shelve` są obiektami Pythona zawierającymi inne obiekty Pythona, możemy je przetwarzać za pomocą normalnej składni i trybów programowania tego języka. Poniżej sesja interaktywna staje się *klientem bazy danych*:

```
>>> import shelve
>>> db = shelve.open('persondb')                                     # Ponowne otwarcie pliku shelve

>>> len(db)                                                       # Przechowano 3 „rekordy”
3
>>> list(db.keys())                                              # keys w celu zindeksowania
['Anna Czerwona', 'Robert Zielony', 'Tomasz Czarny']             # list w celu uzyskania listy w 3.0

>>> bob = db['Robert Zielony']                                    # Pobranie obiektu bob po klucz
>>> print(bob)                                                    # Wykonuje __str__ z klasy AttrDisplay
[Person: job=None, name=Robert Zielony, pay=0]

>>> bob.lastName()                                               # Wykonuje lastName z klasy Person
'Zielony'
```

```

>>> for key in db:                                     # Iteracja, pobranie, wyświetlenie
    print(key, '=>', db[key])

Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=100000]
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

>>> for key in sorted(db):                         # Iteracja po posortowanych kluczach
    print(key, '=>', db[key])

Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=100000]
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

```

Warto zauważyć, że nie musimy tutaj importować naszych klas Person czy Manager w celu załadowania lub wykorzystania przechowywanych obiektów. Możemy na przykład swobodnie wywołać metodę lastName obiektu bob i automatycznie otrzymać własny format wyświetlania, nawet jeśli jego klasa Person nie znajduje się tutaj w naszym zakresie. Dzieje się tak, ponieważ gdy Python serializuje instancję klasy, zapisuje atrybuty `self` tej instancji, a także nazwę klasy, z której została ona utworzona, oraz modułu, w którym klasa ta się znajduje. Kiedy później pobieramy obiekt bob z pliku shelf i poddajemy deserializacji, Python automatycznie ponownie importuje klasę i łączy z nią obiekt bob.

Zaletą tego rozwiązania jest to, że instancje klas automatycznie uzyskują wszystkie zachowania swoich klas, kiedy w przyszłości zostają załadowane. Klasy musimy importować tylko w celu utworzenia nowych instancji, a nie przetwarzania istniejących. Choć jest to celowe rozwiązanie, ma ono nieco mieszane konsekwencje:

- *Wadą* jest to, że klasy i ich pliki modułów trzeba mówić importować, kiedy instancja zostanie później załadowana. Z formalnego punktu widzenia klasy podatne na serializację muszą być zapisane w kodzie na najwyższym poziomie pliku modułu dostępnego z katalogu wymienionego w ścieżce wyszukiwania modułów `sys.path` (i nie powinny się znajdować w metodzie `__main__` modułu większości skryptów, o ile przy użyciu nie znajdują się one zawsze w tym module). Z powodu takich wymagań w zakresie zewnętrznych plików modułów niektóre aplikacje decydują się serializować prostsze obiekty, takie jak słowniki czy listy, zwłaszcza gdy mają one zostać przetransferowane za pomocą Internetu.
- *Zaletą* jest to, że modyfikacje w pliku z kodem źródłowym klasy są pobierane automatycznie, gdy instancje klasy zostaną ponownie załadowane. Często nie ma potrzeby uaktualniać samych przechowywanych obiektów, ponieważ uaktualnienie kodu ich klasy modyfikuje ich działanie.

Obiekty `shelve` mają także dobrze znane ograniczenia (o kilku z nich wspomnialiśmy na końcu niniejszego rozdziału przy omawianiu sugerowanych baz danych). W przypadku prostego przechowywania obiektów moduły `shelve` i `pickle` są jednak niezwykle łatwymi w użyciu narzędziami.

Uaktualnianie obiektów w pliku `shelve`

Czas na ostatni skrypt. Napiszemy program uaktualniający instancję (rekord) z każdym wykonaniem w celu udowodnienia, że nasze obiekty są naprawdę *trwałe* (to znaczy ich wartości bieżące są dostępne z każdym wykonaniem programu Pythona). Poniższy plik, `updatedb.py`, wyświetla zawartość bazy danych i za każdym razem daje podwyżkę jednemu z naszych

obiektów. Jeśli prześledzimy, co się w nim dzieje, zauważymy, że mnóstwo funkcjonalności otrzymujemy gratis — wyświetlanie obiektów automatycznie wykorzystuje ogólną metodę przeciążania operatorów `__str__`, a podwyżki przyznaje się, wywołując napisaną wcześniej metodę `giveRaise`. Wszystko to „po prostu działa” dla obiektów opartych na modelu dziedziczenia z programowania zorientowanego obiektywem, nawet jeśli obiekty te znajdują się w pliku.

```
# Plik updatedb.py: uaktualnienie obiektu klasy Person w bazie danych

import shelve
db = shelve.open('persondb')                                # Ponowne otwarcie pliku shelve z tą samą nazwą pliku

for key in sorted(db):
    print(key, '\t=>', db[key])                            # Iteracja w celu wyświetlenia obiektów bazy danych
                                                            # Wyświetlenie za pomocą własnego formatu

anna = db['Anna Czerwona']                                 # Indeksowanie za pomocą klucza w celu pobrania
anna.giveRaise(.10)                                         # Uaktualnienie w pamięci za pomocą metody klasy
db['Anna Czerwona'] = anna                               # Przypisanie do klucza w celu uaktualnienia w pliku shelve
db.close()                                                 # Zamknięcie po wprowadzeniu zmian
```

Ponieważ powyższy skrypt wyświetla po uruchomieniu zawartość bazy danych, musimy wywołać go kilka razy, by zobaczyć, jak zmieniają się nasze obiekty. Poniżej widać jego działanie — wyświetlenie wszystkich rekordów i zwiększenie atrybutu `pay` obiektu `anna` z każdym wykonaniem (dla obiektu `anna` jest to dość przyjemny skrypt...).

```
c:\misc> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=100000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

c:\misc> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=110000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

c:\misc> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=121000]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]

c:\misc> updatedb.py
Robert Zielony => [Person: job=None, name=Robert Zielony, pay=0]
Anna Czerwona => [Person: job=programista, name=Anna Czerwona, pay=133100]
Tomasz Czarny => [Manager: job=manager, name=Tomasz Czarny, pay=50000]
```

I znów: to, co widzimy tutaj, jest rezultatem działania narzędzi `shelve` i `pickle` otrzymanych od Pythona oraz działań, które sami zapisaliśmy w kodzie naszych klas. Ponownie możemy zweryfikować w sesji interaktywnej, czy nasz skrypt działa (sesja będzie odpowiednikiem klienta bazy danych):

```
c:\misc> python
>>> import shelve
>>> db = shelve.open('persondb')                                # Ponowne otwarcie bazy danych
>>> rec = db['Anna Czerwona']                                  # Pobranie obiektu po kluczu
>>> print(rec)
[Person: job=programista, name=Anna Czerwona, pay=146410]
>>> rec.lastName()
'Czerwona'
>>> rec.pay
146410
```

Kolejny przykład trwałości obiektów w tej książce można znaleźć w rozdziale 30. w ramce zatytułowanej „Znaczenie klas oraz trwałości obiektów”. Przechowuje on nieco większy obiekt kompozytowy w pliku za pomocą pickle, a nie shelve, jednak rezultat jest podobny. Więcej informacji na temat wykorzystywania modułów pickle i shelve można znaleźć w innych książkach lub dokumentacji Pythona.

Przyszłe kierunki rozwoju

I to właściwie koniec niniejszego przykładu. Zaprezentowaliśmy wszystkie podstawy programowania zorientowanego obiektywem w Pythonie w akcji, nauczyliśmy się sposobów unikania powtarzalności kodu, a także — powiązanych z tą kwestią — problemów z utrzymywaniem kodu w przyszłości. Zbudowaliśmy klasy wykonujące prawdziwe zadania. Jako dodatkowy bonus zbudowaliśmy także prawdziwe rekordy bazy danych, przechowując je w pliku shelve Pythona, tak by informacje te były trwałe.

Moglibyśmy oczywiście zrobić jeszcze więcej. Przykładowo moglibyśmy rozszerzyć nasze klasy w taki sposób, by stały się bardziej realistyczne, czy dodać do nich nowe rodzaje zachowania. Dawanie podwyżki powinno na przykład sprawdzać, czy jej wysokość mieści się między zero a jeden — takie rozszerzenie dodamy, gdy w dalszej części książki spotkamy dekoratory. Moglibyśmy także przekształcić ten przykład w osobistą bazę danych kontaktów, modyfikując informacje o stanie przechowywane w obiektach, a także metody klas wykorzystywane do ich przetwarzania. Pozostawimy to jednak jako sugerowane ćwiczenie otwarte, w którym można w pełni wykorzystać swoją wyobraźnię.

Moglibyśmy także rozszerzyć zakres wykorzystywanych narzędzi wbudowanych w Pythona lub swobodnie dostępnych w świecie oprogramowania open source:

Graficzne interfejsy użytkownika

W obecnej postaci możemy jedynie przetwarzać naszą bazę danych za pomocą interfejsu wiersza poleceń oraz skryptów. Moglibyśmy również popracować nad rozszerzeniem użytkownictwa naszej bazy danych, dodając graficzny interfejs użytkownika służący do przeglądania i uaktualniania jej rekordów. GUI można budować w sposób przenośny za pomocą obsługi modułu tkinter (Tkinter w wersji 2.6) biblioteki standardowej Pythona lub zewnętrznych pakietów narzędzi, takich jak WxPython czy PyQt. Moduł tkinter udostępniany wraz z Pythonem pozwala szybko budować proste graficzne interfejsy użytkownika i jest idealny do uczenia się technik programowania GUI. WxPython i PyQt są nieco bardziej skomplikowane w użyciu, ale często dają w rezultacie graficzny interfejs użytkownika wyższej klasy.

Strony internetowe

Choć graficzne interfejsy użytkownika są wygodne i szybkie, jeśli chodzi o dostępność — nic nie pobije Internetu. Moglibyśmy zatem zaimplementować stronę internetową służącą do przeglądania i uaktualniania rekordów zamiast (lub obok) GUI i sesji interaktywnej. Strony internetowe można tworzyć albo za pomocą prostych narzędzi do skryptów CGI wbudowanych w Pythona, albo rozbudowanych platform zewnętrznych, takich jak Django, TurboGears, Pylons, web2Py, Zope czy Google's App Engine. W Internecie dane nadal mogą być przechowywane w obiektach pickle, plikach shelve czy innym medium opartym na

Pythonie. Skrypty przetwarzające dane są po prostu wykonywane automatycznie na serwerze w odpowiedzi na żądania z przeglądarki internetowej bądź innych klientów i zwracają kod HTML umożliwiający interakcję z użytkownikiem — albo bezpośrednią, albo za pomocą interfejsu dla API platform internetowych.

Usługi sieciowe

Choć klienci internetowe często potrafią analizować składniowo informacje pochodzące z odpowiedzi stron internetowych (technika ta znana jest pod nazwą *screen scraping*), możemy pójść o krok dalej i udostępnić bardziej bezpośredni sposób pobierania rekordów w Internecie za pomocą interfejsu usług sieciowych, takiego jak SOAP czy wywołania XML-RPC — te API obsługiwane są albo przez samego Pythona, albo przez narzędzia zewnętrzne z domeny open source. API tego typu zwracają dane w bardziej bezpośredniej postaci zamiast osadzonych w kodzie HTML strony odpowiedzi.

Bazy danych

Jeśli nasza baza danych ma większy rozmiar lub ma duże znaczenie, możemy przenieść ją z plików `shelve` do bardziej rozbudowanego mechanizmu przechowywania, takiego jak zorientowany obiektywnie system bazy danych ZODB lub bardziej tradycyjny system relacyjnej bazy danych oparty na SQL, taki jak MySQL, Oracle, PostgreSQL czy SQLite. Sam Python udostępniany jest z wbudowanym systemem bazy danych SQLite, jednak inne rozwiązania na licencji open source są również łatwo dostępne. ZODB jest na przykład podobny do plików `shelve` Pythona, jednak likwiduje wiele z jego ograniczeń, obsługując większe bazy danych, uaktualnianie współbieżne, przetwarzanie transakcji, a także automatyczne zapisywanie na zmianach w pamięci. Systemy oparte na SQL, takie jak MySQL, oferują profesjonalne narzędzia służące do przechowywania baz danych i można ich używać bezpośrednio ze skryptu Pythona.

Odwzorowanie obiektywo-relacyjne

Jeśli przeniesiemy przechowywane dane do systemu relacyjnej bazy danych, nie musimy wcale rezygnować z narzędzi do programowania zorientowanego obiektywnie w Pythonie. Narzędzia do odwzorowania obiektywo-relacyjnego (ang. *object-relational mappers*, ORM), takie jak `SQLObject` czy `SQLAlchemy`, potrafią automatycznie odwzorowywać relacyjne tabele i wiersze na klasy oraz obiekty Pythona i odwrotnie, w taki sposób, byśmy mogli przetwarzać przechowane dane za pomocą normalnej składni klas Pythona. Takie rozwiązanie stanowi alternatywę dla zorientowanych obiektywnie systemów bazy danych, takich jak `shelve` i ZODB, i wykorzystuje możliwości zarówno relacyjnych baz danych, jak i modelu klas Pythona.

Choć mam nadzieję, że powyższe wprowadzenie zaostrzy apetyt Czytelników na dalsze zapoznanie się z nimi, wszystkie z przedstawionych powyżej zagadnień wykraczają znacznie poza zakres tego przykładu i ogólnie książki. Osoby, które chcą samodzielnie się z nimi zapoznać, odsyłam do Internetu, dokumentacji biblioteki standardowej Pythona, a także książek poświęconych dziedzinie aplikacji, takich jak *Programming Python*. W tej ostatniej pozycji kontynuuję przedstawiony tutaj przykład, pokazując, jak można dodać do bazy danych zarówno graficzny interfejs użytkownika, jak i stronę internetową, tak by można było przeglądać i uaktualniać rekordy instancji. Mam nadzieję, że tam się spotkamy; najpierw jednak wróćmy do podstaw klas i skończmy omawianie zagadnień dotyczących podstaw języka Python.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy praktyczne aspekty działania wszystkich podstaw klas i programowania zorientowanego obiektowo w Pythonie, budując krok po kroku prosty, choć bardziej realistyczny przykład. Dodaliśmy konstruktory, metody, przeciążanie operatorów, dostosowywanie do własnych potrzeb za pomocą klas podanych, narzędzia introspekcji, a także przy okazji zapoznaliśmy się z innymi koncepcjami, takimi jak kompozycja, delegacja i polimorfizm.

Na koniec uczyniliśmy obiekty utworzone za pomocą klas trwałymi, przechowując je w bazie danych obiektu `shelve` — prostego w użyciu systemu służącego do zapisywania i pobierania własnych obiektów Pythona po kluczu. Omawiając podstawy klas, poznaliśmy także kilka sposobów faktoryzacji kodu w celu zmniejszenia jego powtarzalności i zminimalizowania kosztów utrzymania w przyszłości. Na koniec przyjrzelismy się krótko sposobom rozszerzania kodu za pomocą narzędzi do programowania aplikacji, takich jak graficzne interfejsy użytkownika i bazy danych, omówione w bardziej zaawansowanych publikacjach.

W kolejnych rozdziałach tej części książki powrócimy do omawiania szczegółów leżących u podstaw modelu klas Pythona i zapoznamy się z zastosowaniem w tym języku pewnych koncepcji projektowych wykorzystywanych w celu łączenia klas w większych programach. Zanim jednak przejdziemy dalej, czas wykonać quiz, by sprawdzić wiedzę na temat zagadnień omówionych w niniejszym rozdziale. Ponieważ wykonaliśmy w nim sporo praktycznej pracy, zakończymy rozdział zbiorem pytań teoretycznych, zaprojektowanych z myślą o przeszledzeniu części utworzonego tu kodu i zastanowieniu się nad najważniejszymi koncepcjami leżącymi u jego podstaw.

Sprawdź swoją wiedzę — quiz

- Skąd pochodzi logika formatowania wyświetlanego w sytuacji, gdy pobieramy obiekt klasy `Manager` z pliku `shelve` i wyświetlamy go?
- Kiedy pobieramy obiekt klasy `Person` z pliku `shelve` bez importowania jego modułu, skąd obiekt ten wie, że ma metodę `giveRaise`, którą możemy wywołać?
- Dlaczego tak ważne jest, by przetwarzanie umieszczać w metodach, zamiast zapisywać je na stałe poza klasami?
- Dlaczego lepiej jest dostosowywać kod do własnych potrzeb za pomocą klas podanych, niż kopować oryginał i modyfikować go?
- Dlaczego lepiej jest wywoływać metody klas nadrzędnych w celu wykonania działań domyślnych, zamiast kopować je i modyfikować ich kod w klasach podanych?
- Dlaczego lepiej jest korzystać z narzędzi takich jak `__dict__`, pozwalających na uniwersalne przetwarzanie obiektów, zamiast pisać własny kod dla każdego z typów klas?
- Mówiąc ogólnie, kiedy możemy zdecydować się na korzystanie z osadzania obiektów i kompozycji zamiast dziedziczenia?
- W jaki sposób można zmodyfikować klasy z niniejszego rozdziału, tak by implementowały one osobistą bazę danych kontaktów w Pythonie?

Sprawdź swoją wiedzę — odpowiedzi

1. W ostatecznej wersji naszych klas Manager dziedziczy swoją metodę wyświetlania `__str__` po klasie AttrDisplay z oddzielnego modułu `classestools`. Klasa Manager nie ma własnej metody, dlatego wyszukiwanie dziedziczenia wspina się do jej klasy nadzędnej Person. Ponieważ tam także nie ma metody `__str__`, wyszukiwanie wspina się jeszcze wyżej i odnajduje metodę w klasie AttrDisplay. Nazwy klas podane w nawiasach w wierszu nagłówka instrukcji `class` udostępniają łącza do znajdujących się wyżej klas nadzędnych.
2. Pliki `shelve` (a tak naprawdę wykorzystywany przez nie moduł `pickle`) automatycznie ponownie łączą instancję z klasą, z której została utworzona, kiedy instancja ta jest później z powrotem ładowana do pamięci. Python wewnętrznie ponownie importuje klasę z jej modułu, tworzy instancję z przechowywanymi atrybutami i ustawia łącze `__class__` instancji, tak by wskazywało na oryginalną klasę. W ten sposób załadowane instancje automatycznie pozyskują wszystkie oryginalne metody (takie, jak `lastName`, `giveRaise` czy `__str__`), nawet jeśli nie zaimportowaliśmy klasy instancji do naszego zakresu.
3. Przesuwanie przetwarzania do metod jest ważne, ponieważ dzięki temu w przyszłości będziemy mieli tylko jedną kopię do zmodyfikowania, a metody mogą być wykonywane na dowolnej instancji. Na tym polega w Pythonie *hermetyzacja* — opakowywanie logiki w interfejsy, tak by lepiej obsługiwać utrzymywanie kodu w przyszłości. Jeśli tego nie zrobimy, narazimy się na powtarzalność kodu, co może znacznie zwiększyć wysiłek niezbędnego w przyszłości do jego utrzymania w przypadku jego ewolucji.
4. Dostosowanie kodu do własnych potrzeb za pomocą klas podrzędnych zmniejsza ilość wymaganej pracy programistycznej. W programowaniu zorientowanym obiektywo kod tworzymy za pomocą *dostosowywania* tego, co już zostało wykonane, do własnych potrzeb zamiast kopiowania czy modyfikowania istniejącego. To właśnie najważniejsza koncepcja leżąca u podstaw programowania zorientowanego obiektywo — ponieważ możemy z łatwością rozszerzać poprzednią pracę, tworząc kod nowych klas podrzędnych, możemy wykorzystać to, co zrobiliśmy wcześniej. Jest to o wiele lepsze rozwiązanie od zaczynania za każdym razem od podstaw czy wprowadzania kilku powtarzających się kopii kodu, które w przyszłości będą musieli aktualniać.
5. Kopiowanie i modyfikowanie kodu *podwaja* potencjalny wysiłek wymagany w przyszłości, bez względu na kontekst. Jeśli klasa podrzędna musi wykonać działania domyślne zapisane w kodzie klasy nadzędnej, o wiele lepiej jest wywołać je za pośrednictwem nazwy klasy nadzędnej, niż kopiować ich kod. Tak samo jest również w przypadku konstruktorów klas nadzędnych. I znów, kopowanie kodu powoduje jego powtarzalność, co staje się sporym problemem w miarę jego ewoluowania.
6. Narzędzia uniwersalne pomagają uniknąć zapisywania w kodzie na stałe rozwiązań, które muszą pozostać zsynchronizowane z resztą klasy w miarę jej ewoluowania w czasie. Uniwersalna metoda `__str__` nie musi na przykład być aktualniana za każdym razem, gdy w konstruktorze `__init__` do instancji dodawany jest nowy atrybut. Dodatkowo uniwersalna metoda `print` dziedziczona przez wszystkie klasy pojawia się (i musi być modyfikowana) tylko w jednym miejscu — zmiany wprowadzone w wersji ogólnej są pobierane przez wszystkie klasy dziedziczące po klasie uniwersalnej. I znów, eliminacja powtarzalności kodu ogranicza wysiłek niezbędny w przyszłości. Jest to jedna z podstawowych zalet klas.

7. Dziedziczenie najlepiej nadaje się do tworzenia kodu rozszerzeń opartych na bezpośrednim dostosowywaniu do własnych potrzeb (jak nasza klasa `Manager`, będąca wyspecjalizowaną wersją klasy `Person`). Kompozycja dobrze sprawdza się w sytuacjach, gdy kilka obiektów agregowanych jest w całość i sterowanych przez klasę warstwy kontrolera. Dziedziczenie przekazuje wywołania w górę, w celu ponownego ich wykorzystania, natomiast kompozycja przekazuje je w dół, w celu delegowania. Dziedziczenie i kompozycja nie wykluczają się wzajemnie. Często obiekty osadzone w kontrolerze same są kodem dostosowanym do własnych potrzeb za pomocą dziedziczenia.
8. Klasy z niniejszego rozdziału można wykorzystać jako kod szablonu implementującego różne typy baz danych. Wystarczy zmienić ich cel, modyfikując konstruktory w taki sposób, by zapisywały one inne atrybuty, i udostępniając metody właściwe dla docelowego zastosowania. W przypadku bazy danych kontaktów możemy na przykład wykorzystać atrybuty takie, jak `name`, `address`, `birthday`, `phone` czy `email`, a także metody odpowiednie dla tego celu. Metoda o nazwie `sendmail` może na przykład wykorzystać moduł biblioteki standardowej Pythona `smtplib` do automatycznego wysłania po wywołaniu wiadomości e-mail do jednego z kontaktów (więcej informacji na temat takich narzędzi znajduje się w dokumentacji Pythona lub publikacjach poświęconych aplikacjom). Narzędzie `AttrDict` napisane w tym rozdziale można wykorzystać w jego aktualnej postaci do wyświetlenia obiektów, ponieważ celowo zostało ono zaprojektowane jako uniwersalne. Większość kodu baz danych `shelve` można wykorzystać do przechowywania obiektów, z niewielkimi tylko zmianami.

Szczegóły kodu klas

Osoby, które nie zrozumiały w pełni programowania zorientowanego obiektowo w Pythonie, nie powinny się tym martwić — po krótkim wprowadzeniu teraz zaczniemy się bardziej szczegółowo zagłębiać w koncepcje wprowadzone wcześniej. W tym oraz kolejnym rozdziale raz jeszcze przyjrzymy się mechanizmom związanym z klasami. W niniejszym rozdziale omówimy klasy, metody oraz dziedziczenie, rozszerzając niektóre najważniejsze zagadnienia wprowadzone w rozdziale 26. Ponieważ klasa jest naszym ostatnim narzędziem przestrzeni nazw, streścimy tutaj również koncepcję przestrzeni nazw Pythona.

W kolejnym rozdziale kontynuujemy dogłębne poznawanie mechanizmów klas, omawiając jeden ich szczególny aspekt — przeciążanie operatorów. Poza przedstawieniem szczegółów obu rozdziałów — ten i kolejny — dają nam również okazję do zapoznania się z nieco większymi klasami niż omówione dotychczas.

Instrukcja class

Choć instrukcja `class` w Pythonie może się z zewnątrz wydawać podobna do narzędzi z innych zorientowanych obiektowo języków programowania, przy bliższym jej poznaniu okazuje się inna od tego, do czego przyzwyczajeni są niektórzy programiści. Na przykład tak, jak w C++ instrukcja `class` jest podstawowym narzędziem programowania zorientowanego obiektowo w Pythonie, jednak w przeciwieństwie do języka C++ `class` w Pythonie nie jest deklaracją. Tak jak `def`, instrukcja `class` tworzy obiekt, a także jest niejawnym przypisaniem — kiedy się ją wykonuje, generuje ona obiekt klasy i przechowuje referencję do niego w nazwie wykorzystanej w nagłówku. Podobnie jak `def`, instrukcja `class` jest prawdziwym kodem wykonywalnym — klasa nie istnieje, dopóki Python nie dotrze do definiującej ją instrukcji `class` i nie wykona jej (zazwyczaj kiedy importuje się moduł zawierający tę instrukcję, jednak nie wcześniej).

Ogólna forma

Instrukcja `class` jest instrukcją złożoną, z ciałem wciętych instrukcji, które zazwyczaj pojawia się pod nagłówkiem. W nagłówku po nazwie klasy w nawiasach wymienione są klasy nadzędne, rozdzielone przecinkami. Podanie większej liczby klas nadzędnych prowadzi do dziedziczenia wielokrotnego (omówionego nieco bardziej formalnie w rozdziale 30.). Poniżej znajduje się ogólna postać instrukcji `class`.

```
class <nazwa>(klasa_nadrzędna, ...):          # Przypisanie do nazwy
    data = value                                # Współdzielone dane klasy
    def method(self, ...):                      # Metody
        self.member = value                      # Dane instancji
```

Wewnątrz instrukcji `class` przypisania generują atrybuty klas, natomiast metody o specjalnych nazwach przeciążają operatory. Funkcja o nazwie `__init__` jest na przykład po zdefiniowaniu wywoływana w momencie konstruowania obiektu instancji.

Przykład

Jak widzieliśmy, klasy są w dużej mierze po prostu przestrzeniami nazw — to znaczy narzędziami służącymi do definiowania nazw (czyli atrybutów) eksportujących dane oraz logikę do klientów. W jaki sposób przechodzimy zatem od instrukcji `class` do przestrzeni nazw?

A oto, jak się to odbywa. Tak jak w plikach modułów, instrukcje osadzone wewnątrz ciała instrukcji `class` tworzą jej atrybuty. Kiedy Python wykonuje instrukcję `class` (a nie wywołuje klasy), wykonuje wszystkie instrukcje w jej ciele, od góry do dołu. Przypisania mające miejsce w tym czasie tworzą w zakresie lokalnym klasy zmienne, które stają się atrybutami w powiązanym obiekcie klasy. Z tego powodu klasy przypominają zarówno moduły, jak i funkcje.

- Tak jak funkcje, instrukcje `class` są zakresami lokalnymi, w których istnieją zmienne utworzone przez zagnieździone operacje przypisania.
- Tak jak zmienne w module, zmienne przypisane w instrukcjach `class` stają się atrybutami obiektu klasy.

Podstawową cechą wyróżniającą klas jest to, że ich przestrzenie nazw są w Pythonie również podstawą dziedziczenia. Atrybuty referencji, które nie zostaną odnalezione w obiekcie klasy bądź instancji, pobierane są z innych klas.

Ponieważ `class` jest instrukcją złożoną, wewnątrz jej ciała można umieścić dowolną inną instrukcję — `print`, `=`, `if`, `def` i tak dalej. Wszystkie instrukcje wewnątrz instrukcji `class` wykonywane są wtedy, gdy wykonywana jest `class` (a nie wtedy, gdy klasa jest później wywoływana w celu utworzenia instancji). Przypisanie nazw wewnątrz instrukcji `class` tworzy atrybuty klasy, a zagnieździone instrukcje `def` tworzą metody klasy. Również inne operacje przypisania tworzą atrybuty klasy.

Przykładowo przypisania prostych obiektów niebędących funkcjami do atrybutów klasy tworzą *atrybuty danych*, współdzielone przez wszystkie instancje.

```
>>> class SharedData:
...     spam = 42                                # Wygenerowanie atrybutu danych klasy
...
>>> x = SharedData()                          # Utworzenie dwóch instancji
>>> y = SharedData()
>>> x.spam, y.spam                           # Dziedziczą i współdzielą zmienną spam
(42, 42)
```

W kodzie powyżej, ponieważ zmienna `spam` przypisana jest na najwyższym poziomie instrukcji `class`, jest ona dołączana do klasy, dlatego będzie współdzielona przez wszystkie instancje.

Możemy ją zmodyfikować, przechodząc przez nazwę klasy, a także odnieść się do niej albo przez instancje, albo przez klasę.¹

```
>>> SharedData.spam = 99
>>> x.spam, y.spam, SharedData.spam
(99, 99, 99)
```

Takie atrybuty klas można wykorzystać do zarządzania informacjami rozciągającymi się na wszystkie instancje — na przykład licznikiem liczby wygenerowanych instancji (pomysł ten rozwiniemy w przykładzie z rozdziału 31.). Sprawdźmy teraz, co się stanie, jeśli przypiszemy zmienną spam za pośrednictwem instancji, a nie klas.

```
>>> x.spam = 88
>>> x.spam, y.spam, SharedData.spam
(88, 99, 99)
```

Przypisania do atrybutów instancji tworzą lub modyfikują zmienne w instancji, a nie we wspólnie dziedziczony klasie. Mówiąc bardziej ogólnie, wyszukiwanie dziedziczenia występuje jedynie przy *referencjach* atrybutów, a nie przypisywaniu. Przypisanie atrybutu obiektu zawsze modyfikuje ten obiekt, a nie żaden inny.² Przykładowo wyrażenie `y.spam` powoduje wyszukiwanie atrybutu w klasie za pomocą dziedziczenia, natomiast przypisanie wartości do `x.spam` dołącza zmienną do samego obiektu instancji `x`.

Poniżej znajduje się nieco szerszy przykład tego zachowania, który zachowuje tę samą zmienną w dwóch różnych miejscach. Założymy, że wykonamy poniższą instrukcję `class`.

```
class MixedNames:
    data = 'mielonka'                                # Zdefiniowanie klasy
    def __init__(self, value):                         # Przypisanie atrybutu klasy
        self.data = value                             # Przypisanie nazwy metody
    def display(self):                                # Przypisanie atrybutu instancji
        print(self.data, MixedNames.data)             # Atrybut instancji, atrybut klasy
```

Klasa ta zawiera dwie instrukcje `def` wiążące atrybuty klas z funkcjami metod. Zawiera również instrukcję przypisania `=`. Ponieważ operacja ta przypisuje zmienną `data` wewnętrz instrukcji `class`, znajduje się w lokalnym zakresie klasy i staje się atrybutem obiektu klasy. Jak wszystkie atrybuty klasy, zmienna `data` jest dziedziczona i współdzielona przez wszystkie instancje klasy, które nie mają własnego atrybutu o nazwie `data`.

Kiedy tworzymy instancje tej klasy, atrybut `data` jest do nich dołączany przez przypisanie do `self.data` w metodzie konstruktora.

```
>>> x = MixedNames(1)                            # Utworzenie dwóch obiektów instancji
>>> y = MixedNames(2)                            # Każdy ma własne dane
>>> x.display(); y.display()                   # self.data jest inny, Subclass.data jest tym samym
1 mielonka
2 mielonka
```

¹ Osoby, które używały w przeszłości języka C++, mogą rozpoznać to zachowanie jako podobne do „statycznych” składowych klas z tego języka — składowych przechowywanych w klasie i niezależnych od instancji. W Pythonie nie jest to nic specjalnego — wszystkie atrybuty klas są po prostu nazwami przypisany w instrukcji `class`, bez względu na to, czy odnoszą się akurat do funkcji („metod” w C++), czy czegoś innego („składowych” w C++). W rozdziale 31. spotkamy się również z metodami statycznymi Pythona (podobnymi do tych z języka C++), które są po prostu funkcjami bez `self`, przetwarzającymi zazwyczaj atrybuty klas.

² O ile klasa nie redefiniowała operacji przypisania atrybutów za pomocą metody przeciążania operatorów `__setattr__` w taki sposób, by robiła ona coś unikalnego (zostanie to omówione w rozdziale 29.).

W rezultacie zmienna `data` istnieje w dwóch miejscach — w obiektach instancji (utworzona przez przypisanie `self.data` w metodzie `__init__`), a także w klasie, po której jest dziedziczona przez instancje (utworzona przez przypisanie `data` w klasie). Metoda `display` klasy wyświetla obie wersje, najpierw kwalifikując instancję `self`, a później klasę.

Wykorzystując te techniki do przechowania atrybutów w różnych obiektach, określamy ich zakres widoczności. Kiedy zmienne dołączane są do klas, są współdzielone. W instancjach zmienne zapisują dane dla tych instancji, a nie zachowanie współdzielone przez wszystkie instancje klasy. Choć wyszukiwanie dziedziczenia może dla nas wyszukiwać zmienne, zawsze możemy uzyskać dostęp do atrybutu z dowolnego miejsca drzewa poprzez bezpośrednie dotarcie do pożądanego obiektu.

W poprzednim przykładzie wyrażenia `x.data` lub `self.data` zwracają zmienną instancji, która normalnie ukrywa tę samą zmienną z klasy. Wyrażenie `MixedNames.data` pobiera natomiast zmienną klasy w sposób bezpośredni. Później spotkamy się z różnymi rolami tych wzorców kodu; w kolejnym podrozdziale opiszymy jeden z najczęściej spotykanych.

Metody

Ponieważ znamy już funkcje, powinniśmy również znać metody klas. Metody to po prostu obiekty funkcji utworzone za pomocą instrukcji `def` zagnieżdżonych w ciele instrukcji `class`. Z abstrakcyjnego punktu widzenia metody udostępniają zachowanie, które dziedziczą instancje klasy. Z punktu widzenia programowania metody działają dokładnie tak samo jak proste funkcje, z jednym kluczowym wyjątkiem — pierwszy argument metody zawsze otrzymuje obiekt instancji, który jest domniemanym podmiotem wywołania metody.

Innymi słowy, Python automatycznie odwzorowuje wywołania metod instancji na funkcje metod klas w następujący sposób. Wywołania metod wykonywane za pośrednictwem instancji, takie jak poniższe:

```
instancja.metoda(argumenty...)
```

są automatycznie przekładane na wywołania funkcji metod klas w poniższej postaci:

```
klasa.metoda(instancja, argumenty...)
```

gdzie klasa ustalana jest na podstawie odnalezienia nazwy metody za pomocą procedury wyszukiwania dziedziczenia Pythona. Tak naprawdę w Pythonie poprawne są obie formy wywołania.

Poza normalnym sposobem dziedziczenia nazw atrybutów metod specjalny pierwszy argument jest jedną cechą wyróżniającą wywołań metod. W metodzie klasy pierwszy argument jest najczęściej, zgodnie z konwencją, nazywany `self` (z technicznego punktu widzenia znaczenie ma jedynie jego pozycja, nie nazwa). Argument ten udostępnia metodom punkt zaczepienia z powrotem do instancji będącej podmiotem wywołania. Ponieważ klasy generują wiele obiektów instancji, muszą stosować ten argument w celu zarządzania danymi różniącymi się między poszczególnymi instancjami.

Programiści języka C++ mogą rozpoznać argument Pythona `self` jako podobny do wskaźnika `this` z C++. W Pythonie `self` jest jednak zawsze obecne w kodzie w jawnym sposób — metody zawsze muszą przejść przez `self` w celu pobrania lub zmodyfikowania atrybutów instancji

przetwarzanej w bieżącym wywołaniu metody. Taka jawną naturą `self` jest celowa — obecność tej nazwy sprawia, że oczywiste staje się, iż w skrypcie używamy nazw atrybutów instancji, a nie nazw z zakresu lokalnego czy globalnego.

Przykład metody

By wyjaśnić te koncepcje, przedstawmy je na przykładzie. Założymy, że definiujemy poniższą klasę.

```
class NextClass:  
    def printer(self, text):  
        self.message = text  
        print(self.message)  
        # Zdefiniowanie klasy  
        # Zdefiniowanie metody  
        # Modyfikacja instancji  
        # Dostęp do instancji
```

Nazwa `printer` odnosi się do obiektu funkcji. Ponieważ jest przypisana w zakresie instrukcji `class`, staje się atrybutem obiektu klasy i jest dziedziczona przez wszystkie instancje utworzone z tej klasy. Normalnie, ponieważ metody takie, jak `printer` zaprojektowane są do przetwarzania instancji, wywołujemy je przez instancje.

```
>>> x = NextClass()                      # Utworzenie instancji  
  
>>> x.printer('wywołanie instancji')      # Wywołanie jej metody  
wywołanie instancji  
  
>>> x.message                           # Modyfikacja instancji  
'wywołanie instancji'
```

Po wywołaniu metody za pomocą kwalifikacji instancji, jak w powyższym kodzie, metoda `printer` jest najpierw wyszukiwana za pomocą dziedziczenia, a później do jej argumentu `self` automatycznie zostaje przypisany obiekt instancji (`x`). Argument `text` otrzymuje łańcuch znaków przekazany w wywołaniu ('wywołanie instancji'). Ponieważ Python sam automatycznie przekazuje pierwszy argument do `self`, tak naprawdę musimy przekazać tylko jeden argument. Wewnątrz metody `printer` nazwa `self` wykorzystywana jest do dostępu lub ustawienia danych instancji, ponieważ odnosi się z powrotem do aktualnie przetwarzanej instancji.

Metody można wywoływać na dwa sposoby — za pośrednictwem instancji oraz za pośrednictwem samej klasy. Możemy na przykład również wywołać metodę `printer`, przechodząc nazwę klasy, pod warunkiem że w jawnym sposobie przekażemy instancję do argumentu `self`.

```
>>> NextClass.printer(x, 'wywołanie klasy') # Bezpośrednie wywołanie klasy  
wywołanie klasy  
  
>>> x.message                           # Ponowna modyfikacja instancji  
'wywołanie klasy'
```

Wywołania przekazane przez instancję i klasę mają ten sam efekt, o ile w formie z klasą przekażemy ten sam obiekt instancji. Domyslnie otrzymamy komunikat o błędzie, jeśli spróbujemy wywołać metodę bez podania instancji.

```
>>> NextClass.printer('złe wywołanie')  
TypeError: unbound method printer() must be called with NextClass instance...
```

Wywoływanie konstruktorów klas nadzędnych

Metody są normalnie wywoływanie za pośrednictwem instancji. Wywołania metod za pośrednictwem klas pojawiają się jednak w różnych specjalnych rolach. Jeden z często spotykanych scenariuszy obejmuje metodę konstruktora. Metoda `__init__`, podobnie jak wszystkie atrybuty, wyszukiwana jest za pomocą dziedziczenia. Oznacza to, że w momencie konstrukcji Python lokalizuje i wywołuje tylko jedną metodę `__init__`. Jeśli konstruktory klas podrzędnych potrzebują gwarancji wykonania również logiki czasu konstrukcji z klasy nadzędnej, muszą wywołać metodę `__init__` klasy nadzędnej w sposób jawnym, za pośrednictwem tej klasy.

```
class Super:  
    def __init__(self, x):  
        ...kod domyślny...  
  
class Sub(Super):  
    def __init__(self, x, y):  
        Super.__init__(self, x)           # Wykonanie metody __init__ klasy nadzędnej  
        ...własny kod...                 # Wykonanie własnych działań inicjalizacyjnych  
  
I = Sub(1, 2)
```

Jest to jeden z niewielu kontekstów, w których kod będzie prawdopodobnie wywoływał metodę przeciążania operatorów w sposób bezpośredni. Oczywiście powinniśmy wywoływać konstruktor klasy nadzędnej w ten sposób jedynie wtedy, gdy naprawdę musimy ją wykonać — bez tego wywołania zostanie on całkowicie zastąpiony metodą klasy podrzędnej. Bardziej realistyczne zastosowanie tej techniki w praktyce znajduje się w przykładzie klasy Manager z poprzedniego rozdziału.³

Inne możliwości wywoływania metod

Taki wzorzec wywoływania metod za pośrednictwem klas jest podstawą rozszerzania (a nie całkowitego zastępowania) odziedziczonego zachowania metody. W rozdziale 31. spotkamy się również z nową opcją, dodaną w Pythonie 2.2 — metodami *statycznymi*, które pozwalają zapisywać w kodzie metody nieoczekujące podania obiektów instancji w pierwszym argumentie. Takie metody mogą działać jak proste funkcje bez instancji, ich nazwy są lokalne dla klas, w których są zapisane, i mogą one być wykorzystywane do zarządzania danymi klasy. Powiązana z tym koncepcja *metody klasy* otrzymuje po wywoaniu klasę zamiast instancji i można ją wykorzystać do zarządzania danymi dla klasy. Są to jednak rozszerzenia opcjonalne i bardziej zaawansowane. Normalnie zawsze musimy przekazać instancję do metody, bez względu na to, czy jest ona wywołana za pośrednictwem instancji, czy też klasy.

Dziedziczenie

Celem istnienia narzędzia przestrzeni nazw, jakim jest instrukcja `class`, jest obsługa dziedziczenia zmiennych. W niniejszym podrozdziale rozszerzymy nieco informacje na temat niektórych mechanizmów oraz ról dziedziczenia atrybutów w Pythonie.

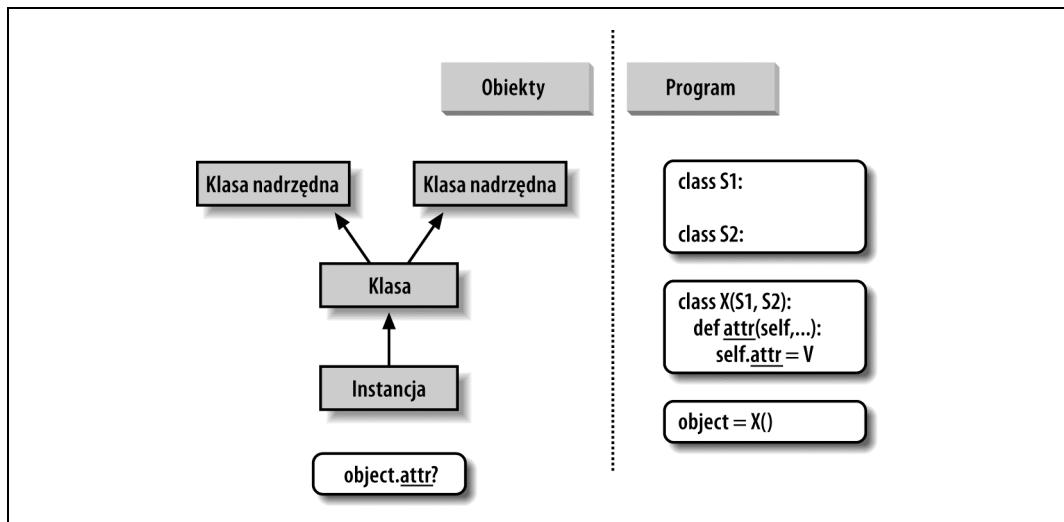
³ Można również zapisać kod kilku metod `__init__` w jednej klasie, jednak wykorzystana zostanie jedynie ostatnia definicja. Więcej informacji na ten temat znajduje się w rozdziale 30.

W Pythonie dziedziczenie ma miejsce przy kwalifikacji obiektu i obejmuje przeszukiwanie drzewa definicji atrybutów (jednej lub większej liczby przestrzeni nazw). Za każdym razem gdy użyjemy wyrażenia w formie `obiekt.atrybut` (gdzie `obiekt` jest obiektem instancji lub klasy), Python przeszukuje przestrzeń nazw od dołu do góry, rozpoczynając od obiektu i szukając pierwszego atrybutu o danej nazwie, jaki potrafi znaleźć. Obejmuje to również referencje do atrybutów `self` w metodach. Ponieważ definicje niżej w drzewie przesłaniają te umieszczone wyżej, dziedziczenie jest podstawą specjalizacji.

Tworzenie drzewa atrybutów

Na rysunku 28.1 przedstawiono sposób tworzenia drzew przestrzeni nazw oraz umieszczania w nich zmiennych. Mówiąc ogólnie:

- Atrybuty instancji generowane są przez przypisania do atrybutów `self` w metodach.
- Atrybuty klas tworzone są przez instrukcje (przypisania) w instrukcjach `class`.
- Łącza do klas nadrzędnych tworzy się, wymieniając te klasy w nawiasach w nagłówku instrukcji `class`.



Rysunek 28.1. Kod programu tworzy drzewo obiektów w pamięci, by móc je przeszukiwać pod kątem dziedziczenia atrybutów. Wywołanie klasy tworzy nową instancję pamiętającą swoją klasę. Wykonanie instrukcji `class` tworzy nową klasę, natomiast jej klasy nadrzędne wymienione są w nawiasach w nagłówku instrukcji `class`. Każda referencja do atrybutu wywołuje nowe wyszukiwanie od dołu drzewa do góry — nawet w przypadku atrybutów `self` wewnątrz metod klas.

W rezultacie otrzymujemy drzewo przestrzeni nazw atrybutów prowadzące od instancji, przez klasę, która ją wygenerowała, do wszystkich klas nadrzędnych z nagłówka instrukcji `class`.

Python szuka od dołu drzewa do góry, od instancji po klasy nadrzędne, za każdym razem, gdy użyjemy kwalifikacji w celu pobrania nazwy atrybutu z obiektu instancji.⁴

Specjalizacja odziedziczonych metod

Opisany wyżej model dziedziczenia oparty na przeszukiwaniu drzew okazuje się doskonałym sposobem specjalizowania systemów. Ponieważ dziedziczenie odnajduje zmienne w klasach podrzędnych przed sprawdzeniem klas nadrzędnych, podklasy mogą zastępować zachowanie domyślne, redefiniując atrybuty swoich klas nadrzędnych. Tak naprawdę możemy budować całe systemy jako hierarchie klas rozszerzane przez dodawanie nowych zewnętrznych klas podrzędnych, zamiast wprowadzać modyfikacje istniejącej logiki w miejscu.

Pomysł redefiniowania odziedziczonych zmiennych prowadzi do różnych technik specjalizacyjnych. Klasa podrzędna może na przykład całkowicie zastępować dziedziczone atrybuty, udostępniać atrybuty, które klasa nadrzędna oczekuje znaleźć, a także rozszerzać metody klasy nadrzędnej poprzez wywołania tej klasy z przesłoniętej metody. Widzieliśmy już, jak w praktyce działa zastępowanie. Poniżej znajduje się przykład rozszerzania.

```
>>> class Super:
...     def method(self):
...         print('w Super.method')
...
>>> class Sub(Super):
...     def method(self):          # Przesłonienie metody
...         print('początek Sub.method')    # Dodanie działań
...         Super.method(self)        # Wykonanie działania domyślnego
...         print('koniec Sub.method')
...
...
```

Kluczem są tu bezpośrednie wywołania metody klasy nadrzędnej. Klasa Sub zastępuje metodę `method` z klasy `Super` za pomocą własnej, wyspecjalizowanej wersji. Wewnątrz tego zastąpienia klasa `Sub` wywołuje z powrotem wersję eksportowaną przez klasę nadrzędną `Super` w celu wykonania zachowania domyślnego. Innymi słowy, `Sub.method` rozszerza jedynie zachowanie metody `Super.method`, zamiast całkowicie je zastępować.

```
>>> x = Super()                      # Utworzenie instancji klasy Super
>>> x.method()                        # Wykonanie metody Super.method
w Super.method
...
>>> x = Sub()                         # Utworzenie instancji klasy Sub
>>> x.method()                        # Wykonuje Sub.method, co wywołuje Super.method
początek Sub.method
w Super.method
koniec Sub.method
```

Taki wzorzec tworzenia rozszerzeń jest często wykorzystywany w przypadku konstruktorów. Przykład takiego działania znajduje się we wcześniejszym podrozdziale „Metody”.

⁴ Opis ten nie jest w stu procentach pełny, ponieważ możemy również tworzyć atrybuty instancji i klas, przypisując je do obiektów poza instrukcjami `class`. Jest to jednak rozwiązanie rzadziej spotykane i czasami bardziej podatne na błędy (zmiany nie są ograniczane do instrukcji `class`). W Pythonie wszystkie atrybuty są zawsze domyślnie dostępne. O prywatności zmiennych powiemy więcej w rozdziale 29., przy okazji omawiania metody `__setattr__`, w rozdziale 30., gdy spotkamy się ze zmiennymi `_X`, a także ponownie w rozdziale 38., gdzie zaimplementujemy ją za pomocą dekoratora klas.

Techniki interfejsów klas

Rozszerzanie jest jedną z metod wykonywania interfejsów klas nadrzędnych. W poniższym pliku o nazwie `specialize.py` zdefiniowano większą liczbę klas w celu zilustrowania różnych popularnych technik.

Super

Definiuje funkcje `method` oraz `delegate`, które oczekują zmiennej `action` w klasach podrzędnych.

Inheritor

Nie udostępnia żadnych nowych zmiennych, dlatego wszystko ma zdefiniowane w klasie Super.

Replacer

Przesłania metodę `method` klasy Super za pomocą własnej wersji.

Extender

Dostosowuje metodę `method` klasy Super do własnych potrzeb, przesłaniając ją i wywołując w celu wykonania działania domyślnego.

Provider

Implementuje metodę `action` oczekiwana przez metodę `delegate` klasy Super.

Należy zapoznać się z każdą z tych klas, by zrozumieć, jak działają różne sposoby dostosowywania wspólnej klasy nadrzędnej do własnych potrzeb. Poniżej widoczna jest zawartość pliku.

```
class Super:
    def method(self):
        print('w Super.method')                      # Zachowanie domyślne
    def delegate(self):
        self.action()                                # Oczekuje zdefiniowania

class Inheritor(Super):                           # Odziedziczenie wszystkich metod
    pass

class Replacer(Super):                          # Całkowite zastąpienie metody method
    def method(self):
        print('w Replacer.method')

class Extender(Super):                         # Rozszerzenie działania metody method
    def method(self):
        print('początek Extender.method')
        Super.method(self)
        print('koniec Extender.method')

class Provider(Super):                         # Uzupełnienie wymaganej metody
    def action(self):
        print('w Provider.action')

if __name__ == '__main__':
    for klass in (Inheritor, Replacer, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

Warto zwrócić tutaj uwagę na dwie kwestie. Po pierwsze, kod samosprawdzający na dole przykładu tworzy instancje trzech różnych klas w pętli `for`. Ponieważ klasy są obiektami, możemy

je umieścić w krotce i tworzyć instancje w sposób uniwersalny (więcej na temat tej koncepcji później). Klasy mają również specjalny atrybut `_name_`, podobnie jak moduły. Jest on od razu ustawiany na łańcuch znaków zawierający nazwę z nagłówka instrukcji `class`. Oto co się stanie, kiedy wykonamy ten plik.

```
% python specialize.py
```

```
Inheritor...
w Super.method
```

```
Replacer...
w Replacer.method
```

```
Extender...
początek Extender.method
in Super.method
koniec Extender.method
```

```
Provider...
w Provider.action
```

Abstrakcyjne klasy nadrzędne

Warto zwrócić uwagę na działanie klasy `Provider` w poprzednim przykładzie. Kiedy wywołamy metodę `delegate` za pośrednictwem instancji `Provider`, następują *dwa* wyszukiwania dziedziczenia.

1. W wywołaniu `x.delegate` Python odnajduje metodę `delegate` w klasie nadrzędnej `Super`, szukając w instancji `Provider` i wyżej. Instancia `x` jest jak zwykle przekazywana do argumentu `self` metody.
2. Wewnątrz metody `Super.delegate` wywołanie `self.action` powoduje nowe, niezależne wyszukiwanie dziedziczenia w `self` i wyżej. Ponieważ `self` odnosi się do instancji `Provider`, metoda `action` zostaje odnaleziona w klasie podrzędnej `Provider`.

Taka struktura kodu wypełniająca luki jest typowa dla programowania zorientowanego obiektowo. Przynajmniej w kontekście metody `delegate` klasa nadrzędna z tego przykładu jest czymś, co czasami nazywa się *abstrakcyjną klasą nadrzędną* — klasą, która oczekuje, że część jej zachowania zostanie udostępniona przez klasę podrzędną. Jeśli oczekiwana metoda nie jest zdefiniowana w klasie podrzędnej, Python zgłasza wyjątek niezdefiniowanej zmiennej, kiedy wyszukiwanie dziedziczenia zawiedzie.

Twórcy klas czasami czynią takie wymagania względem klas podrzędnych bardziej oczywistymi za pomocą instrukcji `assert` lub zgłaszając wbudowany wyjątek `NotImplementedError` za pomocą instrukcji `raise` (instrukcje mogące zgłaszać wyjątki omówmy dogłębnie w kolejnej części książki). Słownemstęp, poniżej znajduje się przykład działania rozwiązania z instrukcją `assert`.

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'działanie musi zostać zdefiniowane!'      # Jeśli wywołana jest ta wersja

>>> X = Super()
>>> X.delegate()
AssertionError: działanie musi zostać zdefiniowane!
```

Z instrukcją assert spotkamy się w rozdziałach 32. oraz 33. W skrócie, jeśli pierwsze związane z nią wyrażenie jest fałszem, zgłasza ona wyjątek z podanym komunikatem o błędzie. Tutaj wyrażenie zawsze będzie fałszem, by wywołać komunikat o błędzie, jeśli metoda nie zostanie redefiniowana, a dziedziczenie zlokalizuje tę wersję. Alternatywnie niektóre klasy mogą w takich krótkich metodach po prostu bezpośrednio zgłosić wyjątek `NotImplemented` w celu zasygnalizowania błędu.

```
class Super:  
    def delegate(self):  
        self.action()  
    def action(self):  
        raise NotImplementedError('działanie musi zostać zdefiniowane!')  
  
>>> X = Super()  
>>> X.delegate()  
NotImplementedError: działanie musi zostać zdefiniowane!
```

W przypadku instancji klas podrzędnych nadal otrzymamy wyjątek, o ile klasa nadrzędna nie udostępnii oczekiwanej metody zastępującej metodę domyślną z klasy nadrzędnej.

```
>>> class Sub(Super): pass  
...  
>>> X = Sub()  
>>> X.delegate()  
NotImplementedError: działanie musi zostać zdefiniowane!  
  
>>> class Sub(Super):  
...     def action(self): print('mielonka')  
...  
>>> X = Sub()  
>>> X.delegate()  
mielonka
```

By znaleźć nieco bardziej realistyczny przykład praktycznego zastosowania zagadnień omówionych w tym podrozdziale, można zanjrzeć do hierarchii zwierząt z zoo w ćwiczeniu 8 na końcu rozdziału 31. oraz do jego rozwiązania w podrozdziale „Część VI Klasy i programowanie zorientowane obiektywne” w dodatku B. Taksonomie takie jak powyższa są tradycyjnym sposobem wprowadzania podstaw programowania zorientowanego obiektywne, jednak są raczej odległe od realnych zadań wykonywanych przez większość programistów.

Abstrakcyjne klasy nadrzędne z Pythona 2.6 oraz 3.0

W przypadku Pythona 2.6 oraz 3.0 opisane powyżej abstrakcyjne klasy nadrzędne (inaczej „abstrakcyjne klasy bazowe”), wymagające, by metody były uzupełniane w klasach podrzędnych, można implementować również za pomocą specjalnej składni klas. Sposób tworzenia ich kodu uzależniony jest w pewnym stopniu od wykorzystywanej wersji Pythona. W Pythonie 3.0 wykorzystujemy argument ze słowem kluczowym w nagłówku instrukcji `class` w połączeniu ze specjalną składnią dekoratorów — obie techniki omówimy bardziej szczegółowo w kolejnej części książki.

```
from abc import ABCMeta, abstractmethod  
  
class Super(metaclass=ABCMeta):  
    @abstractmethod  
    def method(self, ...):  
        pass
```

W Pythonie 2.6 używamy zamiast tego atrybutu klasy.

```
class Super:  
    __metaclass__ = ABCMeta  
    @abstractmethod  
    def method(self, ...):  
        pass
```

Bez względu na sposób rezultat będzie ten sam — nie możemy utworzyć instancji, o ile metoda nie jest zdefiniowana niżej w drzewie klas. Poniżej znajduje się odpowiednik przykładu z poprzedniego podrozdziału, utworzony w Pythonie 3.0 z wykorzystaniem składni specjalnej.

```
>>> from abc import ABCMeta, abstractmethod  
>>>  
>>> class Super(metaclass=ABCMeta):  
...     def delegate(self):  
...         self.action()  
...     @abstractmethod  
...     def action(self):  
...         pass  
...  
>>> X = Super()  
TypeError: Can't instantiate abstract class Super with abstract methods action  
  
>>> class Sub(Super): pass  
...  
>>> X = Sub()  
TypeError: Can't instantiate abstract class Sub with abstract methods action  
  
>>> class Sub(Super):  
...     def action(self): print('mielonka')  
...  
>>> X = Sub()  
>>> X.delegate()  
mielonka
```

W przypadku zastosowania takiego kodu nie można utworzyć instancji, wywołując klasę z metodą abstrakcyjną, o ile wszystkie z metod abstrakcyjnych nie zostały zdefiniowane w klasach podanych. Choć rozwiązanie to wymaga większej ilości kodu, jego zaletą jest to, że błędy wynikające z brakujących metod zwarcane są, gdy próbujemy utworzyć instancję klasy, a nie później, gdy staramy się wywołać brakującą metodę. Opcję tę możemy także wykorzystać do zdefiniowania oczekiwanej interfejsu, automatycznie weryfikowanego w klasach klienta.

Niestety, rozwiązanie to opiera się na dwóch zaawansowanych narzędziach języka, z którymi jeszcze się nie spotkaliśmy — *dekoratorach funkcji*, wprowadzonych w rozdziale 31. i omówionych dogłębnie w rozdziale 38., a także *deklaracjach metaklas*, o których wspominamy w rozdziale 31., a które przedstawimy w rozdziale 39. Pozostałe aspekty tej opcji odłożymy zatem na później. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardej Pythona; warto także zapoznać się z gotowymi abstrakcyjnymi klasami nadzorowanymi udostępnianymi przez ten język.

Przestrzenie nazw — cała historia

Skoro już omówiliśmy obiekty klas oraz instancji, opowieść o przestrzeniach nazw w Pythonie jest pełna. Dla przypomnienia, szybko streszczę tutaj wszystkie reguły wykorzystywane w odnajdywaniu zmiennych. Pierwszą rzeczą, o jakiej należy pamiętać, jest to, że nazwy ze

składnią kwalifikującą oraz bez niej są traktowane w inny sposób, a niektóre zakresy służą do inicjalizacji przestrzeni nazw obiektów.

- Nazwy bez zapisu kwalifikującego (na przykład `X`) związane są z zakresami.
- Nazwy z zapisem kwalifikującym (na przykład `obiekt.X`) wykorzystują przestrzenie nazw obiektów.
- Niektóre zakresy inicjalizują przestrzenie nazw obiektów (dla modułów oraz klas).

Pojedyncze nazwy — globalne, o ile nie przypisane

Proste nazwy bez składni kwalifikującej zachowują się zgodnie z regułami zakresów leksykalnych LEGB, przedstawionymi w omówieniu funkcji z rozdziału 17.

Przypisanie (`X = wartość`)

Sprawia, że zmienna staje się lokalna. Tworzy lub modyfikuje zmienną `X` w bieżącym zakresie lokalnym, o ile nie zostanie ona zadeklarowana jako globalna.

Referencja (`X`)

Wyszukuje zmienną `X` w bieżącym zakresie lokalnym, następnie we wszystkich funkcjach zawierających, później w zakresie globalnym, a na końcu — wbudowanym.

Nazwy atrybutów — przestrzenie nazw obiektów

Nazwy atrybutów ze składnią kwalifikującą odnoszą się do atrybutów określonych obiektów i stosują się do nich reguły dotyczące modułów oraz klas. W przypadku obiektów klas i instancji reguły związane z referencjami rozszerzane są w taki sposób, by obejmować procedurę wyszukiwania dziedziczenia.

Przypisanie (`obiekt.X = wartość`)

Tworzy lub modyfikuje atrybut o nazwie `X` w przestrzeni nazw obiektu kwalifikującego i żaden inny. Przechodzenie w górę drzewa dziedziczenia ma miejsce jedynie przy referencjach do atrybutu, a nie przypisaniach.

Referencja (`obiekt.X`)

W przypadku obiektów opartych na klasach szuka atrybutu o nazwie `X` w obiekcie, a następnie we wszystkich dostępnych klasach znajdujących się ponad nim, wykorzystując do tego procedurę wyszukiwania dziedziczenia. W przypadku obiektów niebędących klasami pobiera zmienną `X` bezpośrednio z obiektu.

Zen przestrzeni nazw Pythona — przypisania klasyfikują zmienne

Mając osobne procedury wyszukiwania dla zmiennych ze składnią kwalifikującą oraz bez niej, a także wiele warstw do przeszukania w każdej z tych procedur, czasami trudno jest powiedzieć, gdzie trafi zmienna. W Pythonie miejsce *przypisania* zmiennej jest kluczowe — w pełni określa ono zakres lub obiekt, w którym zostanie umieszczona zmienna. Plik `manynames.py` ilustruje, jak zasada ta przekłada się na kod, a także streszcza zagadnienia dotyczące przestrzeni nazw, z którymi spotkaliśmy się w książce.

```

# Plik manynames.py

X = 11                                # Globalna zmienna/atribut z modułu (X lub manynames.X)

def f():
    print(X)                            # Dostęp do zmiennej globalnej X (11)

def g():
    X = 22                                # Zmienna lokalna (funkcji) — X ukrywa X z modułu
    print(X)

class C:
    X = 33                                # Atribut klasy (C.X)
    def m(self):
        X = 44                                # Zmienna lokalna metody (X)
        self.X = 55                            # Atribut instancji (instance.X)

```

Plik przypisuje tę samą zmienną X pięć razy. Ponieważ zmienna ta przypisywana jest jednak w pięciu różnych lokalizacjach, wszystkie X z programu są zupełnie innymi zmiennymi. Od góry do dołu przypisanie do X generuje atrybut lokalny (11), zmienną lokalną funkcji (22), atrybut klasy (33), zmienną lokalną metody (44) oraz atrybut instancji (55). Choć wszystkie pięć zmiennych nosi nazwę X, fakt przypisania ich w różnych miejscach w kodzie źródłowym lub do różnych obiektów sprawia, że są one unikalne.

Powinniśmy poświęcić trochę czasu na uważne przestudiowanie tego przykładu, ponieważ gromadzi on w jednym miejscu informacje omawiane w ostatnich kilku częściach książki. Kiedy zacznie on mieć dla nas sens, osiągniemy swego rodzaju nirwanę przestrzeni nazw Pythona. Oczywiście drogą alternatywną jest po prostu wykonanie tego programu i zobaczenie, co się stanie. Poniżej znajduje się reszta jego kodu źródłowego, tworzącainstancję i wyświetlającą wszystkie X, jakie potrafi znaleźć.

```

# Ciąg dalszy pliku manynames.py

if __name__ == '__main__':
    print(X)                            # 11: moduł (czyli manynames.X poza plikiem)
    f()                                 # 11: zmienna globalna
    g()                                 # 22: zmienna lokalna
    print(X)                            # 11: zmienna modułu bez zmian

    obj = C()
    print(obj.X)                        # Utworzenie instancji
                                         # 33: zmienna klasy odziedziczona przez instancję

    obj.m()
    print(obj.X)                        # Dołączenie nazwy atrybutu X do instancji
                                         # 55: instancja
    print(C.X)                          # 33: klasa (czyli obj.X jeśli nie ma X w instancji)

    #print(C.m.X)                      # PORAŻKA: widoczna tylko w metodzie
    #print(g.X)                        # PORAŻKA: widoczna tylko w funkcji

```

Dane wyjściowe wyświetlane, kiedy plik jest wykonywany, zostały podane w komentarzach w kodzie. Warto je przejrzeć w celu sprawdzenia, do której zmiennej X uzyskujemy dostęp za każdym razem. Należy w szczególności sprawdzić, że możemy przejść klasę w celu uzyskania dostępu do jej atrybutu (C.X), jednak nigdy nie możemy pobrać zmiennych lokalnych w funkcjach lub metodach spoza ich instrukcji def. Zmienne lokalne są widoczne jedynie dla kodu wewnętrznej tej samej instrukcji def i tak naprawdę istnieją w pamięci jedynie na czas wykonywania wywołania funkcji bądź metody.

Niektóre ze zmiennych zdefiniowanych przez ten plik są widoczne *poza nim* dla innych modułów, jednak należy pamiętać, że zanim uzyskamy dostęp do zmiennych z innego pliku, musimy ten plik najpierw zimportować — do tego w końcu służą moduły.

```
# Plik otherfile.py

import manynames

X = 66
print(X)                                     # 66: zmienna globalna tutaj
print(manynames.X)                           # 11: po zimportowaniu zmienne globalne stają się atrybutami

manynames.f()
manynames.g()                                # 11: X z manynames, nie zmienna globalna!
                                                # 22: zmienna lokalna z funkcji innego pliku

print(manynames.C.X)                         # 33: atrybut klasy z innego modułu
I = manynames.C()
print(I.X)                                    # 33: nadal z klasy
I.m()
print(I.X)                                    # 55: teraz z instancji!
```

Warto zwrócić uwagę na to, w jaki sposób `manynames.f()` wyświetla zmienną `X` z modułu `manynames`, a nie `X` przypisaną w tym pliku. Zakresy są zawsze określane zgodnie z pozycją przypisania w pliku źródłowym (leksykalnie) i nigdy nie ma na nie wpływu to, co importuje co albo kto importuje kogo. Warto również odnotować, że własna zmienna `X` instancji nie jest tworzona, dopóki nie wywołamy `I.m()` — atrybuty, tak samo jak wszystkie zmienne, zaczynają istnieć przy przypisaniu, nie wcześniej. Normalnie tworzymy atrybuty instancji, przypisując je w metodzie konstruktora `__init__` klasy, jednak nie jest to jedyna możliwość.

Wreszcie, jak wiemy z rozdziału 17., funkcja może modyfikować zmienne spoza siebie samej dzięki instrukcjom `global` i (w Pythonie 3.0) `nonlocal`. Instrukcje te umożliwiają dostęp do zapisu, jednak modyfikują również reguły dowiązań przestrzeni nazw przypisania.

```
X = 11                                         # Zmienna globalna w module

def g1():
    print(X)                                    # Referencja do zmiennej globalnej modułu

def g2():
    global X
    X = 22                                      # Modyfikacja zmiennej globalnej modułu

def h1():
    X = 33                                      # Zmienna lokalna w funkcji
    def nested():
        print(X)                                # Referencja do zmiennej lokalnej w zakresie zawierającym

def h2():
    X = 33                                      # Zmienna lokalna w funkcji
    def nested():
        nonlocal X                            # Instrukcja Pythona 3.0
        X = 44                                # Modyfikacja zmiennej lokalnej w zakresie zawierającym
```

Oczywiście nie powinniśmy na ogół używać tych samych nazw dla każdej zmiennej w naszym skrypcie. Ten przykład pokazuje jednak, że nawet jeśli tak zrobimy, przestrzenie nazw Pythona zadziałają w taki sposób, by zapobiec konfliktom między zmiennymi przypisanymi w różnych kontekstach.

Słowniki przestrzeni nazw

W rozdziale 22. wspomnieliśmy, że przestrzenie nazw są tak naprawdę zaimplementowane jako słowniki i udostępniane za pomocą wbudowanego atrybutu `__dict__`. Tak samo jest w przypadku obiektów klas oraz instancji — kwalifikacja atrybutów jest wewnętrznie operacją indeksowania słownika, a dziedziczenie atrybutów polega na przeszukaniu połączonych słowników. Tak naprawdę obiekty klas oraz instancji są w Pythonie w dużej mierze po prostu słownikami z łączami. Python udostępnia te słowniki wraz z łączami pomiędzy nimi, tak by można je było wykorzystywać w pewnych zaawansowanych zastosowaniach (na przykład do tworzenia narzędzi).

By zrozumieć, jak wewnętrznie działają atrybuty, przyjrzyjmy się sesji interaktywnej śledzącej rosnące słowniki przestrzeni nazw, kiedy w grę wchodzą klasy. Z prostszą wersją poniższego kodu spotkaliśmy się w rozdziale 26., skoro jednak wiemy już więcej na temat metod i klas nadzędnych, trochę ją tutaj upiększymy. Najpierw zdefiniujmy klasę nadzelną oraz podzijną z metodami, które będą przechowywać dane w instancjach.

```
>>> class super:
...     def hello(self):
...         self.data1 = 'mielonka'
...
>>> class sub(super):
...     def hola(self):
...         self.data2 = 'jajka'
```

Kiedy tworzymy instancję klasy podrzędnej, instancja ta ma na początku pusty słownik przestrzeni nazw, jednak zawiera łącza z powrotem do klasy, dzięki którym działa wyszukiwanie dziedziczenia. Tak naprawdę drzewo dziedziczenia jest w jawnym sposobie dostępnego w atrybutach specjalnych, które można sprawdzać. Instancje mają atrybut `__class__` będący łączem do ich klasy, natomiast klasy mają atrybut `__bases__` będący krotką zawierającą łącza do znajdujących się wyżej w drzewie klas nadzędnych. Poniższy kod wykonuję w Pythonie 3.0; formaty nazw i pewne atrybuty wewnętrzne w wersji 2.6 będą się nieco różnić.

```
>>> X = sub()
>>> X.__dict__                                     # Słownik przestrzeni nazw instancji
{ }

>>> X.__class__                                    # Klasa instancji
<class '__main__.sub'>

>>> sub.__bases__                                 # Klasa nadziedziona klasa
(<class '__main__.super'>,)

>>> super.__bases__                               # W Pythonie 2.6 pusta krotka ()
(<class 'object'>,)
```

Kiedy klasy przypisują wartości do atrybutów `self`, zapełniają obiekty instancji. Atrybuty znajdują się zatem w słowniku przestrzeni nazw atrybutów instancji, a nie klas. Przestrzeń nazw obiektu instancji zapisuje dane różniące się z instancją na instancję, natomiast `self` jest punktem zaczepienia dla tej przestrzeni nazw.

```
>>> Y = sub()
>>> X.hello()
>>> X.__dict__
{'data1': 'mielonka'}
```

```

>>> X.hola()
>>> X.__dict__
{'data1': 'mielonka', 'data2': 'jajka'}

>>> sub.__dict__.keys()
['__module__', '__doc__', 'hola']

>>> super().__dict__.keys()
['__dict__', '__module__', '__weakref__', 'hello', '__doc__']

>>> Y.__dict__
{}

```

Warto zwrócić uwagę na specjalne nazwy z podwójnymi _ w słownikach klas. Python ustawia je automatycznie. Większość z nich nie jest wykorzystywana w typowych programach, jednak istnieją narzędzia używające niektórych z nich (na przykład __doc__ przechowuje omówione w rozdziale 15. łańcuchy znaków dokumentacji).

Warto również zauważyć, że Y (druga instancja wykonana na początku tej serii) nadal ma na końcu pusty słownik przestrzeni nazw, pomimo że słownik instancji X został zapełniony przez przypisania w metodach. Jak wspomniano wcześniej, każda instancja ma niezależny słownik przestrzeni nazw, który na początku jest pusty i może z czasem zawierać zupełnie inne atrybuty od tych zapisanych przez słowniki przestrzeni nazw innych instancji tej samej klasy.

Ponieważ atrybuty są tak naprawdę wewnątrz Pythona kluczami słowników, istnieją dwa sposoby pobrania i przypisania ich wartości — za pomocą składni kwalifikującej lub indeksowania po kluczu.

```

>>> X.data1, X.__dict__['data1']
('mielonka', 'mielonka')

>>> X.data3 = 'tost'
>>> X.__dict__
{'data1': 'mielonka', 'data3': 'tost', 'data2': 'jajka'}

>>> X.__dict__['data3'] = 'szynka'
>>> X.data3
'szynka'

```

Jest tak jednak jedynie w przypadku atrybutów naprawdę dołączanych do instancji. Ponieważ składnia kwalifikująca pobierająca atrybuty wykonuje również wyszukiwanie dziedziczenia, ma dostęp do atrybutów, którego nie potrafi uzyskać indeksowanie słowników przestrzeni nazw. Przykładowo dostęp do odziedziczonego atrybutu X.hello za pomocą X.__dict__['hello'] nie jest możliwy.

Wreszcie poniżej znajduje się przykład działania wbudowanej funkcji dir, którą widzieliśmy w rozdziałach 4. oraz 15., na obiektach klas i instancji. Funkcja ta działa na każdym obiekcie mającym atrybuty — wywołanie dir(obejkt) jest podobne do obejekt.__dict__.keys(). Warto jednak zauważyć, że funkcja dir sortuje swoją listę i zawiera pewne atrybuty systemowe. Od Pythona 2.2 automatycznie zbiera odziedziczone atrybuty, natomiast w wersji 3.0 zawiera nazwy odziedziczone po klasie object, będącej domniemaną klasą nadzczną wszystkich klas.⁵

⁵ Jak widać, zawartość słownika atrybutów oraz wynik wywołania funkcji dir mogą się z czasem zmieniać. Ponieważ Python pozwala teraz na tworzenie klas podrzędnych z typów wbudowanych, zawartość wyników wywołania funkcji dir dla typów wbudowanych została rozszerzona, tak by obejmować metody przeciążania operatorów, podobnie jak wyniki funkcji dir dla klas zdefiniowanych przez użytkownika w Pythonie 3.0. Generalnie nazwy atrybutów z początkowymi oraz końcowymi podwójnymi znakami _ są specyficzne dla interpretera. Klasa podrzędna typów zostanie omówiona w rozdziale 31.

```

>>> X.__dict__, Y.__dict__
([{'data1': 'mielonka', 'data3': 'szynka', 'data2': 'jajka'}, {})
>>> list(X.__dict__.keys())
['data1', 'data3', 'data2'] # W Pythonie 3.0 wywołanie list jest niezbędne

# W Pythonie 2.6
>>> dir(X)
['__doc__', '__module__', 'data1', 'data2', 'data3', 'hello', 'hola']
>>> dir(sub)
['__doc__', '__module__', 'hello', 'hola']
>>> dir(super)
['__doc__', '__module__', 'hello']

# W Pythonie 3.0
>>> dir(X)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
... pozostałe nazwy pominięto... 'data1', 'data2', 'data3', 'hello', 'hola']

>>> dir(sub)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
... pozostałe nazwy pominięto... 'hello', 'hola']

>>> dir(super)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__',
... pozostałe nazwy pominięto... 'hello']

```

By lepiej poczuć, w jaki sposób przestrzenie nazw wykonują swoje zadania związane z atrybutami, warto poeksperymentować trochę z tymi atrybutami specjalnymi. Nawet jeśli nigdy nie będziemy z nich korzystać w żadnym pisany samodzielnie programie, przekonanie się, że są to normalne słowniki, powinno pomóc odmitologizować pojęcie przestrzeni nazw.

Łącza przestrzeni nazw

W poprzednim podrozdziale wprowadziliśmy atrybuty specjalne instancji oraz klas `__class__` i `__bases__` bez wyjaśnienia, dlaczego mogą one być dla nas istotne. Mówiąc w skrócie, atrybuty te pozwalają nam badać hierarchie dziedziczenia wewnątrz własnego kodu. Można je na przykład wykorzystać do wyświetlenia drzewa klas, jak w poniższym przykładzie.

```

# Plik classtree.py
"""

Wspinaranie się w górę drzewa dziedziczenia za pomocą łączy przestrzeni nazw, wyświetlające wyższe klasy nadzędne
z wcięciem.
"""

def classtree(cls, indent):
    print('.' * indent, cls.__name__)
    for supercls in cls.__bases__:
        classtree(supercls, indent+3) # Wyświetlenie tu nazwy klasy
                                    # Rekurencja po wszystkich klasach nadzędnych
                                    # Może odwiedzić klasę nadziedną więcej niż raz

def instance(tree):
    print('Drzewo', inst)
    classtree(inst.__class__, 3) # Pokazanie instancji
                                # Przejście do jej klasy

def selftest():
    class A: pass
    class B(A): pass
    class C(A): pass
    class D(B,C): pass
    class E: pass
    class F(D,E): pass

```

```
instancetree(B))  
instancetree(F))  
  
if __name__ == '__main__': selftest()
```

Funkcja `classtree` tego skryptu jest *rekurencyjna* — wyświetla nazwę klasy za pomocą atrybutu `__name__`, a następnie przechodzi do klasy nadzędnej, wywołując samą siebie. Pozwala to funkcji na przechodzenie drzew klas o dowolnym kształcie. Rekurencja pozwala przejść do góry i zatrzymuje się na głównych klasach nadzędnych, które mają puste atrybuty `__bases__`. Bez skorzystania z rekurencji każdy aktywny poziom funkcji otrzymuje własną kopię zakresu lokalnego. Tutaj oznacza to, że `cls` oraz `indent` są inne na każdym poziomie `classtree`.

Większość tego pliku jest kodem samosprawdzającym. Kiedy wykona się go jako samodzielny plik w Pythonie 2.6, buduje on puste drzewo klas, wykonuje jego dwie instancje i wyświetla ich strukturę drzew klas.

```
C:\misc> c:\python26\python classtree.py Drzewo <__main__.B instance at 0x02557328> ...  
B  
.... A  
Drzewo <__main__.F instance at 0x02557328> ... F  
.... D  
..... B  
..... A  
..... C  
..... A  
.... E
```

Po wykonaniu w Pythonie 3.0 drzewo zawiera domniemane klasy nadzędne `object`, które automatycznie dodawane są ponad klasami samodzielnymi, ponieważ w tej wersji wszystkie klasy są klasami w nowym stylu (więcej informacji na temat tej zmiany znajduje się w rozdziale 31.).

```
C:\misc> c:\python30\python classtree.py Drzewo <__main__.B object at 0x02810650> ... B  
.... A  
..... object  
Drzewo <__main__.F object at 0x02810650> ... F  
.... D  
..... B  
..... A  
..... object  
..... C  
..... A  
..... object  
.... E  
..... object
```

Indentacja zaznaczona kropkami wykorzystywana jest do oznaczenia wysokości drzewa klas. Moglibyśmy oczywiście poprawić nieco format danych wyjściowych i być może nawet naszkicować je w graficznym interfejsie użytkownika. Nawet w obecnej formie możemy zaimportować te funkcje w dowolnym miejscu, w którym chcemy szybko wyświetlić drzewo klas.

```
C:\misc> c:\python30\python >>> class Emp: pass  
...  
>>> class Person(Emp): pass >>> bob = Person()
```

```
>>> import classtree
>>> classtree.inстансетree(bob)
Drzewo <__main__.Person object at 0x028203B0> ...
    .... Emp
    .... object
```

Bez względu na to, czy będziemy kiedyś korzystać z tych narzędzi, przykład ten demonstruje jeden z wielu sposobów wykorzystania atrybutów specjalnych udostępniających mechanizmy wewnętrzne interpretera. Kolejny zobaczymy, kiedy w pliku *lister.py* będziemy tworzyć uniwersalną klasę z narzędziami do wyświetlanego — w podrozdziale „Dziedziczenie wielokrotne” w rozdziale 30. W przykładzie tym rozszerzymy tę technikę w taki sposób, by wyświetlała ona również atrybuty każdego obiektu drzewa klas. W ostatniej części książki powrócimy do takich narzędzi w kontekście ogólnego budowania narzędzi Pythona w celu zaimplementowania prywatności atrybutów czy sprawdzania poprawności argumentów. Choć nie jest to przeznaczone dla każdego programisty Pythona, dostęp do mechanizmów wewnętrznych daje narzędzia programistyczne o ogromnych możliwościach.

Raz jeszcze o łańcuchach znaków dokumentacji

Ostatni przykład poprzedniego podrozdziału zawiera łańcuch znaków dokumentacji modułu, jednak należy pamiętać, że łańcuchy znaków dokumentacji można wykorzystywać także z przeznaczeniem dla komponentów klas. łańcuchy znaków dokumentacji, omówione szczegółowo w rozdziale 15., są literałami łańcuchów znaków, które pojawiają się na górze różnych struktur i są w Pythonie automatycznie zapisywane w atrybutach `__doc__` odpowiadających im obiektów. Działa to dla plików modułów, instrukcji `def` funkcji oraz klas i metod.

Skoro już wiemy coś więcej o klasach i metodach, plik *docstr.py* jest krótkim, choć wyczerpującym przykładem pokazującym miejsca, w których łańcuchy znaków dokumentacji mogą się pojawić w kodzie. Mogą się one znaleźć się w blokach z potrójnymi znakami cudzysłowów lub apostrofów.

```
"Jestem: docstr.__doc__"

def func(args):
    "Jestem: docstr.func.__doc__"
    pass

class spam:
    "Jestem: spam.__doc__ lub docstr.spam.__doc__"
    def method(self, arg):
        "Jestem: spam.method.__doc__ lub self.method.__doc__"
        pass
```

Podstawową zaletą łańcuchów znaków dokumentacji jest to, że są one widoczne w czasie wykonywania. W ten sposób jeśli coś zostało zapisane w kodzie jako łańcuch znaków dokumentacji, można w składni kwalifikującej zapisać obiekt oraz jego atrybut `__doc__` w celu pobrania jego dokumentacji.

```
>>> import docstr
>>> docstr.__doc__
'Jestem: docstr.__doc__'

>>> docstr.func.__doc__
'Jestem: docstr.func.__doc__'
```

```
>>> docstr.spam.__doc__
'Jestem: spam.__doc__ lub docstr.spam.__doc__'

>>> docstr.spam.method.__doc__
'Jestem: spam.method.__doc__ lub self.method.__doc__'
```

Omówienie narzędzia *PyDoc*, które potrafi formatować te łańcuchy znaków w raportach, znajduje się w rozdziale 15. Poniżej zaprezentowano wykonanie naszego kodu w Pythonie 2.6. Python 3.0 pokazuje dodatkowe atrybuty odziedziczone po domniemanej klasie nadzędnej `object` z modelu klas w nowym stylu. Można samodzielnie wykonać ten kod w Pythonie 3.0 w celu zobaczenia dodatków z tej wersji; więcej informacji na temat tej różnicy znajduje się w rozdziale 31.

```
>>> help(docstr)
Help on module docstr:

NAME
    docstr - Jestem: docstr.__doc__

FILE
    c:\misc\docstr.py

CLASSES
    spam

    class spam
        | Jestem: spam.__doc__ lub docstr.spam.__doc__
        |
        | Methods defined here:
        |
        |     method(self, arg)
        |     Jestem: spam.method.__doc__ lub self.method.__doc__

FUNCTIONS
    func(args)
        Jestem: docstr.func.__doc__
```

Łańcuchy znaków dokumentacji dostępne są w czasie wykonywania, jednak są one — ze składniowego punktu widzenia — mniej elastyczne od komentarzy ze znakami `#` (które mogą się pojawić w dowolnym miejscu programu). Obie formy są przydatnymi narzędziami, a każda dokumentacja programu jest dobra (o ile, oczywiście, jest poprawna!). Jako ogólną regułę należy wykorzystywać łańcuchy znaków dokumentacji na cele dokumentacji funkcjonalnej (wyjaśniającej, co robią obiekty), natomiast komentarze ze znakami `#` na cele dokumentacji poziomu mikro (mówiącej, jak działają niektóre bardziej zagmatwane wyrażenia).

Klasy a moduły

Zakończymy ten rozdział, porównując krótko zagadnienia z dwóch ostatnich części niniejszej książki, poświęconych modułom oraz klasom. Ponieważ obie te kategorie dotyczą przestrzeni nazw, rozróżnienie ich może być mylące. W skrócie:

- Moduły
 - są pakietami danych oraz logiki,
 - są tworzone przez pisanie plików Pythona lub rozszerzeń języka C,
 - są wykorzystywane, kiedy są importowane.

- Klasy
 - implementują nowe obiekty,
 - są tworzone przez instrukcję `class`,
 - są wykorzystywane, kiedy są wywoływane,
 - zawsze istnieją wewnątrz modułu.

Klasy obsługują również dodatkowe opcje, których nie mają moduły, takie jak przeciążanie operatorów, generowanie wielu instancji oraz dziedziczenie. Choć zarówno klasy, jak i moduły są przestrzeniami nazw, powinniśmy już widzieć, że są zupełnie różnymi elementami.

Podsumowanie rozdziału

Niniejszy rozdział zabrał nas na drugą, bardziej pogłębioną wycieczkę po mechanizmach programowania zorientowanego obiektowo języka Python. Dowiedzieliśmy się więcej o klasach, metodach oraz dziedziczeniu. Zakończyliśmy również omawianie przestrzeni nazw w Pythonie, rozszerzając je w taki sposób, by uwzględnić ich zastosowanie w klasach. Po drodze przyrzeliśmy się pewnym bardziej zaawansowanym zagadnieniom, takim jak abstrakcyjne klasy nadzędne, atrybuty danych klas, słowniki i łącza przestrzeni nazw oraz ręczne wywoływanie metod i konstruktorów klas nadzędnych.

Skoro już wiemy tyle o mechanizmach tworzenia klas w kodzie Pythona, w rozdziale 29. przejdziemy do pewnego szczególnego aspektu tych mechanizmów — przeciążania operatorów. Potem omówimy często wykorzystywane wzorce projektowe — niektóre sposoby wykorzystywania klas oraz łączenia ich w celu zoptymalizowania ponownego użycia kodu. Przed przejściem dalej nie należy jednak zapominać o wykonaniu quizu podsumowującego rozdział.

Sprawdź swoją wiedzę — quiz

1. Co to jest abstrakcyjna klasa nadzędna?
2. Co się dzieje, kiedy prosta instrukcja przypisania znajduje się na najwyższym poziomie instrukcji `class`?
3. Po co klasa miałaby ręcznie wywoływać metodę `__init__` z klasy nadzędnej?
4. W jaki sposób można rozszerzać, zamiast całkowicie zastępować, odziedziczoną metodę?
5. Jak się nazywa stolica Asyrii?

Sprawdź swoją wiedzę — odpowiedzi

1. Abstrakcyjna klasa nadzędna to klasa wywołująca metodę, jednak nie dziedzicząca ani nie definiującą jej — oczekuje ona, że metoda zostanie uzupełniona przez klasę podrzędną. Często wykorzystywana jest jako sposób uogólniania klas, kiedy jakieś zachowanie nie może zostać przewidziane aż do momentu zapisania w kodzie bardziej specyficznej klasy

podrzędnej. Platformy zorientowane obiektowo wykorzystują ten sposób do udostępniania operacji definiowanych przez klienta i możliwych do dostosowania do własnych potrzeb.

2. Kiedy prosta instrukcja przypisania (`X = Y`) pojawia się na najwyższym poziomie instrukcji `class`, dołącza ona atrybut danych do klasy (`Class.X`). Tak jak wszystkie atrybuty klas, będzie on współdzielony przez wszystkie instancje. Atrybuty danych nie są jednak wywoływalnymi funkcjami metod.
3. Klasa musi ręcznie wywoływać metodę `__init__` w klasie nadrzędnej, jeśli definiuje własny konstruktor `__init__`, ale nadal musi także uruchomić kod konstruktora klasy nadrzędnej. Python sam automatycznie wykonuje tylko jeden konstruktor — znajdujący się najbliżej w drzewie. Konstruktory klas nadrzędnych wywoływanie są za pośrednictwem nazwy klasy, z ręcznym przekazaniem instancji `self`. `Superclass.__init__(self, ...)`.
4. By odziedziczoną metodę rozszerzyć, zamiast całkowicie zastąpić, należy redefiniować ją w klasie podrzędnej i z tej nowej wersji wykonać ręczne wywołanie wersji metody z klasy nadrzędnej. Polega to na ręcznym przekazaniu instancji `self` do wersji metody z klasy nadrzędnej za pomocą kodu `Superclass.method(self, ...)`.
5. Aszur (lub Kalaat Szirkat), Kalchu (lub Nimrud) lub przez krótki czas Dur-Szarrukin (lub Chorsabad), a na końcu Niniwa.

Przeciążanie operatorów

Niniejszy rozdział stanowi kontynuację przeglądu mechanizmów klas w Pythonie. Teraz będziemy zajmować się tematyką przeciążania operatorów. W poprzednich rozdziałach mieliśmy kilka przykładów przeciążania operatorów, w niniejszym rozdziale poznamy więcej szczegółów i przyjrzymy się przykładom powszechnych zastosowań tego mechanizmu. Nie będziemy opisywać szczegółowo wszystkich dostępnych metod, ale te, które omówimy, stanowią reprezentatywną próbę możliwości tego mechanizmu klas w Pythonie.

Podstawy

Przeciążanie operatorów to w rzeczywistości mechanizm *przechwytywania* wbudowanych metod klas: Python automatycznie wywołuje metody zdefiniowane przez użytkownika, gdy instancje klas występują w kontekście wbudowanych operatorów, a wartość zwracana z metody staje się następnie wynikiem operacji. Przeciążanie operatorów działa zgodnie z kilkoma podstawowymi koncepcjami:

- przeciążanie operatorów pozwala klasom przechwytywać operacje Pythona,
- klasy mogą przeciągać wszystkie operacje wyrażeń w Pythonie,
- klasy mogą również przeciągać wbudowane operacje, jak wyświetlanie znaków, wywołania funkcji, dostęp do atrybutów itp.,
- przeciążanie operatorów pozwala klasom definiowanym przez użytkownika działać w sposób zbliżony do typów wbudowanych,
- przeciążanie jest implementowane przez definiowanie metod klas o specjalnych nazwach.

Innymi słowy, gdy Python znajdzie w klasie metodę o nazwie odpowiedniej dla wykonywanego wyrażenia, wywoła ją i jej wynik zwróci jako wynik tego wyrażenia. Jak już wiemy, metody specjalne implementujące przeciążanie operatorów nigdy nie są wymagane w definicji klasy, nie ma też ich „domyślnych” implementacji: jeśli klasa nie ma zdefiniowanej takiej metody i nie dziedziczy jej po klasach nadrzędnych, oznacza to, że nie obsługuje danej operacji. W przypadku gdy jednak metody specjalne są zdefiniowane, pozwalają klasom emulować interfejsy obiektów wbudowanych, dzięki czemu implementacje mogą być bardziej spójne.

Konstruktory i wyrażenia — `__init__` i `__sub__`

Rozważmy następujący przykład: klasa `Number` zdefiniowana w pliku `number.py` udostępnia metodę przechwytyującą konstrukcję instancji (`__init__`) oraz drugą, implementującą wyrażenia odejmowania (`__sub__`). Te specjalne metody stanowią punkt zaczepienia pozwalający na użycie instancji klas w kontekście wbudowanych operatorów Pythona.

```
class Number:  
    def __init__(self, start):  
        self.data = start  
    def __sub__(self, other):  
        return Number(self.data - other)  
  
>>> from number import Number  
>>> X = Number(5) # Wywoływana przy Number(start)  
>>> Y = X - 2 # Wywoływana przy instancja - inna  
>>> Y.data # Wynik jest nową instancją  
>>> # Załadowanie klasy z modułu  
>>> # Number.__init__(X, 5)  
>>> # Number.__sub__(X, 2)  
>>> # Y jest nową instancją klasy Number  
>>> 3
```

Jak wspominałem w poprzednich rozdziałach, metoda `__init__` jest najczęstszym przykładem przeciążania operatorów w Pythonie, występuje w większości definiowanych klas. W niniejszym rozdziale przyjrzymy się kilku innym dostępnym metodom oraz ich zastosowaniom w przykładach powszechnego użycia.

Często spotykane metody przeciążania operatorów

Prawie każde działanie, które możemy wykonać na obiektach wbudowanych, takich jak liczby całkowite czy listy, ma odpowiadającą mu metodę przeciążającą o specjalnej nazwie do zastosowania w klasach. W tabeli 29.1 zaprezentowano kilka najczęściej stosowanych metod, ale jest ich o wiele więcej. Tak naprawdę wiele z metod przeciążania operatorów ma kilka wersji (jak `__add__`, `__radd__` oraz `__iadd__` dla dodawania). Wyczerpującą listę dostępnych specjalnych nazw metod można znaleźć w innych książkach poświęconych Pythonowi lub w dokumentacji języka.

Wszystkie metody przeciążania operatorów mają nazwy rozpoczynające się od dwóch znaków `_i` kończące się w ten sam sposób, co pozwala odróżnić je od innych nazw definiowanych w klasach. Odwzorowania ze specjalnych nazw metod na wyrażenia lub operacje są zdefiniowane w języku Python (i opisane w standardowej dokumentacji języka). Nazwa `__add__` zawsze odwzorowywana jest na wyrażenie z operatorem `+`, zgodnie z definicją języka, bez względu na to, co tak naprawdę robi kod metody `__add__`.

Metody przeciążania operatorów mogą być dziedziczone z klas nadzujących, tak samo jak wszelkie inne metody. Wszystkie metody przeciążania operatorów są opcjonalne — jeśli którejś nie umieścimy w kodzie, operacja ta będzie po prostu nieobsługiwana w naszej klasie (i może powodować zwrócenie wyjątku, jeśli spróbujemy jej użyć). Niektóre powszechnie stosowane operacje, jak wyświetlanie, posiadają implementację domyślną, a dokładniej: są dziedziczone po klasie `object` (która jest dziedziczona przez wszystkie klasy w Pythonie 3.0), ale większość operatorów nie zadziała na instancjach klas, jeśli nie są w nich jawnie zdefiniowane odpowiednie metody przeciążające.

Tabela 29.1. Często wykorzystywane metody przeciążania operatorów

Metoda	Przeciąża	Wywoływana dla
<code>__init__</code>	Konstruktor	Tworzenie obiektu — <code>X = Klasa(args)</code>
<code>__del__</code>	Destruktor	Zwolnienie obiektu <code>X</code>
<code>__add__</code>	Operator <code>+</code>	<code>X + Y, X += Y</code> , jeśli nie ma <code>__iadd__</code>
<code>__or__</code>	Operator <code> </code> (OR poziomu bitowego)	<code>X Y, X = Y</code> , jeśli nie ma <code>__ior__</code>
<code>__repr__, __str__</code>	Wyświetlanie, konwersje	<code>print X, repr(X), str(X)</code>
<code>__call__</code>	Wywołania funkcji	<code>X(*args, **kargs)</code>
<code>__getattr__</code>	Kwalifikacja	<code>X.niezdefiniowany_atrybut</code>
<code>__setattr__</code>	Przypisanie atrybutów	<code>X.atrybut = wartość</code>
<code>__delattr__</code>	Usuwanie atrybutu	<code>del X.atrybut</code>
<code>__getattribute__</code>	Przechwytywanie atrybutu	<code>X.atrybut</code>
<code>__getitem__</code>	Indeksowanie, wycinanie, iteracje	<code>X[klucz], X[i:j], pętle for oraz inne iteracje, jeśli nie ma __iter__</code>
<code>__setitem__</code>	Przypisanie indeksu i wycinka	<code>X[klucz] = wartość, X[i:j] = sekwencja</code>
<code>__delitem__</code>	Usuwanie indeksu i wycinka	<code>del X[klucz], del X[i:j]</code>
<code>__len__</code>	Długość	<code>len(X)</code> , testy prawdziwości, jeśli nie ma <code>__bool__</code>
<code>__bool__</code>	Testy logiczne	<code>bool(X)</code> , testy prawdziwości (odpowiednik <code>__nonzero__</code> w 2.6)
<code>__lt__, __gt__, __le__, __ge__, __eq__, __ne__</code>	Porównania	<code>X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y</code> (lub inaczej <code>__cmp__</code> — tylko w przypadku 2.6)
<code>__radd__</code>	Prawostronny operator <code>+</code>	<code>Nieinstancja + X</code>
<code>__iadd__</code>	Dodawanie w miejscu (rozszerzone)	<code>X += Y</code> (lub inaczej <code>__add__</code>)
<code>__iter__, __next__</code>	Konteksty iteracyjne	<code>I=iter(X), next(I); pętle for, jeśli nie ma __contains__, testy in, wszystkie listy składane, funkcje map (F,X), inne (<code>__next__</code> — odpowiednik <code>next</code> w 2.6)</code>
<code>__contains__</code>	Test przynależności	<code>item in X (dowolny iterator)</code>
<code>__index__</code>	Wartość całkowita	<code>hex(X), bin(X), oct(X), 0[X], 0[X:]</code> (zastępuje <code>__oct__</code> , <code>__hex__</code> itp. z Pythona 2.X)
<code>__enter__, __exit__</code>	Menedżer kontekstu (rozdział 33.)	<code>with obj as var:</code>
<code>__get__, __set__, __delete__</code>	Atrybuty deskryptorów (rozdział 37.)	<code>X.attr, X.attr = value, del X.attr</code>
<code>__new__</code>	Tworzenie instancji (rozdział 39.)	Tworzenie instancji, przed <code>__init__</code>

Większość metod przeciążania operatorów wykorzystywana jest jedynie w zaawansowanych programach wymagających, by obiekty zachowywały się jak te wbudowane, konstruktor `__init__` pojawia się jednak w większości klas. Z metodą konstruktora `__init__`, a także kilkoma innymi metodami z tabeli 29.1 spotkaliśmy się już wcześniej. Przedstawmy teraz jednak niektóre z dodatkowych metod tabeli na przykładach.

Indeksowanie i wycinanie — `__getitem__` i `__setitem__`

Jeśli w klasie jest zdefiniowana (lub dziedziczona po klasie nadrzędnnej) metoda `__getitem__`, będzie ona automatycznie użyta w przypadkach prób wydobycia elementów po indeksach. Na przykład jeśli klasa `X` wystąpi w kontekście indeksowania `X[i]`, Python wywoła jej metodę `__getitem__`, przekazując `X` w pierwszym argumencie, a indeks w drugim. Poniższa klasa zwraca kwadrat wartości indeksu:

```
>>> class Indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = Indexer()
>>> X[2]                                # X[i] wywołuje X.__getitem__(i)
4

>>> for i in range(5):
...     print(X[i], end=' ')
...                                         # Przy każdej iteracji wywołuje __getitem__(X, i)
...
0 1 4 9 16
```

Wycinki

Co interesujące, metoda `__getitem__` jest wywoływana również w *wyrażeniach wycinania*. Wbudowane typy obsługują wycinanie w ten sam sposób. Poniższy listing przedstawia operację wycinania przy wykorzystaniu dolnego i górnego zakresu oraz argumentu przesunięcia (więcej informacji na temat operacji wycinania można znaleźć w rozdziale 7.).

```
>>> L = [5, 6, 7, 8, 9]
>>> L[2:4]                                # Wycinanie z użyciem składni wycinków
[7, 8]
>>> L[1:]
[6, 7, 8, 9]
>>> L[:-1]
[5, 6, 7, 8]
>>> L[::2]
[5, 7, 9]
```

W rzeczywistości parametry wycinania są pakowane w specjalny *obiekt wycinka*, który jest przekazywany do instancji listy. Obiekt wycinka można przekazać ręcznie: składanie wycinków to jedynie składniowy skrót, ułatwiający pracę z obiektami wycinków.

```
>>> L[slice(2, 4)]                      # Wycinanie z użyciem obiektów
[7, 8]
>>> L[slice(1, None)]
[6, 7, 8, 9]
>>> L[slice(None, 1)]
[5, 6, 7, 8]
>>> L[slice(None, None, 2)]
[5, 7, 9]
```

Ta obserwacja ma znaczenie dla klas implementujących metodę `__getitem__`, która jest wywoływana przy operacji indeksowania (wówczas otrzyma ona liczbę całkowitą) oraz wycinania (otrzyma obiekt wycinka). Nasz poprzedni przykład nie obsługuje wycinania, ponieważ metoda `__getitem__` zakłada, że otrzymała liczbę całkowitą, co naprawiamy w poniższym przykładzie. W przypadku indeksowania argument metody `__getitem__` jest liczbą całkowitą, jak poprzednio:

```

>>> class Indexer:
...     data = [5, 6, 7, 8, 9]
...     def __getitem__(self, index):
...         print('getitem:', index)
...         return self.data[index]
...
>>> X = Indexer()
>>> X[0]                                # Wywoywany przy indeksowaniu i wycinaniu
getitem: 0
5
>>> X[1]
getitem: 1
6
>>> X[-1]
getitem: 1
9

```

W przypadku operacji wycinania nasza metoda otrzymuje obiekt wycinka, który po prostu przekazujemy do osadzonego obiektu data w nowym wyrażeniu indeksowania:

```

>>> X[2:4]                                # Wycinanie wysyla metodzie __getitem__ obiekt wycinka
getitem: slice(2, 4, None)
[7, 8]
>>> X[1:]
getitem: slice(1, None, None)
[6, 7, 8, 9]
>>> X[:-1]
getitem: slice(None, 1, None)
[5, 6, 7, 8]
>>> X[::-2]
getitem: slice(None, None, 2)
[5, 7, 9]

```

Metoda `__setitem__` implementuje analogiczny mechanizm przypisywania wartości zarówno dla indeksów, jak i dla wycinków — w tym ostatnim przypadku również otrzymuje obiekt wycinka, który można przekazać dalej tak samo jak w przypadku metody `__getitem__`.

```

def __setitem__(self, index, value):      # Przechwytyuje przypisania do indeksu lub wycinka
    ...
    self.data[index] = value             # Przypisanie do indeksu lub wycinka

```

W rzeczywistości metoda `__getitem__` jest wywoływana również w innych kontekstach, co omówimy w następnym punkcie.

Iteracja po indeksie — `__getitem__`

Istnieje sztuczka, która może być trudna do odgadnięcia dla początkujących, ale okazuje się niezwykle użyteczna. Instrukcja `for` przy każdej iteracji odczytuje jeden element, wykorzystując kolejny indeks, licząc od zera, aż zostanie wywołany wyjątek końca zakresu. Z tego powodu metoda `__getitem__` może być jednym ze sposobów implementacji iteratora w Pythonie: jeśli pętla `for` wykryje tę metodę, będzie ją wywoływać z kolejnymi indeksami. Metoda `__getitem__` jest więc przykładem transakcji: „Kup jeden, drugi dostaniesz gratis” — każdy obiekt obsługujący indeksowanie potrafi również obsługiwać iteracje:

```

>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
>>> X = stepper()                         # X jest instancją klasy stepper

```

Wycinanie i indeksowanie w Pythonie 2.6

Przed wersją 3.0 klasy mogły implementować również metody `__getslice__` i `__setslice__`, które odpowiadały za odczyt i przypisywanie wycinków. Metodom tym przekazywane były zakresy wycinków, a ich stosowanie było zalecane w kontekście wycinków zamiast używania metod `__getitem__` i `__setitem__`.

Te metody zostały usunięte w Pythonie 3.0 na rzecz bardziej ogólnego użycia metod `__getitem__` i `__setitem__`. W większości klas obydwa przypadki można obsłużyć bez konieczności stosowania specjalnego kodu, ponieważ operacje indeksowania obsługują obiekty wycinków w nawiasach kwadratowych (jak w naszym przykładzie). Więcej przykładów przeciążania operacji wycinania można znaleźć w punkcie „Test przynależności — `__contains__`, `__iter__` i `__getitem__`”.

Nie należy mylić metody `__index__` Pythona 3.0 (której nazwa rzeczywiście wprowadza w błąd) z przeciążaniem operacji indeksowania. Metoda ta służy bowiem do przekształcania obiektu na liczbę całkowitą i jest wykorzystywana przez funkcje przekształcające obiekt na liczbę lub tekstową reprezentację notacji szesnastkowej, dwójkowej lub ósemkowej.

```
>>> class C:
...     def __index__(self):
...         return 255
...
>>> X = C()
>>> hex(X)                      # Wartość całkowita
'0xff'
>>> bin(X)
'0b11111111'
>>> oct(X)
'0o377'
```

Choć ta metoda nie jest stosowana w przypadku dostępu do elementów obiektu po indeksie, to jednak ma związek z indeksowaniem — jest bowiem używana w sytuacji, gdy dany obiekt zostanie użyty w kontekście wymagającym liczby całkowitej, jak na przykład wartość indeksu:

```
>>> ('C' * 256)[255]
'C'
>>> ('C' * 256)[X]            # Indeks po X (nie X[i])
'C'
>>> ('C' * 256)[X:]          # Indeks po X (nie X[i:])
'C'
```

Ta metoda w 2.6 działa tak samo, z tą różnicą, że nie jest wywoływana przez funkcje `hex` i `oct` (w 2.6 należy w takim przypadku zdefiniować metody specjalne `__hex__` i `__oct__`).

```
>>> X.data = "Mielonka"
>>>
>>> X[1]                      # Indeksowanie wywołuje __getitem__
'p'
>>> for item in X:           # Pętle for wywołują __getitem__
...     print(item, end=' ')
...
M i e l o n k a
```

Tak naprawdę mamy tu przykład sytuacji: „Kup jeden, resztę dostaniesz gratis” — każda klasa obsługująca pętle `for` automatycznie obsługuje dowolne konteksty iteracyjne w Pythonie; wiele z nich poznaliśmy już we wcześniejszych rozdziałach (konteksty iteracyjne były prezentowane w rozdziale 14.). Na przykład test przynależności, listy składane, funkcja wbudowana `map`, przypisania list i krotek oraz w konstruktory typów automatycznie wywołują metodę `__getitem__`, jeśli jest zdefiniowana.

```

>>> 'p' in X                                # Wszystkie przykłady wywołują __getitem__
True

>>> [c for c in X]                         # Lista składana
['S', 'P', 'A', 'M']

>>> list(map(str.upper, X))                 # Funkcja (w 3.0 niejawnie wywołuje list())
['S', 'P', 'A', 'M']

>>> (a, b, c, d) = X                      # Przypisania sekwencji
>>> a, c, d
('S', 'A', 'M')

>>> list(X), tuple(X), ''.join(X)
(['S', 'P', 'A', 'M'], ('S', 'P', 'A', 'M'), 'Spam')

>>> X
<__main__.stepper object at 0x00A8D5D0>

```

W praktyce ta technika może być użyta do tworzenia obiektów udostępniających interfejs sekwencji oraz do definiowania dodatkowej logiki dla wbudowanych operatorów typów sekwencyjnych. Do tej koncepcji wróćmy przy okazji rozszerzania typów wbudowanych w rozdziale 31.

Obiekty iteratorów — `__iter__` i `__next__`

Mimo że metoda `__getitem__` z poprzedniego podrozdziału działa prawidłowo, jest jedynie mechanizmem „awaryjnym”, stosowanym przy iteracjach w przypadku braku lepszych metod. Wszystkie konteksty iteracyjne w nowszych wersjach Pythona w pierwszej kolejności próbują wywołać metodę `__iter__`, a jeśli jej nie znajdą, próbują `__getitem__`. Innymi słowy, protokół iteracyjny opisany w rozdziale 14. jest preferowany i ma przewagę nad kolejnym odczytywaniem elementów po indeksach. Jeśli obiekt nie obsługuje protokołu iteracyjnego, program próbuje odwołać się do indeksów. W większości przypadków również programista powinien preferować metodę `__iter__`: pozwala ona uogólnić konteksty iteracyjne w lepszy sposób, niż to jest możliwe z użyciem `__getitem__`.

Konteksty iteracyjne działają przez wywoływanie wbudowanej funkcji `iter`, wywołującej metodę `__iter__` obiektu iterowanego, która z kolei powinna zwracać iterator. Jeśli tak się stanie, Python w kolejnych iteracjach wywołuje metodę `__next__` iteratora zwracającą kolejne elementy, aż zostanie wywołany wyjątek `StopIteration`. Jeśli metoda `__iter__` nie zostanie odnaleziona w obiekcie, Python przełącza się w tryb wykorzystania metody `__getitem__`, w której kolejne elementy są pobierane przez pobieranie obiektów o kolejnych indeksach, aż zostanie wywołany wyjątek `IndexError`. Do obsługi iteratorów wykorzystywana jest funkcja wbudowana `next`, która może służyć do ręcznej iteracji w iteratorze. Wywołanie `next(I)` jest równoważne wywołaniu `I.__next__()`.



Uwaga na temat wersji: Jak wspominałem w rozdziale 14., w Pythonie 2.6 metoda `I.__next__()` jest nazwana `I.next()`. Funkcja wbudowana `next(I)` została wprowadzona w celu zwiększenia przenośności kodu: w Pythonie 2.6 wywołuje `I.next()`, natomiast w 3.0 `I.__next__()`. W pozostałych szczegółach iteratory działają w 2.6 tak samo jak w 3.0.

Iteratory zdefiniowane przez użytkownika

Po zdefiniowaniu mechanizmu `__iter__` klasy stają się iteratorami zdefiniowanymi przez użytkownika, implementując po prostu protokół iteracji przedstawiony w rozdziałach 14. oraz 20. (można w nich znaleźć więcej informacji na temat iteratorów). W poniższym pliku o nazwie `iters.py` umieszczono na przykład klasę zdefiniowanego przez użytkownika iteratora, która generuje kwadrat liczb.

```
# -*- coding: utf-8 -*-
class Squares:
    def __init__(self, start, stop):
        self.value = start - 1
        self.stop = stop
    def __iter__(self):
        return self
    def next(self):
        if self.value == self.stop:
            raise StopIteration
        self.value += 1
        return self.value ** 2

% python
>>> from iters import Squares
>>> for i in Squares(1, 5):
...     print(i, end=' ')
...
1 4 9 16 25
```

W powyższym kodzie obiektem iteratora jest po prostu instancja `self`, ponieważ metoda `next` jest częścią tej klasy. W bardziej skomplikowanych scenariuszach obiekt iteratora może być zdefiniowany jako osobna klasa i obiekt z własnymi informacjami o stanie, co pozwala na obsługę wielu aktywnych iteracji po tych samych danych (przykład takiej sytuacji zobaczymy za moment). Koniec iteracji sygnalizowany jest za pomocą instrukcji `raise` Pythona (więcej informacji na temat zgłoszania wyjątków znajdzie się w kolejnej części książki). Ręczne iteracje działają również dla typów wbudowanych:

```
>>> X = Squares(1, 5)                      # Ręczna iteracja: tak działają pętle
>>> I = iter(X)                            # iter wywołuje __iter__
>>> next(I)                                # next wywołuje __next__
1
>>> next(I)
4
...pomińnięta część wyniku...
>>> next(I)                                # Można zastosować w instrukcji try:
25
>>> next(I)
StopIteration
```

Odpowiednik tego kodu, wykorzystujący metodę `__getitem__`, może być mniej naturalny, ponieważ pętla `for` wykonywałaby iterację po wszystkich wartościach przesunięcia od zera w górę. Przekazane wartości przesunięcia byłyby jedynie pośrednio związane z podanym przedziałem wartości ($0..N$ musiałoby zostać odwzorowane na `start..stop`). Ponieważ obiekty `__iter__` zachowują zarządzany w jawnym sposób stan pomiędzy wywołaniami metody `next`, mogą być bardziej uniwersalne od metody `__getitem__`.

Z drugiej strony, iteratory oparte na metodzie `__iter__` mogą czasami być bardziej skomplikowane i mniej wygodne od `__getitem__`. Są właściwie zaprojektowane pod kątem iteracji, a nie dowolnego indeksowania — tak naprawdę wcale nie przeciążają one wyrażenia indeksującego.

```
>>> X = Squares(1, 5)
>>> X[1]
AttributeError: Squares instance has no attribute '__getitem__'
```

Model z `__iter__` jest również implementacją wszystkich pozostałych kontekstów iteracyjnych, jakie widzieliśmy w przypadku metody `__getitem__` (testów przynależności, konstruktorów typów czy przypisania sekwencji). W przeciwieństwie do `__getitem__`, metoda `__iter__` zaprojektowana została do jednego przejścia, a nie wielu. Klasa `Squares` jest na przykład jednorazową iteracją — po przejściu jest pusta. Dla każdej nowej iteracji musimy utworzyć nowy obiekt iteratora.

```
>>> X = Squares(1, 5)
>>> [n for n in X]                                # Wyczerpuje elementy
[1, 4, 9, 16, 25]
>>> [n for n in X]                                # Teraz jest pusta
[]
>>> [n for n in Squares(1, 5)]                   # Utworzenie nowego obiektu iteratora
[1, 4, 9, 16, 25]
>>> list(Squares(1, 3))
[1, 4, 9]
```

Warto zauważyć, że przykład ten prawdopodobnie byłby prostszy, gdyby zapisano go z funkcjami generatora (zagadnieniem przedstawionym w rozdziale 20. i powiązanym z iteratorami).

```
>>> def gsquares(start, stop):
...     for i in range(start, stop+1):
...         yield i ** 2
...
>>> for i in gsquares(1, 5):                      # or: (x ** 2 for x in range(1, 5))
...     print(i, end=' ')
...
1 4 9 16 25
```

W przeciwieństwie do klasy, funkcja automatycznie zapisuje stan pomiędzy iteracjami. Oczywiście w tym sztucznym przykładzie moglibyśmy tak naprawdę pominąć obie techniki i po prostu użyć pętli `for`, funkcji `map` czy listy składanej w celu zbudowania całej listy za jednym razem. Najlepsza i najszybsza metoda wykonania jakiegoś zadania w Pythonie jest często najłatwiejsza.

```
>>> [x ** 2 for x in range(1, 6)]
[1, 4, 9, 16, 25]
```

Klasy mogą się jednak lepiej nadawać do modelowania bardziej skomplikowanych iteracji, w szczególności kiedy mogą one skorzystać z informacji o stanie oraz hierarchii dziedziczenia. Poniżej omawiamy jeden taki przypadek użycia.

Wiele iteracji po jednym obiekcie

Wspomniałem wcześniej, że obiekt iteratora można zdefiniować jako osobną klasę z własnymi informacjami o stanie, co pozwala na obsługę wielu aktywnych iteracji po tych samych danych. Warto rozważyć, co się stanie, kiedy przejdziemy typ wbudowany, taki jak łańcuch znaków.

```
>>> S = 'ace'
>>> for x in S:
...     for y in S:
```

```

...     print x + y,
...
aa ac ae ca cc ce ea ec ee

```

W powyższym kodzie zewnętrzna pętla `for` pobiera iterator z łańcucha znaków, wywołując `iter`, a każda pętla zagnieżdzona robi to samo w celu otrzymania niezależnego iteratora. Ponieważ każdy aktywny iterator ma swoje własne informacje o stanie, każda pętla może zachować swoją własną pozycję w łańcuchu znaków, niezależnie od pozostałych aktywnych pętli.

Odpowiednie przykłady mieliśmy okazję poznać wcześniej, w rozdziałach 14. i 20. Na przykład funkcje i wyrażenia generatorów, jak również funkcje wbudowane `map` i `zip`, są w rzeczywistości obiektami generatorów, natomiast funkcja wbudowana `range` oraz typy wbudowane, jak listy, obsługują wielokrotne iteratory działające niezależnie od siebie, które mogą znajdować się w różnych pozycjach sekwencji.

Jeśli samodzielnie definiujemy własne iteratory w postaci klas, możemy zdecydować, czy chcemy obsługiwać pojedynczą iterację, czy wielokrotną. By uzyskać ten sam efekt za pomocą iteratorów zdefiniowanych przez użytkownika, `__iter__` musi po prostu zdefiniować nowy obiekt stanu dla iteratora, zamiast zwracać `self`.

Poniższy kod definiuje na przykład klasę iteratora, która w czasie iteracji pomija co drugi element. Ponieważ obiekt iteratora tworzony jest od nowa dla każdej iteracji, obsługuje większą liczbę aktywnych pętli.

```

class SkipIterator:
    def __init__(self, wrapped):
        self.wrapped = wrapped
        self.offset = 0
    def next(self):
        if self.offset >= len(self.wrapped):      # Informacje o stanie iteratora
            raise StopIteration
        else:
            item = self.wrapped[self.offset]      # Inaczej zwrócenie elementu i pominięcie
            self.offset += 2
            return item

class SkipObject:
    def __init__(self, wrapped):                  # Zapisanie elementu, który ma być użyty
        self.wrapped = wrapped
    def __iter__(self):                         # Za każdym razem nowy iterator
        return SkipIterator(self.wrapped)

if __name__ == '__main__':
    alpha = 'abcdef'
    skipper = SkipObject(alpha)
    I = iter(skipper)                         # Utworzenie obiektu pojemnika
    for i in range(3):                         # Utworzenie na nim iteratora
        print I.next(), I.next(), I.next()       # Odwiedzenie wartości przesunięcia 0, 2, 4

    for x in skipper:                         # for automatycznie wywołuje __iter__
        for y in skipper:                     # Zagnieżdzone for za każdym razem wywołują __iter__
            print x + y,                      # Każdy iterator ma własny stan i przesunięcie

```

Po wykonaniu przykład ten działa jak zagnieżdzone pętle z wbudowanymi łańcuchami znaków — każda aktywna pętla ma własną pozycję w łańcuchu znaków, ponieważ każda uzyskuje niezależny obiekt iteratora, zapisujący jej własne informacje o stanie.

```

% python skipper.py
a c e
aa ac ae ca cc ce ea ec ee

```

W przeciwnieństwie do tego rozwiązania nasz przykład z klasą Squares obsługuje tylko jedną aktywną iterację, o ile nie wywołamy tej klasy ponownie w zagnieżdżonych pętlach w celu otrzymania nowych obiektów. Tutaj jest tylko jeden obiekt klasy SkipObject z wieloma utworzonymi z niego obiektami iteratora.

Jak wcześniej, podobne rezultaty moglibyśmy uzyskać za pomocą narzędzi wbudowanych — na przykład wykorzystując wycinek z trzecim parametrem do pomijania elementów.

```
>>> S = 'abcdef'  
>>> for x in S[::2]:  
...     for y in S[::2]:  
...         print x + y,  
...  
aa ac ae ca cc ce ea ec ee
```

Nowe obiekty w każdej iteracji

Nie jest to jednak to samo — z dwóch powodów. Po pierwsze, każde wyrażenie z wycinkiem fizycznie przechowa listę wyników w całości w pamięci. Iteratory tworzą z kolei po jednej wartości na raz, co pozwala nam zaoszczędzić sporo miejsca w przypadku dużych list wyników. Po drugie, wycinki tworzą nowe obiekty, więc tak naprawdę wcale nie wykonujemy iteracji po tym samym obiekcie w wielu miejscach. By zbliżyć się do klasy, musielibyśmy uzyskać jeden obiekt do przechodzenia, wykonując wycinek z wyprzedzeniem.

```
>>> S = 'abcdef'  
>>> S = S[::2]  
>>> S  
'ace'  
>>> for x in S:  
...     for y in S:  
...         print x + y,  
...  
aa ac ae ca cc ce ea ec ee
```

Ten sam obiekt, nowe iteratory

Teraz bardziej przypomina to nasze rozwiązanie oparte na klasie, jednak nadal musi ono przechować wynik wycinka w pamięci w całości (nie ma na razie żadnej formy generatora dla wycinków) i jest to tylko odpowiednik tego szczególnego przypadku pomijania co drugiego elementu.

Ponieważ iteratory mogą wykonać wszystko to, co klasy, są o wiele bardziej uniwersalne, niż mógłby to sugerować ten przykład. Bez względu na to, czy nasze aplikacje wymagają tego poziomu uniwersalności, iteratory definiowane przez użytkownika są narzędziem o dużych możliwościach — pozwalają na to, by dowolne obiekty wyglądały oraz zachowywały się jak inne spotkanie w książce sekwencje oraz obiekty, po których można iterować. Moglibyśmy użyć tej techniki na przykład na obiekcie bazy danych, aby wykonać iteracje w celu pobierania danych z bazy, z wieloma kurSORAMI w tym samym wyniku zapytania.

Test przynależności — `__contains__`, `__iter__` i `__getitem__`

Mechanizm iteratorów ma jeszcze więcej możliwości. Przeciążanie operatorów często odbywa się *warstwowo* — klasy oferują szczegółowe metody lub bardziej uogólnione alternatywy wykorzystywane w zależności od dostępności. Na przykład:

- Porównania w 2.6 wykorzystują metody `__lt__` do porównywania, czy inny obiekt jest mniejszy od instancji, ale bardziej ogólny interfejs dostarcza operator `__cmp__`. W Pythonie 3.0 wykorzystywane są tylko metody szczegółowe, `__cmp__` nie jest już wywoływana (o czym w dalszej części rozdziału).
- Testy logiczne wykorzystują metodę `__bool__` (zwracającą wynik `True/False`), a jeśli instancja jej nie posiada, wywoływana jest metoda `__len__` (wynik niezerowy jest traktowany jako `True`). Jak się dowiemy w dalszej części rozdziału, Python 2.6 działa tak samo, ale zamiast metody `__bool__` wykorzystuje metodę `__nonzero__`.

W przypadku iteratorów klasy implementują operator przynależności i wykorzystują metodę `__iter__` albo `__getitem__`. W celu zastosowania bardziej zaawansowanej logiki klasy mogą również implementować metodę `__contains__`: gdy ta metoda jest dostępna, jest preferowana i ma przewagę nad `__iter__`, która z kolei ma pierwszeństwo przed `__getitem__`. Metoda `__contains__` powinna definiować przynależność na zasadzie kluczy słownika (wykorzystując szybkie wyszukiwanie) oraz jako mechanizm wyszukiwania w sekwencjach.

Przeanalizujmy poniższą klasę implementującą wszystkie trzy wspomniane metody i testującą przynależność obiektów oraz różne konteksty iteracyjne. Metody wyświetlają komunikaty diagnostyczne przy każdym wywołaniu.

```
class Iters:
    def __init__(self, value):
        self.data = value
    def __getitem__(self, i):
        print('get[%s]: ' % i, end='')
        return self.data[i]
    def __iter__(self):
        print('iter=> ', end='')
        self.ix = 0
        return self
    def __next__(self):
        print('next:', end='')
        if self.ix == len(self.data): raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__(self, x):
        print('contains: ', end='')
        return x in self.data

X = Iters([1, 2, 3, 4, 5])
print(3 in X)
for i in X:
    print(i, end=' | ')

print()
print([i ** 2 for i in X])
print(list(map(bin, X)))

I = iter(X)

while True:
    try:
        print(next(I), end=' @ ')
    except StopIteration:
        break
```

Metoda zastępująca do użycia przez iterację
oraz do indeksowania i wycinania

Metoda preferowana w iteracji
Pozwala na użycie tylko jednego iteratora

Metoda preferowana w operacji 'in'

Utworzenie instancji
Przynależność
Pętle for

Inne konteksty iteracyjne

Ręczna iteracja (demonstracja mechanizmu stosowanego
w kontekstach iteracyjnych)

W przypadku wywołania tego modułu jako skryptu wygeneruje on wynik prezentowany na listingu poniżej. Metoda `__contains__` przechwytuje operację testu przynależności, metoda `__iter__` obsługuje konteksty iteracyjne, w których wywoływana jest metoda `__next__`, natomiast metoda `__getitem__` nie jest nigdy wywoływana.

```
contains: True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:[‘Ob1’, ‘Ob10’, ‘Ob11’, ‘Ob100’, ‘Ob101’]
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Sprawdźmy, co się stanie w przypadku, gdy zostanie zakomentowana metoda `__contains__` — test przynależności jest realizowany przez metodę `__iter__`.

```
iter=> next:next:next:True
iter=> next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=> next:next:next:next:next:[1, 4, 9, 16, 25]
iter=> next:next:next:next:next:[‘Ob1’, ‘Ob10’, ‘Ob11’, ‘Ob100’, ‘Ob101’]
iter=> next:1 @ next:2 @ next:3 @ next:4 @ next:5 @ next:
```

Na koniec przetestujmy wynik skryptu po ukryciu metod `__contains__` i `__iter__` — w takim przypadku wykorzystywana będzie metoda `__getitem__`, wywoływana z kolejnymi indeksami w kontekście testu przynależności oraz w iteracjach.

```
get[0]:get[1]:get[2]:True
get[0]:1 | get[1]:2 | get[2]:3 | get[3]:4 | get[4]:5 | get[5]:
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[1, 4, 9, 16, 25]
get[0]:get[1]:get[2]:get[3]:get[4]:get[5]:[‘Ob1’, ‘Ob10’, ‘Ob11’, ‘Ob100’, ‘Ob101’]
get[0]:1 @ get[1]:2 @ get[2]:3 @ get[3]:4 @ get[4]:5 @ get[5]:
```

Jak widać, metoda `__getitem__` jest jeszcze bardziej ogólna: oprócz iteracji potrafi obsłużyć indeksowanie oraz tworzenie wycinków. Wyrażenia wycinające wywołują metodę `__getitem__` z obiektem wycinka określającym zakresy. Mechanizm działa tak samo dla typów wbudowanych, jak i dla klas zdefiniowanych przez użytkownika, zatem tworzenie wycinków zadziała w naszej klasie bez dodatkowego kodu:

```
>>> X = Iter('spam')                      # Indeksowanie
>>> X[0]                                     # __getitem__(0)
get[0]:'s'

>>> 'spam'[1:]                                # Składnia wycinania
'pam'
>>> 'spam'[slice(1, None)]                   # Obiekt wycinka
'pam'

>>> X[1:]                                     # __getitem__(slice(..))
get[slice(1, None, None)]:'pam'
>>> X[:-1]
get[slice(None, -1, None)]:'spa'
```

W bardziej realistycznych sytuacjach związanych z iteratorami, które nie wykorzystują sekwencji, użycie metody `__iter__` może być łatwiejsze, ponieważ nie wymaga obsługi indeksu, natomiast metoda `__contains__` pozwala w niektórych przypadkach znacznie zoptymalizować testy przynależności.

Metody `__getattr__` oraz `__setattr__` przechwytują referencje do atrybutów

Metoda `__getattr__` przechwytuje atrybuty wywołane za pomocą składni kwalifikującej. Metoda ta jest wywoływana z nazwą atrybutu jako łańcuchem znaków, zawsze gdy próbujemy zapisać w składni kwalifikującej instancję z *niezdefiniowaną* (nieistniejącą) nazwą atrybutu. Nie jest wywoływana, kiedy Python może odnaleźć atrybut, wykorzystując do tego procedurę wyszukiwania w drzewie dziedziczenia. Ze względu na to zachowanie `__getattr__` przydaje się jako punkt zaczepienia dla odpowiadania na żądania atrybutów w sposób ogólny — jak w poniższym kodzie.

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 40
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
40
>>> X.name
...pominieto tekst bledu...
AttributeError: name
```

W powyższym kodzie klasa `empty` oraz jej instancja `X` nie mają prawdziwych własnych atrybutów, dlatego dostęp do `X.age` powoduje przekazanie do metody `__getattr__`. Do atrybutu `self` zostaje przypisana instancja (`X`), a do atrybutu `attrname` przypisuje się łańcuch znaków nazwy niezdefiniowanego atrybutu ("age"). Klasa sprawia, że `age` wygląda jak prawdziwy atrybut, zwracając prawdziwą wartość (40) jako wynik wyrażenia ze składnią kwalifikującą `X.age`. W rezultacie `age` staje się *dynamicznie obliczonym* atrybutem.

W przypadku atrybutów, z którymi klasa nie wie, jak sobie radzić, metoda `__getattr__` zgłasza wbudowany wyjątek `AttributeError`, który przekazuje Pythonowi, że są to faktycznie zmienne niezdefiniowane — próba uzyskania `X.name` powoduje wystąpienie błędu. Z metodą `__getattr__` spotkamy się ponownie, kiedy zobaczymy działanie delegacji oraz właściwości w kolejnych dwóch rozdziałach. Wyjątkom poświęcona jest część VII książki.

Podobna metoda przeciążania operatorów o nazwie `__setattr__` przechwytuje *wszystkie* przypisania atrybutów. Jeśli jest ona zdefiniowana, `self.atrybut = wartość` staje się `self.__setattr__('atrybut', wartość)`. Jest to nieco trudniejsze w użyciu, ponieważ przypisanie do dowolnych atrybutów `self` wewnętrz wywołania metody `__setattr__` ponownie wywołuje `__setattr__`, powodując nieskończoną pętlę rekurencji (i w końcu wyjątek przerwania stosu). Jeśli chcemy korzystać z tej metody, musimy pamiętać, że przypisuje ona dowolne atrybuty instancji, indeksując omówiony w kolejnym podrozdziale słownik atrybutów. Należy używać `self.__dict__['name'] = x` zamiast `self.name = x`.

```
>>> class accesscontrol:
...     def __setattr__(self, attr, value):
...         if attr == 'age':
...             self.__dict__[attr] = value
...         else:
...             raise AttributeError, attr + ' nie jest dozwolony'
...
```

```

>>> X = accesscontrol( )
>>> X.age = 40
>>> X.age
40
>>> X.name = 'amadeusz'
...tekst pominięto...
AttributeError: name nie jest dozwolony

```

Te dwie metody przeciążania operatorów pozwalają nam kontrolować lub specjalizować dostęp do atrybutów w obiektach. Zazwyczaj pełnią one bardzo wyspecjalizowane role; część z nich omówimy w dalszej części książki.

Inne narzędzia do zarządzania atrybutami

Warto zapamiętać, że w Pythonie istnieje kilka sposobów zarządzania dostępem do atrybutów:

- Metoda `__getattribute__` przechwytuje wszystkie próby dostępu do atrybutów, nie tylko niezdefiniowanych. W przypadku jej użycia należy wykazać więcej ostrożności niż w przypadku `__getattr__`, aby uniknąć zapętleń.
- Funkcja wbudowana `property` pozwala powiązać z określona nazwą metody realizujące odczyt i zapis atrybutu.
- *Dekryptory* udostępniają protokół wiążący metody klasy `__get__` i `__set__` z dostępem do specyficznych atrybutów klasy.

Wymienione metody należą do zaawansowanych technik programowania i nie wszyscy programiści Pythona korzystają z nich na co dzień. Z tego powodu ich omówienie odłożymy do rozdziału 31., a szczegółową analizę technik zarządzania atrybutami do rozdziału 37.

Emulowanie prywatności w atrybutach instancji

Poniższy kod uogólnia poprzedni przykład w celu zezwolenia każdej klasie podzielonej na posiadanie własnej listy zmiennych prywatnych, które nie mogą być przypisane do jej instancji.

```

class PrivateExc(Exception): pass          # Więcej o wyjątkach później

class Privacy:
    def __setattr__(self, attrname, value):  # Dla self.attrname = value
        if attrname in self.privates:
            raise PrivateExc(attrname, self)
        else:
            self.__dict__[attrname] = value      # Pętla Self.attrname = value!

class Test1(Privacy):
    privates = ['age']

class Test2(Privacy):
    privates = ['name', 'pay']
    def __init__(self):
        self.__dict__['name'] = 'Amadeusz'

x = Test1( )
y = Test2( )

x.name = 'Edward'

```

```

y.name = 'Ernest'                                # <== porażka
y.age = 30
x.age = 40                                       # <== porażka

```

Tak naprawdę jest to pierwsze podejście do implementacji *prywatności zmiennych* w Pythonie (czyli niepozwolenia na modyfikację nazw atrybutów poza klasą). Choć Python nie obsługuje deklaracji prywatności jako takich, takie techniki mogą emulować dużą część tej koncepcji. Jest to jednak rozwiązanie częściowe — by było bardziej efektywne, musi zostać rozszerzone w taki sposób, by zezwalać również klasom podrzędnym na ustawianie atrybutów prywatnych, a także na używanie metody `__getattribute__` oraz klasy pośredniczącej (inaczej proxy) sprawdzającej pobrania prywatnych atrybutów.

Bardziej kompletne rozwiązanie ochrony prywatności odłożymy do rozdziału 38., w którym w sposób ogólny użyjemy *dekoratorów klas* do przechwytywania i analizy atrybutów. Programiści Pythona potrafią pisać duże platformy oraz aplikacje zorientowane obiektywnie bez deklaracji prywatności — co jest ciekawym odkryciem dotyczącym ogólnej kontroli dostępu, wykraczającym poza zakres naszych aktualnych celów.

Przechwytywanie referencji do atrybutów oraz przypisań jest przydatną techniką. Obsługuje *delegację* — technikę projektowania pozwalającą obiektom kodu kontrolującego na opakowywanie osadzonych obiektów, dodawanie nowego zachowania i przekierowywanie innych operacji z powrotem do opakowanych obiektów (więcej informacji na temat delegacji oraz klas opakowujących znajdzie się w rozdziale 30.).

Metody `__repr__` oraz `__str__` zwracają reprezentacje łańcuchów znaków

Kolejny przykład wykorzystuje konstruktor `__init__`, omówioną przed chwilą metodę przeciążania operatorów `__add__` oraz definiuje również metodę `__repr__` zwracającą reprezentacje łańcucha znaków dla instancji. Formatowanie łańcuchów znaków wykorzystane zostanie do konwersji zarządzanego obiektu `self.data` na łańcuch znaków. Po zdefiniowaniu metoda `__repr__` (i jej krewniak — metoda `__str__`) wywoływana jest automatycznie, kiedy instancje klas są wyświetlane lub przekształcane na łańcuchy znaków. Metody te pozwalają na zdefiniowanie lepszego formatu wyświetlania dla obiektów w porównaniu z domyślnym wyświetlaniem instancji.

Domyślny sposób wyświetlania obiektów instancji nie jest ani praktyczny, ani ładny:

```

>>> class adder():
...     def __init__(self, value=0):
...         self.data = value           # Inicjalizacja zmiennej data
...     def __add__(self, other):
...         self.data += other          # Dodanie zmiennej other w miejscu
...
>>> x = adder()                      # Domyślne wyświetlanie
>>> print(x)
<__main__.adder object at 0x025D66B0>
>>> x
<__main__.adder object at 0x025D66B0>

```

Ale napisanie własnej metody reprezentacji łańcucha znaków lub jej odziedziczenie pozwala na dostosowanie wyświetlania do własnych potrzeb:

```

>>> class addrepr(addr):
...     def __repr__(self):
...         return 'addrepr(%s)' % self.data
...
>>> x = addrepr(2)                                # Odziedziczenie metod __init__ oraz __add__
>>> x + 1                                         # Dodanie reprezentacji łańcucha znaków
...                                                 # Konwersja na łańcuch znaków jako kod
...
>>> x
addrepr(3)                                         # Wykonuje metodę __init__
>>> print x                                       # Wykonuje metodę __add__
addrepr(3)                                         # Wykonuje metodę __repr__
...
>>> str(x), repr(x)                            # Wykonuje metodę __repr__
('addrepr(3)', 'addrepr(3)')

```

Po co nam dwie metody wyświetletania? Głównie w celu obsługi różnych sytuacji. A dokładniej:

- `__str__` próbują w pierwszym rzędzie wykorzystać operacje wyświetletania przyjazne dla użytkownika, takie jak instrukcja `print` czy wbudowana funkcja `str`. Metoda `__str__` powinna zwracać ciąg znaków przyjazny dla użytkownika.
- `__repr__` wykorzystywana jest we wszystkich innych przypadkach: do zwracania wartości w sesji interaktywnej, wyniku wywołania funkcji `repr`, jak również do wyświetlania w sytuacji, gdy obiekt nie implementuje metody `__str__`. Metoda `__repr__` powinna z reguły zwracać łańcuch znaków przypominający kod źródłowy, który można wykorzystać do odtworzenia obiektu oraz jako informację dla programistów, zawierającą dodatkowe informacje o obiekcie.

W skrócie: metoda `__repr__` jest wykorzystywana wszędzie, z wyjątkiem wywołań funkcji `print` i `str` w sytuacji, gdy zdefiniowana jest metoda `__str__`. Należy jednak zauważyć, że jeśli w klasie nieobecna jest metoda `__str__`, Python korzysta z metody `__repr__` (jednak odwrotna zależność już nie działa). W innych przypadkach, jak zwracanie wartości obiektu w konsoli interaktywnej, wykorzystywana jest wyłącznie metoda `__repr__`, a `__str__` w ogóle nie jest używana.

```

>>> class addstr(addr):
...     def __str__(self):                               # __str__, ale bez __repr__
...         return '[Wartość: %s]' % self.data # Konwersja na ładny łańcuch znaków
...
>>> x = addstr(3)
>>> x + 1                                         # Domyślnie metoda repr
<__main__.addstr instance at 0x00B35EFO>
>>> print x                                       # Wykonuje metodę __str__
[Wartość: 4]
>>> str(x), repr(x)
(['[Wartość: 4]', '<__main__.addstr instance at 0x00B35EFO>'])

```

Z tego powodu metoda `__repr__` może być najlepsza, jeśli chcemy mieć jeden sposób wyświetlania dla wszystkich kontekstów. Definiując jednak obie metody, możemy obsługiwać odmienne sposoby wyświetlania w różnych kontekstach — na przykład wyświetlanie przeznaczone dla użytkownika obsługiwane przez `__str__` i wyświetlanie przeznaczone dla programistów w czasie ich pracy, dostępne za pomocą `__repr__`.

```

>>> class addboth(addr):
...     def __str__(self):
...         return '[Wartość: %s]' % self.data # Łanícuch znaków przyjazny dla użytkownika
...     def __repr__(self):
...         return 'addboth(%s)' % self.data   # Łanícuch znaków jako kod
...
>>> x = addboth(4)

```

```
>>> x + 1
>>> x
addboth(5)                                     # Wykonuje metodę __repr__
>>> print x
[Wartość: 5]                                    # Wykonuje metodę __str__
>>> str(x), repr(x)
('['Wartość: 5]', 'addboth(5)')
```

W tym miejscu warto przekazać dwie użyteczne wskazówki. Po pierwsze, należy pamiętać, że zarówno `__str__` jak i `__repr__` muszą zwracaćłańcuchy znaków; inne typy nie są obsługiwane i spowodują wyświetlanie błędów, w razie potrzeby należy je przekonwertować. Po drugie, w zależności od logiki konwersji ciągów znaków kontenera metoda `__str__` jest używana wyłącznie w przypadku, gdy obiekt jest dostępny bezpośrednio. Jeśli jest zagnieżdżony w innym obiekcie, do jego wyświetlenia zostanie użyta metoda `__repr__`. Poniższy listing demonstruje obydwie te obserwacje.

```
>>> class Printer:  
...     def __init__(self, val):  
...         self.val = val  
...     def __str__(self):  
...         return str(self.val)  
...  
>>> objs = [Printer(2), Printer(3)]  
>>> for x in objs: print(x)  
...  
2  
3  
  
>>> print(objs)  
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...]  
>>> objs  
[<__main__.Printer object at 0x025D06F0>, <__main__.Printer object at ...more...]
```

Aby mieć pewność, że zdefiniowane wyświetlanie działa we wszystkich kontekstach niezależnie od kontenera, należy używać metody `__repr__`, nie `__str__`. Pierwsza z nich działa we wszystkich przypadkach, w których druga nie ma zastosowania.

```
>>> class Printer:  
...     def __init__(self, val):  
...         self.val = val  
...     def __repr__(self):  
...         return str(self.val)  
...  
>>> objs = [Printer(2), Printer(3)]  
>>> for x in objs: print(x)  
...  
2  
3  
  
>>> print(objs)  
[2, 3]  
>>> objs  
[2, 3]
```

`__repr__` wywoływanie przez `print`, jeśli nie ma `__str__`
`__repr__` wywoływanie w konsoli i przy zagnieżdżeniach

Nie ma `__str__`: wywołuje `__repr__`

Wywołuje `__repr__`, nie `__str__`

W praktyce metoda `__str__` (oraz jej krewniak, metoda `__repr__`) wydaje się drugą najczęściej wykorzystywaną metodą przeciążania operatorów stosowaną w skryptach napisanych w Pythonie, zaraz za `__init__`. Za każdym razem, gdy możemy wyświetlić obiekt i zobaczyć własny sposób wyświetlania, najprawdopodobniej w użyciu jest jedna z tych metod.

Metoda `_radd_` obsługuje dodawanie prawostronne i modyfikację w miejscu

Ściśle rzecz biorąc, metoda `_add_`, która pojawiła się w poprzednim przykładzie, nie obsługuje użycia obiektów instancji po prawej stronie operatora `+`. By zaimplementować takie wyrażenia i tym samym obsługiwać operatory *przemienne*, należy zapisać w kodzie klasy również metodę `_radd_`. Python wywołuje tę metodę tylko wtedy, gdy obiekt po prawej stronie operatora `+` jest naszą instancją klasy, jednak obiekt z lewej strony nie jest instancją naszej klasy. We wszystkich innych przypadkach dla obiektu po lewej stronie wywoływana jest zamiast tego metoda `_add_`.

```
>>> class Commuter:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         print('add', self.val, other)
...         return other + self.val
...     def __radd__(self, other):
...         print('radd', self.val, other)
...         return other + self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> x + 1                                     # __add__: instancja + nieinstancja
add 88 1
89
>>> 1 + y                                     # __radd__: nieinstancja + instancja
radd 99 1
100
>>> x + y                                     # __add__: instancja + instancja, wywołuje __radd__
add 88 <__main__.Commuter instance at 0x0086C3D8>
radd 99 88
187
```

Warto zwrócić uwagę na odwrócenie kolejności w metodzie `_radd_`. Bezpieczniej jest, gdy argument `self` znajduje się tak naprawdę po prawej stronie operatora `+`, natomiast `other` po lewej. Warto również zauważyc, że `x` oraz `y` są tutaj instancjami tej samej klasy. Kiedy w wyrażeniach mieszanych pojawiają się instancje różnych klas, Python preferuje klasę instancji znajdującej się po lewej stronie. Gdy dodajemy do siebie dwie instancje, Python wywołuje metodę `_add_`, która z kolei wywołuje `_radd_`, upraszczając lewy operand.

Sytuacja nieco się komplikuje w bardziej realistycznych przypadkach, gdy typ klasy powinien być przeniesiony na wynik. Do realizacji zadania należy wprowadzić sprawdzanie typów operandów, aby przekonać się, czy operację można wykonać oraz uniknąć zagnieźdżeń. Na przykład w poniższym listingu, gdyby pominąć sprawdzanie typów, moglibyśmy doprowadzić do sytuacji, w której instancja klasy `Commuter` otrzymałaby wartość `val` również klasy `Commuter`: wystarczyłoby dodać dwie instancje tej klasy, co wywołałoby metodę `_add_`, która wywołuje `_radd_`.

```
>>> class Commuter:                      # Przeniesienie klasy operandu na wynik
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):
...         if isinstance(other, Commuter): other = other.val
...         return Commuter(self.val + other)
```

```

...     def __radd__(self, other):
...         return Commuter(other + self.val)
...     def __str__(self):
...         return '<Commuter: %s>' % self.val
...
>>> x = Commuter(88)
>>> y = Commuter(99)
>>> print(x + 10)                                # Wynik jest instancją klasy Commuter
<Commuter: 98>
>>> print(10 + y)
<Commuter: 109>

>>> z = x + y                                    # Niezagnieżdżone: nie wywołuje __radd__ rekurencyjnie
>>> print(z)
<Commuter: 187>
>>> print(z + 10)
<Commuter: 197>
>>> print(z + z)
<Commuter: 374>

```

Dodawanie w miejscu

Aby obiekt obsłużył operację dodawania w miejscu `+=`, należy zaimplementować metodę `__iadd__` lub `__add__`. Ta druga jest stosowana w przypadku, gdy klasa nie obsługuje pierwnej. Klasa `Commuter` omawiana w poprzednim punkcie miała zaimplementowaną tę metodę właśnie z myślą o operacji `+=`, ale `__iadd__` oferuje wydajniejsze modyfikacje w miejscu:

```

>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __iadd__(self, other):           # __iadd__ wywołuje: x += y
...         self.val += other               # Z reguły zwraca self
...         return self
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7
>>> class Number:
...     def __init__(self, val):
...         self.val = val
...     def __add__(self, other):          # __add__: x = (x + y)
...         return Number(self.val + other) # Przeniesienie typu na wynik
...
>>> x = Number(5)
>>> x += 1
>>> x += 1
>>> x.val
7

```

Każdy operator dwuelementowy wykorzystuje podobne metody do obsługi operacji prawostronnych oraz operacji w miejscu (na przykład `__mul__`, `__rmul__` i `__imul__`). Metody prawostronne stanowią zaawansowane zagadnienie i są dość rzadko stosowane, ponadto są konieczne wyłącznie w przypadku, gdy operatory muszą być naprzemienne. Zapewne potrzebowalibyśmy takich metod na przykład w klasie `Wektor`, ale wydają się one zbędne na przykład w klasie `Pracownik` lub `Przycisk`.

Metoda `__call__` przechwytuje wywołania

Metoda `__call__` wywoływana jest, kiedy wywołana zostaje nasza instancja. Nie jest to jednak definicja cykliczna — po zdefiniowaniu Python wykonuje metodę `__call__` dla wyrażeń wywołania funkcji stosowanych do naszych instancji, przekazując otrzymane argumenty pozycyjne i argumenty ze słowami kluczowymi:

```
>>> class Callee:  
...     def __call__(self, *pargs, **kargs):  
...         print('Wywołanie:', pargs, kargs)      # Przechwytyuje wywołania instancji  
...         # Akceptuje dowolne argumenty  
...  
>>> C = Callee()  
>>> C(1, 2, 3)                                # C jest obiektem wywoływanym  
Wywołanie: (1, 2, 3) {}  
>>> C(1, 2, 3, x=4, y=5)  
Wywołanie: (1, 2, 3) {'y': 5, 'x': 4}
```

Bardziej formalnie, wszystkie tryby przesyłania argumentów, które analizowaliśmy w rozdziale 18., są obsługiwane przez metodę `__call__`: wszystko, co jest przesyłane do instancji, zostaje przekazane do tej metody, wraz z argumentemem instancji. Poniższy listing prezentuje przykłady definicji metody `__call__`.

```
class C:  
    def __call__(self, a, b, c=5, d=6): ...      # Argumenty zwykłe i domyślne  
  
class C:  
    def __call__(self, *pargs, **kargs): ...       # Dowolne argumenty  
  
class C:  
    def __call__(self, *pargs, d=6, **kargs): ...   # Argumenty mogące być tylko słowami kluczowymi w 3.0
```

Powyższe definicje pozwalają wykonać następujące wywołania:

```
X = C()  
X(1, 2)                                         # Pominiecie argumentów domyślnych  
X(1, 2, 3, 4)                                    # Argumenty pozycyjne  
X(a=1, b=2, d=4)                                  # Argumenty ze słowami kluczowymi  
X(*[1, 2], **dict(c=3, d=4))                   # Rozpakowanie dowolnych argumentów  
X(1, *(2,), c=3, **dict(d=4))                  # Tryb mieszany
```

W efekcie klasy i instancje posiadające zdefiniowaną metodę `__call__` obsługują tę samą składnię i semantykę argumentów co zwykłe funkcje i metody.

Ten sposób przechwytywania wyrażeń wywołania funkcji pozwala instancjom klas emulować wygląd i zachowanie takich obiektów jak funkcje, ale z dodatkowymi możliwościami, jak zachowanie stanu między wywołaniami (podobne możliwości mieliśmy okazję zaobserwować przy okazji omawiania zakresów w rozdziale 17., ale w niniejszym rozdziale Czytelnik powinien lepiej rozumieć działanie mechanizmu przeciążania operatorów).

```
>>> class Prod:  
...     def __init__(self, value):           # Przyjmuje jeden argument  
...         self.value = value  
...     def __call__(self, other):  
...         return self.value * other  
...  
>>> x = Prod(2)                           # "Zapamiętuje" wartość 2  
>>> x(3)                                 # 3 (przekazane) * 2 (zapamiętane)  
6  
>>> x(4)  
8
```

W tym przykładzie `__call__` może się wydawać zbędna. Podobne narzędzie może udostępniać prosta metoda.

```
>>> class Prod:
...     def __init__(self, value):
...         self.value = value
...     def comp(self, other):
...         return self.value * other
...
>>> x = Prod(3)
>>> x.comp(3)
9
>>> x.comp(4)
12
```

Metoda `__call__` staje się jednak bardziej przydatna przy interfejsach API oczekujących funkcji. Pozwala nam na tworzenie w kodzie obiektów zgodnych z oczekiwanyim interfejsem wywołania funkcji i przechowujących również informacje o stanie. Tak naprawdę jest to chyba trzecia najczęściej wykorzystywana metoda przeciążania operatorów, po konstruktorze `__init__` oraz alternatywnych metodach formatu wyświetlanego `__str__` oraz `__repr__`.

Interfejsy funkcji i kod oparty na wywołaniach zwrotnych

Jako przykład weźmy wspomniany wcześniej zbiór narzędzi graficznego interfejsu użytkownika (noszący w Pythonie 2.6 nazwę Tkinter), który pozwala na rejestrowanie funkcji jako programów obsługi zdarzeń (ang. *event handler*, inaczej wywołań zwrotnych, ang. *callback*). Kiedy następuje zdarzenie, Tkinter wywołuje zarejestrowane obiekty. Jeśli chcemy, by program obsługi zdarzeń zachowywał stan pomiędzy zdarzeniami, możemy albo zarejestrować metodę klasy bound, albo instancję zgodną z oczekiwanyim interfejsem dzięki `__call__`. W kodzie z tego podrozdziału zarówno `x.comp` z drugiego przykładu, jak i `x` z pierwszego mogą w ten sposób zostać uznane za obiekty podobne do funkcji.

Więcej informacji na temat metod z wiązaniem znajdzie się w kolejnym rozdziale, natomiast poniżej znajduje się hipotetyczny przykład zastosowania metody `__call__` w dziedzinie graficznych interfejsów użytkownika. Poniższa klasa definiuje obiekt obsługujący interfejs wywołania funkcji, ale również posiada informacje o stanie zapamiętujące kolor, na jaki po naciśnięciu powinien zmieniać się przycisk.

```
class Callback:
    def __init__(self, color):                      # Funkcja i informacje o stanie
        self.color = color
    def __call__(self):                            # Obsługa wywołań bez argumentów
        print 'łącz', self.color
```

W kontekście graficznego interfejsu użytkownika możemy zarejestrować instancje tej klasy jako programy obsługi zdarzeń przycisków, nawet jeśli GUI oczekuje, że programy obsługi zdarzeń będzie można wywołać jak proste funkcje bez argumentów.

```
cb1 = Callback('niebieski')                      # 'Pamięta' niebieski
cb2 = Callback('zielony')                         # 'Pamięta' zielony

B1 = Button(command=cb1)                          # Rejestrowanie programów obsługi
B2 = Button(command=cb2)                          # Rejestrowanie programów obsługi
```

Po późniejszym naciśnięciu przycisku instancja obiektu wywoływana jest jako prosta funkcja — dokładnie tak, jak w poniższych wywołaniach. Ponieważ zachowuje ona stan jako atrybuty instancji, pamięta również, co ma zrobić.

```
cb1()  
cb2()  
# Przy zdarzeniu: wyświetla 'niebieski'  
# Wyświetla 'zielony'
```

Tak naprawdę jest to chyba najlepszy sposób zachowania informacji o stanie w Pythonie — lepszy od technik opisanych wcześniej przy okazji omawiania funkcji (zmienne globalne, referencje do zakresu funkcji zawierającej oraz domyślne argumenty zmienne). Dzięki programowaniu zorientowanemu obiektowo zapamiętany stan jest jawnym dzięki użyciu przypisania atrybutów.

Zanim przejdziemy dalej, powiem jeszcze, że istnieją dwa kolejne sposoby przypinania w ten sposób informacji do funkcji zwrotnych. Jedna możliwość to wykorzystanie argumentów domyślnych w funkcjach lambda.

```
cb3 = (lambda color='czerwony': 'włącz ' + color) # Lub: wartości domyślne  
print cb3()
```

Drugą opcję jest wykorzystanie metod klasy z *wiązaniem* — rodzaju obiektu, który pamięta instancję `self` oraz funkcję z referencji w taki sposób, że może później być wywołany jak prosta funkcja bez instancji.

```
class Callback:  
    def __init__(self, color):  
        self.color = color  
    def changeColor(self):  
        print('włącz', self.color)  
  
    cb1 = Callback('niebieski')  
    cb2 = Callback('zielony')  
  
    B1 = Button(command=cb1.changeColor) # Referencja, ale bez wywołania  
    B2 = Button(command=cb2.changeColor) # Pamięta funkcję i self
```

Kiedy przycisk ten zostanie później naciśnięty, zachowuje się, jakby odpowiedzialny za to był graficzny interfejs użytkownika, wywołując metodę `changeColor` w celu przetworzenia informacji o stanie obiektu.

```
object = Callback('niebieski')  
cb = object.changeColor  
cb() # Zarejestrowany program obsługi zdarzeń  
      # Po wystąpieniu zdarzenia wyświetla 'niebieski'
```

W powyższym przykładzie nie musimy używać wyrażenia `lambda`, ponieważ sama referencja do metody pozwala opóźnić jej wywołanie (więcej informacji o metodach związanych przekaże w rozdziale 30.). Ta technika jest prostsza, jednak mniej uniwersalna od przeciążania wywołań za pomocą metody `__call__`. Przypominam, że informacje na temat metod z wiązaniem znajdują się w następnym rozdziale.

Kolejny przykład zastosowania metody `__call__` zobaczymy w rozdziale 31., w którym wykorzystamy ją do zaimplementowania czegoś, co znane jest pod nazwą *dekoratora funkcji* (ang. *function decorator*) — obiektu wywoływalnego, który dodaje warstwę logiki na wierzchu funkcji osadzonej. Ponieważ metoda `__call__` pozwala na dołączanie informacji o stanie do obiektu wywoływalnego, jest to naturalna technika implementacyjna dla funkcji, która musi pamiętać i wywołać inną funkcję.

Porównania — `_lt_`, `_gt_` i inne

Jak sugeruje tabela 29.1, klasy mogą definiować metody przechwytyjące sześć operacji porównania: `<`, `>`, `<=`, `>=` i `==` i `!=`. Metody przeciążające te operatory są dość proste w użyciu, ale należy pamiętać o pewnych podstawowych zasadach:

- W przeciwnieństwie do powiązań metod typu `_add_` z ich metodami prawostronnymi typu `_radd_` w przypadku porównań nie ma takich odpowiedników. Natomiast są dostępne odwrócenia operacji: odwróceniem operacji `_lt_` jest `_gt_` i vice versa.
- Nie istnieją jawne powiązania pomiędzy operatorami porównania. Prawda zwrócona przez operator `==` nie oznacza, że operator `!=` na tych samych operandach zwróci fałsz, dlatego metody `_eq_` i `_ne_` powinny być zdefiniowane w sposób niezależny, aby zapewnić prawidłowe działanie operatorów.
- W Pythonie 2.6 metoda `_cmp_` jest wykorzystywana przez wszystkie operacje porównania, jeśli nie są zdefiniowane odpowiednie metody dedykowane. Metoda ta zwraca liczbę mniejszą od zera, równą zeru lub większą w zależności od tego, czy pierwszy operand (`self`) jest odpowiednio mniejszy, równy drugiemu lub od niego większy. Ta metoda często używa funkcji wbudowanej `cmp(x, y)`. Metoda `_cmp_` oraz funkcja wbudowana `cmp` zostały usunięte w Pythonie 3.0, zatem jesteśmy zmuszeni do wykorzystania dedykowanych metod.

Nie mamy miejsca na dokładną analizę metod obsługujących operatory porównania, ale w ramach szybkiego wprowadzenia weźmy pod uwagę następujący kod klasy:

```
class C:  
    data = 'spam'  
    def __gt__(self, other):  
        return self.data > other  
    # 3.0 i 2.6  
    def __lt__(self, other):  
        return self.data < other  
  
X = C()  
print(X > 'ham')  
# True (wywołuje __gt__)  
print(X < 'ham')  
# False (wywołuje __lt__)
```

W przypadku uruchomienia w Pythonie 3.0 lub 2.6 skrypt ten wypisze dla każdego porównania wyniki podane w komentarzach — dzięki temu, że obsługa operatorów przechwytyują metody klasy `C`.

Metoda `_cmp_` w 2.6 (usunięta w 3.0)

W Pythonie 2.6 dostępna jest metoda `_cmp_` wykorzystywana jako mechanizm zastępczy w przypadku, gdy nie jest zdefiniowana metoda dedykowana do obsługi danego operatora. Metoda ta zwraca wyniki całkowite sygnalizujące wynik porównania operandów. Poniższy kod zadziała w 2.6, ale nie zadziała w 3.0, ponieważ w tej wersji metoda `_cmp_` nie jest już wywoływana.

```
class C:  
    data = 'spam'  
    def __cmp__(self, other):  
        return cmp(self.data, other)  
    # Tylko 2.6  
    # __cmp__ nieużywane w 3.0  
    # cmp niezdefiniowane w 3.0  
  
X = C()  
print(X > 'ham')  
# True (wywołuje __cmp__)  
print(X < 'ham')  
# False (wywołuje __cmp__)
```

Powodem niedziałania powyższego kodu nie jest to, że funkcja `cmp` nie jest dostępna, ale to, że metoda `__cmp__` w ogóle nie zostaje wywołana. Zmodyfikujmy metodę `__cmp__`, aby uniknąć wywołania funkcji `cmp`. Kod nadal będzie działał w 2.6 i nadal nie będzie działał w 3.0.

```
class C:  
    data = 'spam'  
    def __cmp__(self, other):  
        return (self.data > other) - (self.data < other)
```



Ktoś mógłby zapytać, po co prezentuję metodę, której nie ma już w Pythonie 3.0. Z pewnością często wygodna byłaby możliwość kompletnego wymazania historii, ale niniejsza książka ma za zadanie omawiać wersje 2.6 i 3.0. Operator `__cmp__` jest obsługiwany w 2.6, przez co osoby programujące w tej wersji mogą napotkać jego użycie w kodzie, który muszą obsługiwać, zatem jestem zobowiązany wspomnieć o tej zasłości historycznej. Co więcej, metoda `__cmp__` została porzucona dość nagle — w porównaniu z inną porzuconą metodą `__getslice__` omawianą wcześniej, ale była stosunkowo częściej stosowana w kodzie, przez co łatwiej się na nią natknąć. Jeśli ktoś chce zachować zgodność pisanej kodu z Pythonem 3.0, powinien zrezygnować ze stosowania metody `__cmp__` i przepisać swój kod tak, aby używać dedykowanych metod obsługujących operacje porównania.

Testy logiczne — `__bool__` i `__len__`

Jak wspomniałem wcześniej, klasy mogą również definiować metody zwracające wartość logiczną (boolowską) swoich instancji. W takich operacjach Python wywoła metodę `__bool__`, aby uzyskać bezpośrednią interpretację instancji, a jeśli metoda ta nie jest zdefiniowana, wywoła metodę `__len__`, aby określić reprezentację obiektu na podstawie tego, czy zawiera elementy (`True`), czy też jest pusty (`False`). Metoda `__bool__` do wyznaczania wartości logicznej obiektu najczęściej wykorzystuje jego stan lub inne informacje zapisane w instancji.

```
>>> class Truth:  
...     def __bool__(self): return True  
...  
>>> X = Truth()  
>>> if X: print('tak!')  
...  
tak!  
  
>>> class Truth:  
...     def __bool__(self): return False  
...  
>>> X = Truth()  
>>> bool(X)  
False
```

Jeśli ta metoda nie jest zdefiniowana, Python próbuje określić długość obiektu: obiekty niepuste są w operacjach logicznych traktowane jako odpowiedniki wartości `True` (długość większa od zera implikuje wartość `True`, zero oznacza `False`).

```
>>> class Truth:  
...     def __len__(self): return 0  
...  
>>> X = Truth()  
>>> if not X: print('nie!')  
...  
nie!
```

Jeśli w klasie są zdefiniowane obie te metody, w kontekście logicznym Python wybierze metodę `__bool__`, ponieważ jest specjalizowana do tego typu operacji.

```
>>> class Truth:  
...     def __bool__(self): return True  
...     def __len__(self): return 0  
...  
>>> X = Truth()  
>>> if X: print('tak!')  
...  
tak!
```

Jeśli w obiekcie nie jest zdefiniowana żadna z tych metod, obiekt zostanie zinterpretowany jako wartość `True` (co może mieć poważne skutki uboczne dla Czytelników o zainteresowaniach metafizycznych).

```
>>> class Truth:  
...     pass  
...  
>>> X = Truth()  
>>> bool(X)  
True
```

Skoro już udało się nam zahaczyć o obszary metafizyczne, nadszedł czas, aby przejść do ostatniego zagadnienia związanego z przeciążaniem operatorów — śmierci obiektów.

Destrukcja obiektu — `__del__`

Jak widzieliśmy, konstruktor `__init__` jest wywoływany przy każdym tworzeniu instancji. W Pythonie istnieje też druga strona: metoda destruktora `__del__` wywoływana automatycznie w momencie, gdy interpreter zwalnia pamięć zajmowaną przez instancję (na przykład w procesie „odśmiecania”, ang. *garbage collection*).

```
>>> class Life:  
...     def __init__(self, name='nieznajomy'):  
...         print('Witaj', name)  
...         self.name = name  
...     def __del__(self):  
...         print('Żegnaj', self.name)  
...  
>>> brian = Life('Brian')  
Witaj Brian  
>>> brian = 'loretta'  
Żegnaj Brian
```

Gdy zmiennej `brian` przypisujemy ciąg znaków, tracimy ostatnią referencję do utworzonej wcześniej instancji klasy `Life`, co powoduje wywołanie destruktora. Tego typu zachowanie może być pomocne przy tworzeniu procedur porządkujących (na przykład w celu zwolnienia połączeń do serwera). Jednak z kilku powodów destruktory nie są tak powszechnie stosowane w Pythonie jak w niektórych innych językach zorientowanych obiektowo.

Pierwszym powodem jest to, że Python automatycznie zwalnia pamięć zajmowaną przez nieużywane obiekty, a więc destruktory nie są potrzebne do zarządzania pamięcią.¹ Drugi powód

¹ W oficjalnej implementacji Pythona w C nie ma konieczności zamknięcia obiektów plikowych otwartych w instancji, ponieważ takie pliki zostaną zamknięte automatycznie w ramach procedury zwalniania pamięci po obiekcie. Jednak, jak wspomniałem w rozdziale 9., warto nabrać nawyku zamknięcia plików, ponieważ automatyczne zamknięcie plików jest cechą *implementacji*, nie cechą języka (w innej implementacji, na przykład Jython, obiekty plikowe mogą zachowywać się inaczej).

Wartości boolowskie w Pythonie 2.6

Użytkownicy Pythona 2.6 zamiast metody `__bool__` w tekście punktu „Testy logiczne — `__bool__` i `__len__`” muszą wykorzystać metodę `__nonzero__`. W Pythonie 3.0 nastąpiła zmiana nazwy metody `__nonzero__` na `__bool__`, ale pomijając ten zabieg nazewnicy, operacje logiczne działają w 2.6 tak samo jak w 3.0 (w obydwu wersjach pod nieobecność odpowiedniej metody używana jest metoda `__len__`).

Uruchamiając w 2.6 pierwszy listing z poprzedniego punktu, otrzymamy takie same wyniki, ale to tylko przypadek, ponieważ metoda `__bool__` nie jest rozpoznawana, a pod nieobecność metod specjalnych wszystkie obiekty zwracają wartość `True`!

Aby zaobserwować różnice między wersjami, zmuśmy kod do zwracania wartości `False`.

```
C:\misc> c:\python30\python
>>> class C:
...     def __bool__(self):
...         print('bool')
...         return False
...
...
>>> X = C()
>>> bool(X)
bool
False
>>> if X: print(99)
...
99
```

W wersji 3.0 kod zadziała zgodnie z oczekiwaniami. Jednak w 2.6 metoda `__bool__` jest ignorowana, co powoduje, że obiekty są interpretowane jako `True`.

```
C:\misc> c:\python26\python
>>> class C:
...     def __bool__(self):
...         print('bool')
...         return False
...
...
>>> X = C()
>>> bool(X)
True
>>> if X: print(99)
...
99
```

W 2.6 powinniśmy zastosować metodę `__nonzero__` (albo zwrócić zero z metody `__len__`, sygnalizując wartość `False`).

```
C:\misc> c:\python26\python
>>> class C:
...     def __nonzero__(self):
...         print('nie zero')
...         return False
...
...
>>> X = C()
>>> bool(X)
nie zero
False
>>> if X: print(99)
...
nie zero
```

Należy pamiętać, że metoda `__nonzero__` zadziała tylko w 2.6; jeśli zostanie użyta w 3.0, będzie ignorowana poprzez konteksty logiczne, tak samo jak metoda `__bool__` jest ignorowana w 2.6.

jest taki, że nie zawsze łatwo przewidzieć, kiedy instancja może zostać zwolniona, dlatego zwolnienie zajmowanych przez nią zasobów powinno odbywać się w specjalnej metodzie wywoływanej automatycznie (lub w bloku `try/finally` opisywanym w dalszej części książki). W niektórych przypadkach do obiektu mogą istnieć dodatkowe referencje, na przykład w strukturach systemowych, które uniemożliwią wywołanie jego destruktora.



W rzeczywistości użycie metody `__del__` może sprawiać problemy z jeszcze kilku, bardziej subtelnych powodów. Wyjątki wywoływanego w tej metodzie po prostu wyświetla komunikaty na standardowym wyjściu błędów programu `sys.stderr`, ponieważ kontekst wywołania destruktora trudno określić, gdyż ta metoda nie jest wywoływana przez sam program, a przez wewnętrzne mechanizmy interpretera (mechanizm odśmieciania). Ponadto obiekty mogą powodować zapętlenia referencji (referencje cykliczne), które zupełnie uniemożliwią wywołanie odśmieciania na zajmowanej przez nie pamięci. Interpreter posiada mechanizm wykrywający tego typu cykliczne referencje, który domyślnie jest aktywny i potrafi wykryć i usunąć z pamięci takie referencje, ale jest stosowany wyłącznie w przypadku, gdy obiekt posiada metodę `__del__`. Szczegóły są dość zagmatwane, nie będę ich zatem tu omawiał. Zainteresowanych odsyłam do standardowego podręcznika Pythona: warto zajrzeć do dokumentacji metody `__del__` oraz modułu odśmiecacza `gc`.

Podsumowanie rozdziału

To już koniec przykładów przeciążania metod, na więcej nie mamy miejsca. Wiele innych, nieomówionych szczegółowo metod przeciążania operatorów działa w podobny sposób do tych, którym poświęciliśmy więcej uwagi. Wszystkie są natomiast jedynie punktami zaczepienia operatorów do instancji klas. Niektóre metody przeciążania operatorów wymagają zastosowania specjalnych list argumentów lub zwracają nietypowe wartości. Kilka przykładów będziemy mieli okazję poznać w dalszych częściach książki:

- Rozdział 33. wykorzystuje metody `__enter__` i `__exit__` służące do obsługi menedżera kontekstu.
- Rozdział 37. wykorzystuje metody `__get__` i `__set__` używane w deskryptorach klas do przechwycenia metod `fetch` i `set`.
- Rozdział 39. wykorzystuje metodę `__new__` służącą do użycia w kontekście metaklas metody tworzenia obiektów.

Dodatkowo niektóre metody omawiane w niniejszym rozdziale, jak `__call__` i `__str__`, będą dość powszechnie wykorzystywane w wielu przykładach tej książki. Szczegółowe omówienie wszystkich dostępnych metod można znaleźć w innych źródłach. Odsyłam Czytelnika do standardowego podręcznika Pythona oraz innych książek omawiających szczegółowo pozostałe metody przeciążania operatorów.

W następnym rozdziale odkładamy omawianie wewnętrznych mechanizmów klas, aby przejść do najczęściej stosowanych wzorców projektowych służących do optymalizacji ponownego użycia kodu. Zanim jednak Czytelnik przejdzie dalej, sugeruję zatrzymać się na chwilę nad quizem do zagadnień z niniejszego rozdziału.

Sprawdź swoją wiedzę — quiz

1. Jakie dwie metody specjalne można wykorzystać do zaimplementowania iteracji w tworzonych klasach?
2. Jakie dwie metody specjalne obsługują wyświetlanie obiektów i w jakich kontekstach są stosowane?
3. W jaki sposób samodzielnie obsłużyć w klasie operację wycinania fragmentu sekwencji?
4. W jaki sposób przechwycić w klasie operację dodawania w miejscu?
5. W jakich przypadkach należy wykorzystywać mechanizm przeciążania operatorów?

Sprawdź swoją wiedzę — odpowiedzi

1. Klasy mogą implementować iterację przez zdefiniowanie (lub odziedziczenie) metody `__getitem__` lub `__iter__`. We wszystkich kontekstach iteracyjnych Python próbuje użyć metody `__iter__` (która powinna zwrócić obiekt obsługujący protokół iteracyjny udostępniający metodę `__next__`). Jeśli metoda `__iter__` nie zostanie znaleziona, Python przełącza się w tryb użycia metody indeksującej `__getitem__` (która jest wywoływana sekwencyjnie z kolejnymi indeksami elementów).
2. Wyświetlanie obiektów jest obsługiwane przez metody `__str__` i `__repr__`. Pierwsza z nich jest wywoływana w przypadku użycia funkcji wbudowanych `print` lub `str`, druga jest wywoywana dla funkcji `print` i `str` w przypadku, gdy obiekt nie posiada metody `__str__`, a dodatkowo zawsze w przypadku użycia funkcji wbudowanej `repr`, w konsoli interaktywnej do zwracania wartości obiektu na konsolę (`echo`) oraz w sytuacjach zagnieżdżenia obiektów. Innymi słowy, `__repr__` jest używana zawsze, z wyjątkiem użycia `print` i `str`, gdy obiekt posiada metodę `__str__`. Zwyczajowo funkcja `__str__` powinna zwracać informacje o obiekcie w formacie czytelnym dla użytkownika, natomiast `__repr__` wyświetla szczegóły obiektu w formacie przypominającym kod źródłowy.
3. Tworzenie wycinków odbywa się przez metodę indeksującą `__getitem__`, wywołaną z obiektem wycinka zamiast zwykłego indeksu. W Pythonie 2.6 dodatkowo dostępna jest metoda `__getslice__` (wycofana w 3.0).
4. Operacja dodawania w miejscu próbuje wywołać metodę `__iadd__`, a jeśli jej nie znajdzie, szuka `__add__`. Taka sama zasada jest stosowana dla wszystkich operacji dwuoperatorowych. Dodatkowo dostępna jest metoda `__radd__`, służąca do dodawania argumentów po prawej stronie.
5. Przeciążanie operatorów należy stosować w sytuacjach, gdy klasa powinna być używana w kontekstach operacji wbudowanych. Na przykład kolekcje mogą emulować interfejsy sekwencji lub mapowań. Nie powinno się implementować operatorów, które nie są logiczne w kontekście użycia obiektu; w takich przypadkach zaleca się stosowanie zwykłych metod.

Projektowanie z użyciem klas

Dotychczas w tej części książki koncentrowaliśmy się na wykorzystywaniu narzędzia programowania zorientowanego obiektowego Pythona, czyli klasy. Programowanie zorientowane obiektowo to jednak również *kwestie związane z projektowaniem*, czyli to, jak wykorzystać klasy do modelowania przydatnych obiektów. W niniejszym rozdziale zajmiemy się kilkoma zagadnieniami kluczowymi dla programowania zorientowanego obiektowo i zaprezentujemy dodatkowe przykłady, bardziej realistyczne od pokazanych wcześniej.

Omówimy wiele powszechnie stosowanych w Pythonie wzorców projektowych, jak dziedziczenie, delegacja, kompozycja czy fabryki. Przeanalizujemy też koncepcje klas z punktu widzenia projektowania, omawiając atrybuty pseudoprivatne, wielokrotne dziedziczenie i metody związane. Wiele z zawartych tu koncepcji wymaga nieco szerszego objaśnienia niż to, które jestem w stanie zamieścić w niniejszej książce. Jeśli kogoś one zaciekały, sugeruję znalezienie jakiegoś tekstu poświęconego projektowaniu w programowaniu zorientowanym obiektowo lub wzorcom projektowym.

Python a programowanie zorientowane obiektowo

Implementację programowania zorientowanego obiektowo w Pythonie można sprowadzić do trzech kwestii.

Dziedziczenie

Dziedziczenie jest w Pythonie oparte na wyszukiwaniu atrybutów (w wyrażeniach `X.nazwa`).

Polimorfizm

W wyrażeniu `X.metoda` znaczenie atrybutu `metoda` uzależnione jest od typu (klasy) obiektu `X`.

Hermetyzacja (enkapsulacja)

Metody oraz operatory implementują zachowanie. Ukrywanie danych jest domyślną konwencją.

Na tym etapie lektury każdy powinien wiedzieć już, czym jest w Pythonie dziedziczenie. Kilka razy wspominaliśmy również o polimorfizmie Pythona — wypływa on z braku deklaracji typów w tym języku. Ponieważ atrybuty są zawsze interpretowane w czasie wykonywania, obiekty implementujące te same interfejsy są wymienne. Klient nie musi wiedzieć, jakie rodzaje obiektów implementują wywoływane metody.

Hermetyzacja oznacza w Pythonie pakowanie, czyli ukrywanie szczegółów implementacyjnych za interfejsem obiektu. Nie chodzi tu o wymuszenie prywatności danych, choć i taką własność daje się osiągnąć w kodzie, o czym przekonamy się w rozdziale 38. Hermetyzacja pozwala na modyfikację implementacji interfejsu obiektu bez ingerowania w użytkowników tego obiektu.

Przeciążanie za pomocą sygnatur wywołań (lub bez nich)

Niektóre języki zorientowane obiektowo definiują polimorfizm jako przeciążanie funkcji w oparciu o sygnatury typów argumentów. Ponieważ jednak w Pythonie nie istnieją deklaracje typów, koncepcja ta nie ma w rzeczywistości zastosowania. Polimorfizm w Pythonie oparty jest na *interfejsach* obiektów, nie na typach.

Mogemy spróbować przeciążać metody za pomocą ich list argumentów, jak w poniższym kodzie.

```
class C:  
    def meth(self, x):  
        ...  
    def meth(self, x, y, z):  
        ...
```

Kod ten będzie działał, jednak ponieważ instrukcja `def` po prostu przypisuje obiekt do nazwy w zakresie klasy, ostatnia definicja funkcji metody będzie tą, która zostanie zachowana (to dokładnie tak samo jak wywołanie najpierw `X = 1`, a później `X = 2` — zmienna `X` będzie miała wartość 2).

Wybór oparty na typie można zawsze zapisać w kodzie z wykorzystaniem technik sprawdzania typów omówionych w rozdziałach 4. oraz 9. lub narzędzi list argumentów z rozdziału 18.

```
class C:  
    def meth(self, *args):  
        if len(args) == 1:  
            ...  
        elif type(arg[0]) == int:  
            ...
```

Normalnie nie powinniśmy tego jednak robić. Zgodnie z informacjami z rozdziału 16. kod powinniśmy pisać w taki sposób, by oczekwał on określonego interfejsu obiektu, a nie specyficznego typu danych. W ten sposób będzie przydatny dla szerszej kategorii typów oraz zastosowań, zarówno teraz, jak i w przyszłości.

```
class C:  
    def meth(self, x):  
        x.operation() # Zakładamy, że x robi coś właściwego
```

Powszechnie uważa się również, że lepiej jest używać osobnych nazw metod dla różnych operacji, zamiast polegać na sygnaturach wywołania (bez względu na to, w jakim języku programowania tworzymy kod).

Koncepcja obiektowa w Pythonie jest łatwa do zrozumienia, jednak cała sztuka programowania obiektowego polega na sposobie łączenia klas ze sobą w celu realizacji zadań pisanej programu. Kolejny podrozdział rozpocznie analizę dostępnych w Pythonie koncepcji programowania obiektowego.

Programowanie zorientowane obiektowo i dziedziczenie — związek „jest”

Omówiliśmy już dość szczegółowo mechanizmy dziedziczenia, jednak chciałbym zaprezentować przykład tego, w jaki sposób można wykorzystać dziedziczenie do modelowania relacji ze świata rzeczywistego. Z punktu widzenia programisty dziedziczenie uruchamiane jest przez kwalifikację atrybutów powodującą wyszukiwanie zmiennych w instancjach, ich klasach oraz wszystkich klasach nadzędnych. Z punktu widzenia projektanta dziedziczenie jest sposobem określania przynależności do zbioru. Klasa definiuje zbiór właściwości, które mogą być dziedziczone oraz dostosowywane przez bardziej specyficzne podzbiory (czyli klasy podzadane).

By to zilustrować, powróćmy do naszego robota do robienia pizzy z początku tej części książki. Założymy, że zdecydowaliśmy się wybrać inną ścieżkę kariery i otwieramy pizzerię. Jednym z pierwszych naszych działań jest zatrudnienie pracowników obsługujących klientów czy przygotowujących jedzenie. Jednak ponieważ w głębi serca nadal jesteśmy inżynierami, decydujemy się zbudować robota wytwarzającego pizzę. Ze względu na poprawność polityczną oraz cybernetyczną uznajemy, że nasz robot będzie również pełnoprawnym pracownikiem z wynagrodzeniem.

Nasz zespół pracowników został w przykładowym pliku *employees.py* zdefiniowany przez cztery klasy. Klasa najbardziej ogólna (*Employee*) udostępnia wspólne zachowanie, takie jak wzrost wynagrodzenia (*giveRaise*) czy reprezentację tekstową (*__repr__*). Istnieją dwa typy pracowników i tym samym dwie klasy podzadane — *Chef*, czyli kucharz, oraz *Server*, czyli kelner. Obie przesyłają odziedziczoną metodę *work* w celu wyświetlania bardziej specyficznych komunikatów. Nasz robot wytwarzający pizzę należy do jeszcze bardziej wyspecjalizowanej klasy — *PizzaRobot* jest klasą potomną klasy *Chef*, która jest z kolei klasą potomną klasy *Employee*. W terminologii programowania zorientowanego obiektowo nazywamy te relacje *związkami typu „jest”* (ang. „*is-a*”) — robot jest kucharzem, który z kolei jest pracownikiem. Poniżej widać zawartość pliku *employees.py*.

```
class Employee:
    def __init__(self, name, salary=0):
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print self.name, "robi różne rzeczy"
    def __repr__(self):
        return "<Pracownik: imię=%s, wynagrodzenie=%s>" % (self.name, self.salary)

class Chef(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print self.name, "przygotowuje jedzenie"

class Server(Employee):
    def __init__(self, name):
        Employee.__init__(self, name, 40000)
    def work(self):
        print self.name, "obsługuje klienta"
```

```

class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__(self, name)
    def work(self):
        print self.name, "przygotowuje pizzę"

if __name__ == "__main__":
    bob = PizzaRobot('robert')                      # Tworzy robota o imieniu Robert
    print bob                                       # Wykonuje dziedziczoną metodę __repr__
    bob.work()                                      # Wykonuje działanie specyficzne dla typu
    bob.giveRaise(0.20)                             # Daje robotowi 20-procentową podwyżkę
    print bob; print

    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()

```

Kiedy wykonujemy ten moduł jako samodzielny program, zostanie uruchomiony kod testujący, który tworzy wytwarzającego pizzę robota o nazwie bob, dziedziczącego atrybuty po trzech klasach — PizzaRobot, Chef oraz Employee. Wyświetlenie obiektu instancji bob wykonuje metodę Employee.__repr__, a danie robotowi podwyżki wywołuje metodę Employee.giveRaise, ponieważ mechanizm dziedziczenia odnajdzie tę metodę właśnie w tej klasie.

```
C:\python\examples> python employees.py
<Employee: name=robert, salary=50000>
robert przygotowuje pizzę
<Employee: name=robert, salary=60000.0>
```

```
Employee robi różne rzeczy
Chef przygotowuje jedzenie
Server obsługuje klienta
PizzaRobot przygotowuje pizzę
```

W takiej hierarchii klas zazwyczaj możemy tworzyć instancje dowolnej klasy, nie tylko tej znajdującej się na dole. Pętla for w kodzie testującym moduł tworzy instancje wszystkich czterech klas. Każda z nich odpowiada w inny sposób, kiedy wywoła się metoda work, ponieważ w każdej klasie metoda ta jest inną. Tak naprawdę klasy symulują jedynie obiekty z prawdziwego świata. Metoda work wyświetla na razie tylko komunikat, jednak można ją rozszerzyć, tak by naprawdę wykonywała jakąś pracę.

Programowanie zorientowane obiektowo i kompozycja — związki typu „ma”

Pojęcie kompozycji zostało wprowadzone w rozdziale 25. Z punktu widzenia programisty kompozycja obejmuje osadzanie innych obiektów w obiekcie pojemnika i aktywację ich w taki sposób, by implementowały jego metody. Dla projektanta kompozycja to inny sposób reprezentowania związków w określonej dziedzinie zastosowania. Zamiast jednak określać przynależność, kompozycja wiąże się z komponentami — częściami całości.

Kompozycja odzwierciedla również związki pomiędzy częściami, zazwyczaj nazywane *związaniami typu „ma”* (ang. „has-a”). Niektóre teksty poświęcone projektowaniu w dziedzinie programowania zorientowanego obiektowo określają kompozycję jako *agregację* (lub rozróżniają te dwa pojęcia, wykorzystując agregację do opisania słabszych zależności pomiędzy pojemnikiem

a komponentem). W niniejszej książce pojęcie „kompozycja” odnosi się po prostu do zbioru osadzonych obiektów. Klasa kompozytowa udostępnia własny interfejs i implementuje go, kierując osadzonymi obiektami.

Skoro już mamy pracowników, umieścimy ich w naszej pizzerii i czymś zajmiemy. Nasza pizzeria jest obiektem kompozytowym — ma piec, a także pracowników, takich jak kucharze czy osoby obsługujące klientów. Kiedy klient wchodzi do lokalu i składa zamówienie, komponenty restauracji zaczynają działać — kelner przyjmuje zamówienie, kucharz wytwarza pizzę i tak dalej. Poniższy przykład o nazwie *pizzashop.py* symuluje wszystkie obiekty oraz związki w tym scenariuszu.

```
from employees import PizzaRobot, Server

class Customer:
    def __init__(self, name):
        self.name = name
    def order(self, server):
        print self.name, "zamawia od", server
    def pay(self, server):
        print self.name, "płaci za zamówienie", server

class Oven:
    def bake(self):
        print "piec piecze"

class PizzaShop:
    def __init__(self):
        self.server = Server('Ernest')          # Osadzenie innych obiektów
        self.chef = PizzaRobot('Robert')         # Robot o imieniu Robert
        self.oven = Oven()
    def order(self, name):
        customer = Customer(name)             # Aktywacja innych obiektów
        customer.order(self.server)            # Klient zamawia od kelnera
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)

if __name__ == "__main__":
    scene = PizzaShop()                    # Utworzenie kompozytu
    scene.order('Amadeusz')                # Symulacja zamówienia Amadeusza
    print '...'                            # Symulacja zamówienia Aleksandra
    scene.order('Aleksander')
```

Klasa *PizzaShop* jest pojemnikiem oraz kontrolerem. Jej konstruktor tworzy i osadza instancje klas pracowników, które utworzyliśmy przed chwilą, a także zdefiniowaną tutaj nową klasę *Oven*. Kiedy kod testujący tego modułu wywołuje metodę *order* klasy *PizzaShop*, osadzone obiekty są proszone o wykonanie swoich zadań jeden po drugim. Warto zwrócić uwagę na tworzenie nowego obiektu klasy *Customer* dla każdego zamówienia, a także przekazanie osadzonego obiektu *Server* do metod klasy *Customer*. Klienci pojawiają się i odchodzą, natomiast osoby ich obsługujące są częścią kompozytu pizzerii. Widać również, że pracownicy nadal znajdują się w relacji dziedziczenia — kompozycje oraz dziedziczenie są uzupełniającymi się narzędziami.

Kiedy wykonujemy ten moduł, nasza pizzeria obsługuje dwa zlecenia — jedno od Amadeusza, a drugie od Aleksandra.

```
C:\python\examples> python pizzashop.py
Amadeusz zamawia od <Pracownik: imię=Ernest, wynagrodzenie=40000>
Robert przygotowuje pizzę
```

```

piec piecze
Amadeusz płaci za zamówienie <Pracownik: imię=Ernest, wynagrodzenie=40000>
...
Aleksander zamawia od <Pracownik: imię=Ernest, wynagrodzenie=40000>
Robert przygotowuje pizzę
piec piecze
Aleksander płaci za zamówienie <Pracownik: imię=Ernest, wynagrodzenie=40000>
```

Jest to oczywiście dziecinna symulacja, jednak obiekty oraz interakcje są reprezentatywne dla działania kompozytów. Klasy mogą reprezentować dowolne obiekty i związki, jakie jesteśmy w stanie opisać słownie. Wystarczy zastąpić *rzeczowniki klasami*, a *czasowniki* metodami i mamy już pierwszy szkic projektu.

Raz jeszcze procesor strumienia danych

By zobaczyć nieco bardziej realistyczny przykład, warto powrócić do funkcji uniwersalnego procesora strumienia danych, utworzonej w części we wprowadzeniu do programowania zorientowanego obiektowo w rozdziale 25.

```
def processor(reader, converter, writer):
    while 1:
        data = reader.read()
        if not data: break
        data = converter(data)
        writer.write(data)
```

Zamiast korzystać tutaj z prostej funkcji, można również zapisać to w postaci klasy wykorzystującej kompozycję w celu uzyskania struktury i obsługiwanego dziedziczenia. W poniższym pliku *streams.py* zaprezentowano sposób umieszczenia tego w kodzie klasy.

```
class Processor:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer
    def process(self):
        while 1:
            data = self.reader.readline()
            if not data: break
            data = self.converter(data)
            self.writer.write(data)
    def converter(self, data):
        assert 0, 'konwerter musi być zdefiniowany' # lub wywołać wyjątek
```

Ta klasa definiuje metodę *converter()*, która musi być przeciążona w klasie potomnej. Jest to przykład modelu *klasy abstrakcyjnej*, szczegóły tej techniki można znaleźć w rozdziale 28. (informacje o asercjach znajdują się w części VII). Zapisane w ten sposób obiekty *reader* oraz *writer* osadzone są wewnątrz instancji klasy (*kompozycja*); logikę konwertera podajemy w klasie podzielonej, zamiast przekazywać funkcję konwertera (*dziedziczenie*). Plik *converters.py* pokazuje, jak można to zrobić.

```
from streams import Processor

class Uppercase(Processor):
    def converter(self, data):
        return data.upper()

if __name__ == '__main__':
    import sys
    obj = Uppercase(open('spam.txt'), sys.stdout)
    obj.process()
```

Klasa `Uppercase` dziedziczy logikę pętli przetwarzającej strumień (i wszystko, co może być zapisane w kodzie jej klas nadrzędnych). Musi zdefiniować tylko jeden unikalny element — logikę konwersji danych. Kiedy plik ten jest wykonywany, tworzy oraz wykonuje instancję wczytującą plik `spam.txt` i wypisującą odpowiednik tego pliku zapisany wielkimi literami do strumienia wyjścia `stdout`.

```
C:\lp4e> type spam.txt
mielonka
Mielonka
MIELONKA!

C:\lp4e> python converters.py
MIELONKA
MIELONKA
MIELONKA!
```

By przetwarzać inne rodzaje strumieni, należy przekazać inne rodzaje obiektów do wywołania konstruującego klasy. Poniżej zamiast strumienia wykorzystano plik wyjściowy.

```
C:\lp4e> python
>>> import converters
>>> prog = converters.Uppercase(open('spam.txt'), open('spamup.txt', 'w'))
>>> prog.process()

C:\lp4e> type spamup.txt
MIELONKA
MIELONKA
MIELONKA!
```

Jak jednak zasugerowano wcześniej, moglibyśmy również przekazać dowolne obiekty opakowane w klasy, definiujące wymagane interfejsy metod wejścia oraz wyjścia. Poniżej znajduje się prosty przykład przekazujący klasę zapisującą opakowującą tekst w znaczniki języka HTML.

```
C:\lp4e> python
>>> from converters import Uppercase
>>>
>>> class HTMLize:
...     def write(self, line):
...         print '<PRE>%s</PRE>' % line[:-1]
...
>>> Uppercase(open('spam.txt'), HTMLize()).process()
<PRE>MIELONKA</PRE>
<PRE>MIELONKA</PRE>
<PRE>MIELONKA!</PRE>
```

Sledząc przebieg sterowania w tym przykładzie, zobaczymy, że otrzymujemy zarówno konwersję tekstu na wielkie litery (dzięki dziedziczeniu), jak i formatowanie HTML (przez kompozycję), mimo że logika przetwarzania w oryginalnej klasie nadrzędnej `Processor` nie wie nic o żadnym z tych kroków. Dla kodu przetwarzającego znaczenie ma jedynie to, by obiekty zapisujące miały metodę `write`, a także by została zdefiniowana metoda o nazwie `convert`. Nieistotne jest to, co te wywołania tak naprawdę robią. Taki polimorfizm oraz hermetyzacja logiki są podstawą dużego potencjału klas.

W tej chwili klasa nadrzędna `Processor` udostępnia jedynie logikę przeglądającą plik. W bardziej realnym przykładzie moglibyśmy ją rozszerzyć w taki sposób, by obsługiwała dodatkowe narzędzia programistyczne dla klas podrzędnych i tym samym mogła stać się pełnowymiarową platformą. Utworzenie takiego narzędzia raz w klasie nadrzędnej pozwala nam je zastosować

ponownie we wszystkich naszych programach. Nawet w tym prostym przykładzie, ponieważ tak wiele zostało spakowane i odziedziczone w klasach, wystarczyło utworzyć kod z formatowaniem HTML. Całą resztę otrzymaliśmy gratis.

Inny przykład działania kompozycji można znaleźć w ćwiczeniu 9. pod koniec rozdziału 31. oraz w jego rozwiążaniu z dodatku B — jest on podobny do przykładu z pizzerią. W książce skupiliśmy się na dziedziczeniu, ponieważ jest to podstawowe narzędzie programowania zorientowanego obiektywnego w Pythonie. W praktyce jednak kompozycja wykorzystywana jest tak samo często jak dziedziczenie do strukturyzowania klas, w szczególności w większych systemach. Jak widzieliśmy, dziedziczenie oraz kompozycja często się dopełniają, a czasami są technikami alternatywnymi. Ponieważ kompozycja jest zagadnieniem z dziedziny projektowania pozostającym poza zakresem języka Python oraz niniejszej książki, osoby zainteresowane odsyłam po więcej informacji do innych źródeł.

Znaczenie klas oraz trwałości obiektów

W tej części książki kilka razy wspomniałem o serializacji za pomocą modułu `pickle`, ponieważ działa ona szczególnie dobrze w przypadku instancji klas. W rzeczywistości narzędzia do serializacji stanowią często wystarczająco atrakcyjną motywację do stosowania klas: zapisanie na dysku twardym obiektu klasy z użyciem modułu `pickle` lub `shelve` jest prostą implementacją mechanizmu magazynowania danych oraz logiki aplikacji.

Poza symulowaniem interakcji ze świata rzeczywistego utworzone tutaj klasy pizzerii mogły również zostać wykorzystane jako podstawa trwałej bazy danych restauracji. Instancje klasy mogą być przechowywane na dysku w jednym kroku za pomocą modułów `pickle` lub `shelve` Pythona. Modułu `shelve` użyliśmy w rozdziale 27. do zapisywania na dysku obiektów klas, również interfejs serializacji obiektów za pomocą modułu `pickle` jest zadziwiająco prosty w użyciu.

```
import pickle
object = someClass()
file = open(filename, 'wb')                                # Utworzenie pliku zewnętrznego
pickle.dump(object, file)                                  # Zapisanie obiektu w pliku

import pickle
file = open(filename, 'rb')
object = pickle.load(file)                                 # Pobranie go z powrotem później
```

Moduł `pickle` konwertuje obiekty znajdujące się w pamięci na zserializowane strumienie bajtów, które mogą być przechowywane w plikach czy przesypane za pośrednictwem sieci. Przeciwna operacja konwertuje strumień bajtów z powrotem na identyczne obiekty w pamięci. Moduł `shelve` jest podobny, jednak automatycznie serializuje on obiekty na bazę danych dostępną po kluczu, eksportującą interfejs podobny do słownika.

```
import shelve
object = someClass()
dbase = shelve.open('filename')
dbase['key'] = object                                    # Zapisanie pod kluczem

import shelve
dbase = shelve.open('filename')
object = dbase['key']                                    # Pobranie z powrotem później
```

W naszym przykładzie wykorzystanie klas do modelowania pracowników oznacza, że możemy otrzymać prostą bazę danych pracowników i firm za pomocą niewielkiej ilości pracy. Serializacja takich obiektów instancji do pliku sprawia, że są one trwałe pomiędzy niezależnymi wywołaniami programów w Pythonie.

```
>>> from pizzashop import PizzaShop
>>> shop = PizzaShop()
>>> shop.server, shop.chef
(<Pracownik: imię=Ernest, wynagrodzenie=40000>, <Pracownik: imię=Robert,
←wynagrodzenie=50000>)
>>> import pickle
>>> pickle.dump(shop, open('shopfile.dat', 'wb'))
```

Ta prosta operacja skutkuje zapisem do pliku na dysku całej skomplikowanej struktury obiektu `shop`. Taki magazyn danych może być odczytany w innej sesji tego samego programu, a nawet w zupełnie osobnym programie. Obiekty zapisane w ten sposób przechowują swój stan i zachowanie.

```
>>> import pickle
>>> obj = pickle.load(open('shopfile.dat', 'rb'))
>>> obj.server, obj.chef
(<Employee: name=Pat, salary=40000>, <Employee: name=Bob, salary=50000>)
>>> obj.order('Zuzanna')
Zuzanna zamawia od <Pracownik: imię=Ernest, wynagrodzenie=40000>
Robert przygotowuje pizzę
piec piecze
Zuzanna płaci za zamówienie <Pracownik: imię=Ernest, wynagrodzenie=40000>
```

Więcej przykładów użycia modułów `pickle` i `shelve` można znaleźć w dalszej części książki oraz w podręczniku standardowej biblioteki Pythona.

Programowanie zorientowane obiektowo a delegacja — obiekty „opakowujące”

Zwolennicy programowania zorientowanego obiektowo często wspominają o *delegacji*, co zazwyczaj dotyczy obiektów kontrolerów osadzających inne obiekty, do których przekazują żądania operacji. Kontrolery mogą się zajmować zadaniami administracyjnymi, takimi jak na przykład śledzenie dostępu. W Pythonie delegacja jest często implementowana za pomocą metody `__getattribute__`. Ponieważ przechwytuje ona dostęp do nieistniejących atrybutów, *klasa opakowująca* (czasami nazywana *klassą pośredniczącą*, ang. *wrapper* lub *proxy class*) może wykorzystać metodę `__getattribute__` do przekierowania dowolnego dostępu do opakowanego obiektu. Klasa pośrednicząca zachowuje interfejs opakowanego obiektu i może dodawać własne operacje.

Rozważmy na przykład poniższy plik `trace.py`.

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object
    def __getattribute__(self, attrname):
        print 'Śledzenie:', attrname
        return getattr(self.wrapped, attrname)
```

Łatwo sobie przypomnieć, że, zgodnie z informacjami z rozdziału 29., metoda `__getattribute__` pobiera nazwę atrybutu w postaci łańcucha znaków. Kod ten wykorzystuje wbudowaną

funkcję `getattr` do pobrania atrybutu z opakowanego obiektu po łańcuchu znaków jego nazwy — `getattr(X, N)` jest tym samym co `X.N`, oprócz tego, że `N` jest wyrażeniem, które w czasie wykonywania obliczane jest do łańcucha znaków, a nie do zmiennej. Tak naprawdę wywołanie `getattr(X, N)` jest podobne do `X.__dict__[N]`, jednak ta pierwsza wersja wykonuje również wyszukiwanie dziedziczenia, podobnie jak `X.N`, podczas gdy ta druga tego nie robi (więcej informacji na temat atrybutu `__dict__` można znaleźć w podrozdziale „Słowniki przestrzeni nazw” rozdziału 28.).

Takie podejście do klas opakowujących modułu można wykorzystać do zarządzania dostępem do dowolnego obiektu z atrybutami — list, słowników, a nawet klas oraz instancji. W poniższym przykładzie klasa `wrapper` po prostu wyświetla komunikat śledzenia dla każdego dostępu do atrybutu i deleguje żądanie atrybutu do opakowanego obiektu `wrapped`.

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])                                     # Opakowanie listy
>>> x.append(4)                                            # Delegacja do metody listy
Śledzenie: append
>>> x.wrapped                                              # Wyświetlenie mojej składowej
[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2})                           # Opakowanie słownika
>>> x.keys() 1                                             # Delegacja do metody słownika
Śledzenie: keys
['a', 'b']
```

Rezultatem jest rozszerzenie całego interfejsu opakowanego obiektu za pomocą dodatkowego kodu z klasy opakowującej. Takie rozwiązanie można wykorzystać na przykład do logowania wywołań metod albo przekierowywania wywołań metod do dodatkowej czy specjalnej logiki.

Do pojęć obiektów opakowanych oraz delegowanych operacji powróćmy w rozdziale 31. jako do jednego ze sposobów rozszerzania typów wbudowanych. Osoby zainteresowane wzorcem projektowym delegacji powinny zwrócić uwagę na omówienie *dekoratorów funkcji* w rozdziale 31. — koncepcje te są ze sobą mocno powiązane i zostały zaprojektowane z myślą o rozszerzaniu określonego wywołania funkcji lub metody zamiast całego interfejsu obiektu oraz *dekoratorów klas* w rozdziale 38., które służą do delegowania logiki wszystkich instancji klasy.



Uwaga na temat zgodności: W Pythonie 2.6 metody przeciążające operatory wywoływanie w kontekście wbudowanych operatorów są wywoływanie za pośrednictwem ogólnych mechanizmów uzyskiwania dostępu do atrybutów, jak metoda `__getattribute__()`. Na przykład wywołanie instrukcji `print` na takim obiekcie spowoduje wywołanie jego metody `__repr__()` lub `__str__()` z przekazanym obiektem. W Pythonie 3.0 taka sytuacja nie ma już miejsca: wywołanie funkcji `print()` nie wywołuje metody `__getattribute__()`, przez co wykorzystywana jest domyślna metoda reprezentacji tekstuowej obiektu. W 3.0 zaimplementowany w klasach nowego typu mechanizm wyszukiwania przeciążeń operatorów nie wykorzystuje mechanizmu przeszukiwania klas nadzędnych, szuka jedynie w klasie obiektu instancji. Do tego zagadnienia wrócimy w rozdziale 37. w kontekście atrybutów zarządzanych. W tej chwili należy jedynie pamiętać, że w Pythonie 3.0 w celu wykorzystania mechanizmu przeciążania operatorów przeciążanie należy zaimplementować w klasie opakowującej (bezpośrednio, z użyciem narzędzi lub klas nadzędnych).

Pseudoprыватне атрибуты клас

Obok zagadnień strukturalnych dobry projekt zorientowany obiektowo powinien uwzględniać kwestie użycia nazw. W części V dowiedzieliśmy się, że każda zmienna przypisana na najwyższym poziomie pliku modułu jest eksportowana. Domyślnie tak samo jest również w przypadku klas — ukrywanie danych jest konwencją, a klient może pobrać lub zmodyfikować każdy atrybut klasy lub instancji w taki sposób, na jaki ma ochotę. Tak naprawdę wszystkie atrybuty są — zgodnie z terminologią języka C++ — „publiczne” oraz „wirtualne”. Są dostępne z każdego miejsca i wyszukiwane dynamicznie w czasie wykonywania.¹

To wszystko nadal jest prawdziwe. Python obsługuje jednak również pojęcie „znieksztalconia” nazw zmiennych (to jest rozszerzania) w celu lokalizacji niektórych zmiennych w klasach. Znieksztalcone nazwy są czasami zwodniczo nazywane „atributami prywatnymi”, jednak tak naprawdę jest to jedynie sposób *lokalizacji* zmiennej do klasy, która ją utworzyła — znieksztalconie nie zapobiega dostępowi z kodu znajdującego się poza klasą. Opcja ta ma w zamierzeniach zapobiec konfliktom przestrzeni nazw w instancjach, a nie służyć do ogólnego ograniczenia dostępu do zmiennych. Znieksztalcone zmienne lepiej jest zatem nazywać „pseudoprыватnymi” niż „prywatnymi”.

Zmienne pseudoprыватne są zaawansowane i całkowicie opcjonalne. Prawdopodobnie nie przydadzą nam się do niczego, dopóki nie zaczniemy pisać dużych hierarchii klas w projektach tworzonych przez wielu programistów. W rzeczywistości nie zawsze są używane nawet tam, gdzie w teorii powinny. Programiści Pythona wewnętrzne nazwy w klasach najczęściej sygnalizują z użyciem przedrostka złożonego z pojedynczego podkreśnika (`_X`), co stanowi nieformalną konwencję nazewnictwa mającą sugerować, że mamy do czynienia z nazwą, której nie należy modyfikować (natomiast dla Pythona ta konwencja nie ma żadnego znaczenia).

Ponieważ jednak możemy spotkać się z tą opcją w kodzie utworzonym przez kogoś innego, musimy być przynajmniej świadomymi jej działania, nawet jeśli sami nie będziemy z niej korzystać.

Przegląd znieksztalconia nazw zmiennych

Poniżej widać, jak działa znieksztalconanie. Nazwy zmiennych wewnętrz instrukcji `class` zaczynające się od podwójnych znaków `_`, ale nie kończące się takimi samymi znakami, są automatycznie rozszerzane w taki sposób, by zawierały również nazwę zawierającej je klasy. Na przykład zmienna `_X` wewnątrz klasy o nazwie `Spam` zostaje automatycznie zmieniona na `_Spam__X` — oryginalną nazwę poprzedza się pojedynczym znakiem `_` oraz nazwą klasy zawierającą zmienną. Ponieważ zmodyfikowana nazwa zawiera nazwę klasy, jest w pewnym stopniu unikalna. Nie będzie wchodziła w konflikt z podobnymi nazwami utworzonymi przez inne klasy hierarchii.

¹ Wydaje się to niepotrzebnie niepokoić programistów języka C++. W Pythonie można nawet zmodyfikować lub całkowicie usunąć metodę klasy w czasie wykonywania. Z drugiej strony prawie nikt tego nie robi w prawdziwych programach. Jako język skryptowy Python bardziej podkreśla umożliwianie niż ograniczanie. Można sobie przypomnieć, że w omówieniu przeciążania operatorów w rozdziale 29. wspomnialiśmy, iż metody `__getattr__` oraz `__setattr__` można wykorzystać do emulacji prywatności. Zazwyczaj nie są one jednak używane w tym celu w praktyce. Wróćmy do tego tematu przy okazji tworzenia bardziej realistycznego dekoratora prywatności w rozdziale 38.

Znieksztalcanie nazw ma miejsce jedynie w instrukcji `class` i tylko dla zmiennych, których nazwy rozpoczynają się od dwóch znaków `_`. Ma jednak miejsce dla *każdej* nazwy poprzedzonej dwoma takimi znakami, w tym nazw metody oraz nazw atrybutów instancji (na przykład w klasie `Spam` referencja do atrybutu instancji `self.__X` zostałaby przekształcona na `self._Spam__X`). Ponieważ dodawać atrybuty do instancji może więcej niż jedna klasa, znieksztalcanie pomaga zapobiec konfliktom. Żeby jednak zobaczyć, jak się to odbywa, musimy przejść do przykładu.

Po co używa się atrybutów pseudoprzywatnych?

Jednym z podstawowych problemów, które ma rozwiązać stosowanie nazw pseudoprzywatnych, jest sposób zapisu atrybutów instancji klas. W Pythonie wszystkie atrybuty instancji są przyłączone do jego obiektu na samym dole drzewa dziedziczenia. To różni Pythona od modelu stosowanego w C++, w którym każda klasa otrzymuje własną przestrzeń dla zdefiniowanych w niej atrybutów (danych).

Wewnątrz metody klasy w Pythonie za każdym razem, gdy metoda przypisuje coś do atrybutu `self` (na przykład `self.atrybut = wartość`), modyfikuje lub tworzy atrybut w instancji (wyszukiwanie dziedziczenia odbywa się jedynie przy referencjach, nie przy przypisaniu). Ponieważ jest tak nawet wtedy, gdy wiele klas z hierarchii przypisuje wartość do tego samego atrybutu, konflikty są możliwe.

Załóżmy na przykład, że kiedy programista tworzy klasę, zakłada, że ma atrybut o nazwie `X` w instancji. W metodach tej klasy nazwa ta zostaje ustawniona, a później pobrana.

```
class C1:  
    def meth1(self): self.X = 88 # Zakładam, że X jest moje  
    def meth2(self): print (self.X)
```

Załóżmy jeszcze, że inny programista, pracujący w odosobnieniu, poczyni to samo założenie w tworzony przez siebie klasie.

```
class C2:  
    def metha(self): self.X = 99 # Ja też tak uważam  
    def methb(self): print (self.X)
```

Obie klasy same z siebie działają dobrze. Problem pojawia się, kiedy zostaną one zmieszane ze sobą w tym samym drzewie klas.

```
class C3(C1, C2): ...  
I = C3() # Tylko jedna zmienna X w I!
```

Teraz wartość zwracana przez każdą z klas dla kodu `self.X` będzie uzależniona od tego, która klasa przypisała ją jako ostatnia. Ponieważ wszystkie przypisy do `self.X` odnoszą się do jednego instancji, istnieje tylko jeden atrybut `X` — `I.X` — bez względu na to, ile klas wykorzystuje tę samą nazwę atrybutu.

By zagwarantować, że atrybut należy do klasy, która go wykorzystuje, należy poprzedzić jego nazwę podwójnymi znakami `_` w każdym miejscu, w jakim jest użyty w klasie, tak jak w poniższym pliku `private.py`.

```
class C1:  
    def meth1(self): self.__X = 88 # Teraz X jest moje  
    def meth2(self): print(self.__X) # Staje się __C1__X w I  
  
class C2:  
    def metha(self): self.__X = 99 # Jest też moje  
    def methb(self): print(self.__X) # Staje się __C2__X w I
```

```

class C3(C1, C2): pass
I = C3()
# Dwie zmienne X w I

I.meth1(); I.metha()
print I.__dict__
I.meth2(); I.methb()

```

Dzięki takiemu przedrostkowi atrybuty `X` zostaną przed dodaniem do instancji rozszerzone w taki sposób, by obejmować również nazwy klas. Jeśli wykonamy wywołanie `dir` dla `I` lub przejrzymy słownik przestrzeni nazw tej instancji po przypisaniu atrybutów, zobaczymy w nim rozszerzone zmienne `_C1__X` oraz `_C2__X`, ale już nie samo `X`. Ponieważ takie rozszerzenie sprawia, że nazwy te stają się unikalne wewnętrz instancji, osoby tworzące klasy mogą zakładać, że są prawdziwymi właścicielami wszystkich zmiennych poprzedzonych dwoma znakami `_`.

```

% python private.py
{'_C2__X': 99, '_C1__X': 88}
88
99

```

Powyzsza sztuczka pozwala uniknąć potencjalnych konfliktów nazw w instancji, jednak należy pamiętać, że nie jest to jednoznaczne z prawdziwą prywatnością zmiennych. Jeśli znamy nazwę klasy zawierającej zmienną, nadal możemy uzyskać dostęp do atrybutów w dowolnym miejscu, w którym mamy referencję do instancji, wykorzystując rozszerzoną nazwę zmiennej (na przykład `I._C1__X = 77`). Z drugiej strony, opcja ta sprawia, że o wiele mniej prawdopodobne jest przypadkowe natknięcie się na zmienne klas.

Atrybuty pseudoprzywatne przydają się również w większych bibliotekach lub narzędziach. Pozwalają uniknąć przypadkowego przesłonięcia nazwy w którejś z klas potomnych. Jeśli dana metoda ma być używana wyłącznie w jednej klasie, która może być osadzana w innych klasach, nazwa poprzedzona dwoma znakami podkreślenia nie będzie kolidowała z innymi nazwami w drzewie dziedziczenia, co jest szczególnie użyteczne w przypadku dziedziczenia wielokrotnego.

```

class Super:
    def method(self): ...                                # Zwykła metoda

class Tool:
    def __method(self): ...                            # Nazwa zostanie niejawnie zmieniona na _Tool__method
    def other(self): self.__method()                  # Użycie metody wewnętrznej

class Sub1(Tool, Super): ...
    def actions(self): self.method()                 # Wywołuje Super.method()

class Sub2(Tool):
    def __init__(self): self.method = 99            # Nie psuje Tool.__method

```

Z wielokrotnym dziedziczeniem spotkaliśmy się już w rozdziale 25., a w dalszej części tego rozdziału pogłębimy to zagadnienie. Jak pamiętamy, klasy nadzędne są przeszukiwane w kolejności od lewej do prawej. W powyższym przykładzie oznacza to, że obiekty klasy `Sub1` będą preferowały atrybuty zdefiniowane w klasie `Tool` — przed atrybutami zdefiniowanymi w klasie `Super`. W tym przykładzie moglibyśmy wpływać na tę kolejność, zamieniając w deklaracji kolejność klas nadzędnych, ale dzięki metodom pseudoprzywatnym mamy jeszcze większą kontrolę. Nazwy pseudoprzywatne zabezpieczają przed przypadkowym przesłonięciem nazwy, co demonstruje definicja klasy `Sub2`.

Podkreśl jeszcze raz, że ta cecha języka powstała z myślą o dużych projektach, w których bierze udział wielu programistów. Jednak również w takich przypadkach nie należy przesadzać.

Nie warto niepotrzebnie zaśmiecać kodu. Mechanizm ten powinien być stosowany wyłącznie w celu zapewnienia kontroli nad wybranymi nazwami przez określoną klasę. W przypadku prostych programów użycie tej konstrukcji najczęściej mija się z celem.

Więcej przykładów zastosowania nazw atrybutów w konwencji `_X` można znaleźć w dalszej części rozdziału przy okazji omawiania modułu `lister.py`, w punkcie poświęconym wielokrotnemu dziedziczeniu oraz w punkcie omawiającym dekorator klas `Private` w rozdziale 38. Jeśli ktoś jest zainteresowany emulacją prywatnych atrybutów instancji, szkic takiego rozwiązania znajdzie w punkcie „Metody `__getattr__` oraz `__setattr__` przechwytyują referencje do atrybutów” w rozdziale 29. oraz w opartym na tej koncepcji wspomnianym dekoratorze klas `Private` w rozdziale 38. W klasach Pythona mamy możliwość pełnej kontroli nad dostępem do atrybutów, jednak takie podejście jest praktykowane niezwykle rzadko, nawet w bardzo dużych systemach.

Metody są obiektami — z wiązaniem i bez wiązania

Metody, a w szczególności metody związane, upraszczają implementację wielu wzorców projektowych w Pythonie. Metody związane omawialiśmy w skrócie w rozdziale 29., przy okazji omawiania metody `__call__()`. W niniejszym rozdziale omówimy ten temat bardziej szczegółowo i przekonamy się, że to mechanizm znacznie bardziej ogólny i elastyczniejszy, niż mogłoby się wydawać.

W rozdziale 19. dowiedzieliśmy się, w jaki sposób przetwarzarć funkcje, traktując je jak zwykłe obiekty. Również metody są obiektami i można ich używać podobnie jak innych obiektów: można im przypisywać wartości, przekazywać je do funkcji, zapisywać w strukturach danych itp. Dostęp do metod klas uzyskuje się z instancji lub klasy i dlatego w Pythonie istnieją dwa rodzaje metod.

Obiekty metod klas bez wiązania (ang. unbound) — bez `self`

Dostęp do atrybutu funkcji klasy za pomocą składni kwalifikującej z klasą zwraca obiekt metody bez wiązania. By wywołać metodę, musimy w sposób jawnego dostarczyć jej w pierwszym argumencie obiekt instancji. W Pythonie 3.0 metoda niezwiązana jest traktowana tak samo jak zwykła funkcja i może być wywołana z poziomu klasy. W 2.6 metody niezwiązane są osobnym typem i nie można ich wywoływać bez podania instancji.

Obiekty metod instancji z wiązaniem (ang. bound) — pary `self + funkcja`

Dostęp do atrybutu funkcji klasy za pomocą składni kwalifikującej z instancją zwraca obiekt metody z wiązaniem. Python automatycznie pakuje instancję z funkcją w obiekt metody z wiązaniem, dzięki czemu nie musimy do wywołania tej metody przekazywać instancji.

Oba rodzaje metod są pełnowymiarowymi obiektami — można je przekazywać czy przechowywać w listach. Oba wymagają również przy wykonywaniu podania w pierwszym argumencie instancji (to znaczy wartości dla `self`). Dlatego właśnie musielibyśmy przekazać instancję w sposób jawny, kiedy wywoływaliśmy metody klas nadrzędnych z metod klas podrzędnych w poprzednim rozdziale. Z technicznego punktu widzenia wywołania takie tworzą obiekty metod bez wiązania.

Kiedy wywołuje się obiekt metody z wiązaniem, Python automatycznie udostępnia nam instancję — wykorzystywaną do tworzenia obiektu metody z wiązaniem. Oznacza to, że obiekty

metod z wiązaniem są zazwyczaj wymienne z prostymi obiektami funkcji, co czyni je szczególnie przydatnymi dla interfejsów oryginalnie utworzonych dla funkcji (realistyczny przykład można znaleźć w ramce „Znaczenie metod z wiązaniem oraz wywołań zwrotnych”).

By to zilustrować, założymy, że definiujemy następującą klasę.

```
class Spam:  
    def doit(self, message):  
        print message
```

Teraz w normalnej operacji tworzymy instancję i wywołujemy jej metodę w celu wyświetlenia przekazanego argumentu.

```
object1 = Spam()  
object1.doit('Witaj, świecie!')
```

Tak naprawdę jednak po drodze, tuż przed nawiasami wywołania metody, generowany jest obiekt metody z *wiązaniem*. Możemy nawet pobrać metodę z wiązaniem bez prawdziwego wywoływania jej. Składnia kwalifikująca *obiekt.atrybut* jest wyrażeniem obiektu. W poniższym kodzie zwraca ona obiekt metody z wiązaniem, pakujący instancję (*object1*) z metodą funkcji (*Spam.doit*). Możemy przypisać tę metodę z wiązaniem do innej nazwy, a następnie wywołać ją tak, jakby była prostą funkcją.

```
object1 = Spam()  
x = object1.doit  
x('Witaj, świecie!')  
# Obiekt metody z wiązaniem: instancja + funkcja  
# Ten sam efekt co object1.doit('...')
```

Z drugiej strony, jeśli użyjemy składni kwalifikującej z klasą w celu dotarcia do metody *doit*, z powrotem otrzymamy obiekt metody *bez wiązania*, będący po prostu referencją do obiektu funkcji. By wywołać metodę tego typu, musimy przekazać instancję jako argument znajdujący się najbardziej na lewo.

```
object1 = Spam()  
t = Spam.doit  
t(object1, 'siema')  
# Obiekt metody bez wiązania  
# Przekazanie instancji
```

Przez rozszerzenie te same reguły mają zastosowanie wewnątrz metody klasy, jeśli odniesiemy się do atrybutów *self* odnoszących się do funkcji w klasie. Wyrażenie *self.metoda* jest obiektem metody z wiązaniem, ponieważ *self* jest obiektem instancji.

```
class Eggs:  
    def m1(self, n):  
        print n  
    def m2(self):  
        x = self.m1  
        x(42)  
Eggs().m2()  
# Inny obiekt metody z wiązaniem  
# Wygląda jak prosta funkcja  
# Wyświetla 42
```

Zazwyczaj metody wywołuje się natychmiast po pobraniu ich za pomocą składni kwalifikującej, dlatego nie zawsze zauważamy obiekty wygenerowane po drodze. Jeśli jednak zaczniemy pisać kod wywołujący obiekty w sposób uniwersalny, musimy uważać na specjalne traktowanie metod bez wiązania — normalnie wymagają one jawnego przekazania obiektu instancji.²

² Opcjonalny wyjątek od tej reguły można zobaczyć w rozdziale 31. przy omówieniu metod statycznych oraz metod klasy. Tak jak metody z wiązaniem, oba te typy mogą się również maskować jako normalne funkcje, ponieważ przy wywołaniu nie oczekują one instancji.

Metody niezwiązane w 3.0

W Pythonie 3.0 porzucono koncepcję metod *niezwiązańych*. To, co w niniejszym rozdziale nazywamy metodą niezwiązaną, jest w Pythonie 3.0 traktowane tak samo jak funkcja. W większości przypadków nie powinno to mieć znaczenia w napisanym kodzie — w pierwszym parametrze wywołania takiej metody wywołanej na obiekcie otrzymamy obiekt klasy.

Jedyny problem będą stanowić programy, w których stosowana jest kontrola typów. W Pythonie 2.6 wywołanie funkcji type na takiej metodzie klasy zwróci ciąg znaków unbound method, natomiast w Pythonie 3.0 otrzymamy function.

Co więcej, w 3.0 wszystkie metody można wywoływać bez obiektu (instancji klasy), jeśli metoda go nie potrzebuje, dzięki czemu można je wywoływać z poziomu klasy, nie obiektu. Innymi słowy, Python 3.0 przekaże instancję do metody tylko w przypadku wywołania tej metody z poziomu instancji. W przypadku wywołania z poziomu klasy instancja (o ile jest potrzebna metodzie) musi być przekazana ręcznie.

```
C:\misc> c:\python30\python
>>> class Selfless:
...     def __init__(self, data):
...         self.data = data
...     def selfless(arg1, arg2):          # w 3.0 jest zwykła funkcja
...         return arg1 + arg2
...     def normal(self, arg1, arg2):      # oczekuje instancji
...         return self.data + arg1 + arg2
...
>>> X = Selfless(2)
>>> X.normal(3, 4)                  # instancja jest przekazana automatycznie
9
>>> Selfless.normal(X, 3, 4)        # parametr self oczekiwany przez metodę przekazany ręcznie
9
>>> Selfless.selfless(3, 4)         # brak instancji: działa w 3.0, ale nie w 2.6!
7
```

Ostatnie wywołanie z powyższego przykładu w Pythonie 2.6 zakończy się błędem, ponieważ metody niezwiązane wymagają przekazania instancji klasy. W 3.0 wywołanie to zadziała dzięki temu, że metody niezwiązane są traktowane jak funkcje i nie wymagają przekazania instancji. Jest to zmiana, która z jednej strony pozbawia programistę wsparcia ze strony mechanizmów kontrolnych Pythona (na przykład w przypadku, gdy programistaomyłkowo nie przekazał instancji klasy), ale z drugiej daje więcej swobody, pozwalając używać metody tak samo jak zwykłych funkcji, jeśli instancja klasy nie zostanie przekazana, a w metodzie nie jest wykorzystywany argument self.

Poniższe dwa wywołania nie zadziałają ani w 2.6, ani w 3.0. Pierwsze z nich (wywołane z obiektu) powoduje automatyczne przekazanie obiektu do metody, która go nie oczekuje. Drugie (wywołane z klasy) nie przekazuje klasy do metody, która jej oczekuje:

```
>>> X.selfless(3, 4)
TypeError: selfless() takes exactly 2 positional arguments (3 given)
>>> Selfless.normal(3, 4)
TypeError: normal() takes exactly 3 positional arguments (2 given)
```

Dzięki tej zmianie od wersji 3.0 do definiowania metod nieobsługujących parametru self nie jest konieczne wykorzystywanie opisanego w następnym rozdziale dekoratora staticmethod(). Tego typu metody są wywoływane jak zwykłe funkcje, bez przekazywania obiektu w argu-

mencie. W 2.6 tego typu wywołania wywołują błąd, chyba że obiekt instancji będzie przekazany ręcznie. Więcej szczegółów na temat dekoratora `staticmethod()` poznamy w następnym rozdziale.

Należy znać zmiany wprowadzone w zakresie metod niezwiązanych w Pythonie 3.0, ale metody te mają o wiele większe znaczenie praktyczne. Metoda wiązana stanowi połączenie konkretnego obiektu z funkcją, więc można ją traktować tak samo jak inne obiekty wywoływanie (ang. *callable*). W następnym punkcie zademonstruję praktyczne zastosowania tej właściwości.



Aby lepiej zrozumieć sposoby wykorzystania metod niezwiązańych w Pythonie 3.0 i 2.6, warto zająć się przykładem `lister.py` w punkcie omawiającym wielokrotne dziedziczenie w dalszej części rozdziału. Przedstawiam tam implementację klas służących do wypisywania atrybutów instancji i klas, działających w obydwu wersjach Pythona.

Metody związane i inne obiekty wywoływanie

Jak wspomniałem wcześniej, metody związane mogą być przetwarzane jako uogólnione obiekty, podobnie jak zwykłe funkcje. Innymi słowy, metody można przekazywać tak samo jak inne obiekty. Co więcej, dzięki temu, że metody związane łączą w sobie funkcję (logikę) i dane (obiekt), mogą być wywoływanie jak każdy inny obiekt wywoływanego (*callable*) i nie wymagają stosowania specjalnej składni. W poniższym listingu definiujemy cztery obiekty metod związanych i wykorzystujemy je później tak, jak zwykłe wywołania funkcji.

```
>>> class Number:
...     def __init__(self, base):
...         self.base = base
...     def double(self):
...         return self.base * 2
...     def triple(self):
...         return self.base * 3
...
>>> x = Number(2)                                     # Obiekty klasy
>>> y = Number(3)                                     # Stan i metody
>>> z = Number(4)
>>> x.double()                                       # Zwykłe wywołania bezpośrednie
4
>>> acts = [x.double, y.double, y.triple, z.double] # Lista metod związanych
>>> for act in acts:                                # Opoźnienie wywołań
...     print(act())                                    # Wywołanie w trybie zwykłych funkcji
...
4
6
9
8
```

Obiekty metod związanych oferują mechanizm introspekcji, podobnie jak zwykłe funkcje. Dzięki temu mamy dostęp do związanego obiektu instancji klasy oraz funkcji metody. Wywołanie obiektu metody związanej powoduje wywołanie tej funkcji na obiekcie instancji:

```
>>> bound = x.double
>>> bound.__self__, bound.__func__
(<__main__.Number object at 0x0278F610>, <function double at 0x027A4ED0>)
>>> bound.__self__.base
2
>>> bound()                                         # Wywołuje bound.__func__(bound.__self__, ...)
4
```

Metody związane to w rzeczywistości jeden z przykładów użytecznych obiektów wywoływalnych oferowanych przez Pythona. Poniższy listing ilustruje właściwości obiektów wywoływalnych — funkcje definiowane z użyciem instrukcji `def`, `lambda`, jak również instancje klas definiujących metodę `__call__` obsługującą interfejs wywoływania i można je wykorzystywać w ten sam sposób:

```
>>> def square(arg):
...     return arg ** 2
...
>>> class Sum:
...     def __init__(self, val):
...         self.val = val
...     def __call__(self, arg):
...         return self.val + arg
...
>>> class Product:
...     def __init__(self, val):
...         self.val = val
...     def method(self, arg):
...         return self.val * arg
...
>>> sobject = Sum(2)
>>> pobject = Product(3)
>>> actions = [square, sobject, pobject.method] # Funkcja, instancja, metoda
...
>>> for act in actions:                         # wszystkie z nich wywołuje się tak samo
...     print(act(5))                            # jako obiekt wywoływany z jednym argumentem
...
25
7
15
>>> actions[-1](5)                           # Indeksowanie, składanie, mapy
15
>>> [act(5) for act in actions]
[25, 7, 15]
>>> list(map(lambda act: act(5), actions))
[25, 7, 15]
```

Z technicznego punktu widzenia wszystkie klasy są również obiektami wywoływanymi, ale różnica polega na tym, że wywołuje się je w celu utworzenia instancji, a nie do wykonania określonej pracy.

```
>>> class Negate:
...     def __init__(self, val):
...         self.val = -val
...     def __repr__(self):           # Klasa też są obiektami wywoływanymi
...         return str(self.val)    # Wywołanie z poziomu obiektu nie zadziała
...                             # Format reprezentacji tekstowej
...
>>> actions = [square, sobject, pobject.method, Negate] # Wywołanie klasy
>>> for act in actions:
...     print(act(5))
...
25
7
15
-5
>>> [act(5) for act in actions]                  # Wywołuje __repr__(), nie __str__()!
[25, 7, 15, -5]
...
>>> table = {act(5): act for act in actions}      # Składanie słownika z 2.6/3.0
>>> for (key, value) in table.items():
...     print('{0:2} => {1}'.format(key, value))       # Formatowanie ciągu znaków w 2.6/3.0
```

```
...
-5 => <class '__main__.Negate'>
25 => <function square at 0x025D4978>
15 => <bound method Product.method of <__main__.Product object at 0x025D0F90>>
7 => <__main__.Sum object at 0x025D0F70>
```

Jak widzimy, metody związane, a ogólniej: model obiektów wykonywanych w Pythonie, to tylko wybrane dowody na to, że Python został zaprojektowany jako niezwykle elastyczny język programowania.

W tym momencie Czytelnik powinien już rozumieć model obiektów metod. Inne przykłady metod związanych można znaleźć w ramce „Znaczenie metod z wiązaniem oraz wywołań zwrotnych” oraz w poprzednim rozdziale — przy okazji dyskusji na temat metody `__call__`.

Znaczenie metod z wiązaniem oraz wywołań zwrotnych

Ponieważ metody z wiązaniem automatycznie łączą instancje oraz funkcje metod klasy w pary, możemy je wykorzystywać we wszystkich miejscach, w których oczekiwana jest prosta funkcja. Jednym z najczęściej spotykanych miejsc, w których możemy zobaczyć zastosowanie tego pomysłu, jest kod rejestrujący metody jako programy obsługi zdarzeń w graficznym interfejsie użytkownika Tkinter. Poniżej znajduje się prosty przykład.

```
def handler():
    ...wykorzystanie zmiennych globalnych dla stanu...
...
widget = Button(text='mielonka', command=handler)
```

By zarejestrować program obsługi zdarzeń kliknięcia przycisku, zazwyczaj przekazujemy obiekt wywoływalny nieprzymającą argumentów do argumentu ze słowem kluczowym `command`. Działają tutaj nazwy funkcji (oraz `lambda`), a także metody klas, o ile są one metodami z wiązaniem.

```
class MyWidget:
    def handler(self):
        ...użycie self.attr dla stanu...
    def makewidgets(self):
        b = Button(text='mielonka', command=self.handler)
```

W powyższym kodzie programem obsługi zdarzeń jest `self.handler` — obiekt metody z wiązaniem pamiętający zarówno `self`, jak i `MyWidget.handler`. Ponieważ kiedy metoda `handler` jest później wywoływana w momencie wystąpienia zdarzenia, `self` odnosi się do oryginalnej instancji, metoda będzie miała dostęp do atrybutów instancji, które mogą przechowywać stan pomiędzy zdarzeniami. W prostych funkcjach stan musiałby normalnie zostać zachowany w zmiennych globalnych. Inny sposób zapewnienia zgodności klas z API opartym na funkcjach można znaleźć w omówieniu metody przeciążania `__call__` w rozdziale 29.

Dziedziczenie wielokrotne — klasy mieszane

Wiele projektów wykorzystujących klasy wykorzystuje skomplikowane zestawy metod z użyciem dziedziczenia. W instrukcji `class` w nawiasach znajdujących się w wierszu nagłówka można wymienić więcej niż jedną klasę nadzczną. Jeśli tak zrobimy, wykorzystujemy coś, co nosi nazwę *dziedziczenia wielokrotnego* (ang. *multiple inheritance*) — klasa oraz jej instancje dziedziczą zmienne po *wszystkich* wymienionych klasach nadzędnych.

Kiedy próbujemy odnaleźć atrybut, Python przeszukuje klasy nadrzędne z wiersza nagłówka klasy od lewej do prawej strony tak długo, aż nie znajdzie dopasowania. Ścisłe rzeczą biorąc, proces wyszukiwania postępuje od dołu do góry drzewa dziedziczenia, a następnie od lewej strony do prawej, ponieważ każda klasa nadrzędna może mieć swoje własne klasy nadrzędne.

- W klasach „klasycznych” (stosowanych domyślnie przed wersją 3.0) wyszukiwanie nazw odbywało się metodą „w górę drzewa dziedziczenia, a następnie od lewej do prawej”.
- W klasach nowego stylu (dotyczy to również wszystkich klas w Pythonie 3.0) wyszukiwanie nazw odbywa się w sposób bardziej skomplikowany. Szczegóły można znaleźć w następnym rozdziale przy okazji dyskusji o nowych klasach.

W każdym z tych modeli w przypadku użycia dziedziczenia wielokrotnego nazwy są wyszukiwane od lewej do prawej, zgodnie z kolejnością w deklaracji dziedziczenia.

Dziedziczenie wielokrotne przydaje się do modelowania obiektów należących do więcej niż jednego zbioru. Osoba może na przykład być inżynierem, pisarzem oraz muzykiem i może dziedziczyć właściwości po wszystkich tych zbiorach. W przypadku wielokrotnego dziedziczenia obiekty uzyskują dostęp do wszystkich nazw zadeklarowanych w klasach nadrzędnych.

Chyba najpopularniejszym sposobem wykorzystywania dziedziczenia wielokrotnego jest mieszanie metod ogólnego przeznaczenia z klasami nadrzędnymi. Takie klasy nadrzędne nazywane są zazwyczaj *klasami mieszanymi* (ang. *mix-in class*) — udostępniają one metody dodawane do klas aplikacji za pomocą dziedziczenia. W przeciwieństwie do prostych funkcji w modułach metody w klasach mieszanych mają dostęp również do obiektu instancji, za pośrednictwem którego uzyskują dostęp do stanu i innych metod obiektu. Przykładowo domyślny sposób wyświetlania obiektu instancji klasy w Pythonie nie jest szczególnie przydatny.

Tworzenie klas mieszanych

Jak widzieliśmy, domyślny dla Pythona sposób wyświetlania informacji o klasach nie jest szczególnie użyteczny:

```
>>> class Spam:  
...     def __init__(self):  
...         self.data1 = "jedzenie"  
...  
>>> X = Spam()  
>>> print X  
<__main__.Spam instance at 0x00864818>>  
# Nie ma __repr__  
# Wygląd domyślny: klasa, adres  
# W Pythonie 2.6 wyświetla "instance"
```

Jak widzieliśmy w rozdziale 29. poświęconym przeciążaniu operatorów, możemy udostępnić własną metodę `__str__()` lub `__repr__()` implementującą naszą reprezentację tekstową. Zamiast jednak zapisywać kod metody `__repr__()` w każdej klasie, którą chcemy wyświetlić, dlaczego nie zapisać jej raz w klasie narzędzia ogólnego przeznaczenia dziedziczonej przez wszystkie klasy?

Do tego właśnie służą klasy mieszane. Definiując na przykład metodę reprezentacji tekstowej w klasie nadrzędnej, mamy możliwość użycia jej do wypisania informacji o obiekcie dziedziczącej po niej klasie mieszanej. Analogiczne możliwości mieliśmy już okazję spotkać również przy innych okazjach:

- rozdział 27.: klasa `attrDisplay` formatowała atrybuty instancji klasy i wypisywała je za pomocą generycznej metody `__str__()`, ale nie zastosowaliśmy wówczas wielopoziomowej hierarchii klas, a jedynie pojedyncze dziedziczenie,
- rozdział 28.: moduł `classtree.py` definiował funkcje służące do nawigowania i rysowania schematów drzew dziedziczenia, ale kod ten nie wyświetlał atrybutów obiektów, nie był też zaprojektowany z myślą o dziedziczeniu.

Rozwińmy techniki prezentowane w tych przykładach, tworząc zestaw trzech klas mieszanych implementujących narzędzia do wyświetlania atrybutów obiektów, atrybutów dziedziczonych oraz atrybutów wszystkich obiektów w drzewie klas. Naszych narzędzi użyjemy w kontekście wielodziedziczenia, prezentując przy okazji techniki programowania dostosowane do tworzenia uogólnionych narzędzi.

Odczyt listy atrybutów obiektu — `__dict__()`

Zaczniemy od prostego przypadku: wypisywania atrybutów instancji. Poniższy listing (moduł `lister.py`) definiuje klasę mieszana `ListInstance` przeciążającą metodę `__str__()`. Ta metodą będzie wykorzystana we wszystkich klasach dziedziczących po `ListInstance`. Dzięki temu, że ta logika została zaimplementowana w postaci klasy, `ListInstance` jest uogólnionym narzędziem, którego funkcję można wykorzystać w dowolnej klasie potomnej.

```
# Plik lister.py
class ListInstance:
    """
    Klasa ListInstance obsługuje formatowanie ciągów znaków zwracanych
    z funkcji print() i str(). Mechanizm jest definiowany w metodzie specjalnej
    __str__(), w naszym przypadku wypisuje nazwy i wartości atrybutów.
    Argument self jest instancją klasy na najniższym poziomie dziedziczenia.
    W przykładzie wykorzystujemy atrybuty pseudoprывatne __X w celu
    uniknięcia konfliktów nazw z atrybutami klas z drzewa dziedziczenia.
    """
    def __str__(self):
        return '<Instancja klasy %s, adres %s:\n%s>' % (
            self.__class___.name__, # nazwa klasy
            id(self), # adres
            self.__attrnames())
    def __attrnames(self):
        result = ''
        for attr in sorted(self.__dict__): # słownik atrybutów instancji
            result += '\tnazwa %s=%s\n' % (attr, self.__dict__[attr])
        return result
```

Klasa `ListInstance` wykorzystuje niektóre ze znanych nam już technik introspekcji nazwy klasy oraz atrybutów obiektu:

- Każda instancja obsługuje wbudowany atrybut `__class__` wskazujący klasę, z której została ona utworzona. Każda klasa obsługuje atrybut `__name__` określający jej nazwę, zatem odwołanie `__class__.....name__` zwraca nazwę klasy obiektu.
- Nasza klasa wykonuje większość swojej pracy, wykorzystując słownik atrybutów instancji (udostępniany w atrybucie `__dict__`), z którego buduje ciąg znaków wypisujący nazwy i wartości atrybutów. Klucze odczytane z tego słownika są wstępnie sortowane w celu

Pozostałe mechanizmy klasy `ListInstance` przypominają prezentowany w rozdziale 27. mechanizm wypisywania atrybutów — w zasadzie ten przykład można uznać za jedną z wariacji na temat tego zagadnienia. Warto jednak zwrócić uwagę na dwie cechy podejścia wykorzystanego w tej klasie:

- wypisuje adres instancji w pamięci, wykorzystując wbudowaną funkcję `id()` (jest to z definicji unikalny identyfikator obiektu, co przyda się w przyszłych modyfikacjach naszego kodu),
- wykorzystuje wzorzec nazw *pseudoprivatnych* do obsługi metody `__attnames`. Jak dowiedzieliśmy się wcześniej w tym rozdziale, Python automatycznie modyfikuje tego typu nazwę, dopisując do niej nazwę klasy (w tym przypadku nazwa zmienia się na `_ListInstance__attnames`). Ta zasada obowiązuje w stosunku do atrybutów klas (również metod) oraz związanych z nimi obiektów instancji. Ten mechanizm przydaje się w przypadku ogólnych narzędzi, ponieważ pozwala się upewnić, że nazwy zdefiniowane w klasie nie kolidują z nazwami zdefiniowanymi w klasach potomnych.

Klasa `ListInstance` definiuje metodę `__str__()`, dzięki czemu wywołanie funkcji `print()` na instancjach klas potomnych wypisze więcej informacji na ich temat, niż oferuje standardowy mechanizm Pythona. Poniższy listing prezentuje naszą klasę w działaniu, w sytuacji pojedynczego dziedziczenia (kod daje takie same wyniki w Pythonie 3.0 i 2.6):

```
>>> from lister import ListInstance
>>> class Spam(ListInstance):
...     def __init__(self):
...         self.data1 = 'food'
...
>>> x = Spam()
>>> print(x)
# Dziedziczy metodę __str__
<Instancja klasy Spam, adres 18931664:
    nazwa data1=food
>
```

Reprezentację tekstową instancji możemy również uzyskać za pomocą funkcji `str()`. Wypisanie wyniku na konsoli wykorzystuje standardowe formatowanie:

```
>>> str(x)
'<Instancja klasy Spam, adres 18931664:\n\tname data1=food\n>'
>>> x
# Nadal domyślna jest metoda __repr__
<__main__.Spam instance at 0x0120DFD0>
```

Klasa `ListInstance` może być przydatna w dowolnych klasach tworzonych przez użytkownika, nawet w przypadku klas posiadających jedną lub kilka klas nadzędnych. W takich przypadkach *wielokrotne dziedziczenie* okazuje się szczególnie użyteczne: dodając `ListInstance` do listy dziedziczenia w deklaracji klasy (wmieszanie), otrzymujemy dostęp do zdefiniowanej w niej metody `__str__()`, nie tracąc atrybutów dziedziczonych po pozostałych klasach. Właściwość tę demonstruje listing `testmixin.py`:

```
# Plik testmixin.py
from lister import *                                         # Import klas modułu lister

class Super:
    def __init__(self):
        self.data1 = 'spam'                                     # Tworzenie atrybutów instancji
    def ham(self):
        pass

class Sub(Super, ListInstance):                                # Wmieszanie metod ham() i __str__()
    def __init__(self):
        Super.__init__(self)
```

```

        self.data2 = 'eggs'           # Więcej atrybutów instancji
        self.data3 = 42
    def spam(self):
        pass

if __name__ == '__main__':
    X = Sub()
    print(X)                      # Wywołuje wmieszaną metodę __str__()

```

Klasa Sub dziedziczy po klasach Super i ListInstance, w efekcie czego stanowi połączenie własnych nazw i nazw zdefiniowanych w klasach nadzędnych. Po utworzeniu instancji klasy Sub i wywołaniu na niej funkcji print() automatycznie zostaje wywołana metoda __str__() wmieszana z klasą ListInstance (w tym przypadku wynik skryptu jest identyczny w przypadku Pythona 3.0 i 2.6, z wyjątkiem adresów metod):

```
C:\misc> C:\python30\python testmixin.py
<Instancja klasy Sub, adres 18922552:
    nazwa data1=spam
    nazwa data2=eggs
    nazwa data3=42
>
```

Klasa ListInstance działa z dowolną klasą potomną, ponieważ jej metody odwołują się do instancji tej klasy potomnej, z której wydobywają niezbędne informacje. W pewnym sensie klasy mieszane są odpowiednikiem modułów, czyli pakietów narzędzi, których można użyć w różnych kontekstach. Poniższy listing przedstawia przykład użycia modułu lister z inną klasą potomną, przy czym atrybuty instancji są zdefiniowane poza definicją klasy:

```
>>> import lister
>>> class C(lister.ListInstance): pass
...
>>> x = C()
>>> x.a = 1; x.b = 2; x.c = 3
>>> print(x)
<Instancja klasy C, adres 18922232:
    nazwa a=1
    nazwa b=2
    nazwa c=3
>
```

Oprócz wygody stosowania klasy mieszanej pozwalają na optymalizację utrzymania kodu, co jest ogólnie cechą programowania zorientowanego obiektowo. Jeśli jakiś czas po utworzeniu klasy zdecydujemy się na przykład zmodyfikować format ciągów znaków zwracanych z metody __str__() klasy ListInstance w taki sposób, aby zwracał również atrybuty klasy odziedziczone po klasach nadzędnych, nie będzie problemu. Dzięki temu, że metoda ta jest dziedziczona, wszystkie klasy dziedziczące po ListInstance automatycznie skorzystają z tej zmiany. Uznajmy, że wspomniany „jakis czas po utworzeniu klasy” właśnie minął i zajmijmy się tym zagadnieniem.

Wydobywanie atrybutów odziedziczonych z użyciem dir()

Nasz moduł lister wypisuje wyłącznie atrybuty instancji (to znaczy nazwy przypisane do samego obiektu instancji). Bardzo łatwo można jednak rozszerzyć ten mechanizm o wypisywanie wszystkich atrybutów dostępnych w instancji, zarówno własnych, jak i odziedziczonych po klasach nadzędnych. W tym celu wystarczy skorzystać z funkcji wbudowanej dir(), zamiast czytać ze słownika __dict__. Słownik __dict__ zawiera bowiem jedynie atrybuty instancji, natomiast funkcja dir() dodatkowo zwraca atrybuty odziedziczone (od Pythona 2.2 wzwyż).

Poniższy listing prezentuje odpowiednią modyfikację klasy `ListInstance`. Zmieniłem nazwę klasy, aby uprościć testowanie, ale w rzeczywistej sytuacji można by z powodzeniem zmodyfikować istniejącą klasę, dzięki czemu klasy dziedziczące mogłyby bez jakichkolwiek modyfikacji korzystać ze zmienionej logiki.

```
# Plik lister.py, ciąg dalszy class ListInherited:  
    """  
        Klasa ListInstance obsługuje formatowanie ciągów znaków zwracanych  
        z funkcji print() i str(). Mechanizm jest definiowany w metodzie specjalnej  
        __str__(), w naszym przypadku wypisuje nazwy i wartości atrybutów.  
        Argument self jest instancją klasy na najniższym poziomie dziedziczenia.  
        W przykładzie wykorzystujemy atrybuty pseudoprivałne __X w celu  
        uniknięcia konfliktów nazw z atrybutami klas z drzewa dziedziczenia.  
        Wykorzystujemy funkcję dir() do uzyskania listy atrybutów instancji  
        oraz atrybutów odziedziczonych. W Pythonie 3.0 uzyskamy większą liczbę  
        nazw w porównaniu z 2.6 z powodu różnic w drzewie dziedziczenia w nowym  
        modelu klas. Do uzyskania dostępu do atrybutów niedostępnych w słowniku  
        __dict__ należy posłużyć się funkcją getattr(). Nie należy w ten sposób  
        przeszukiwać metody __repr__(), ponieważ spowoduje to zapętlenie przy  
        próbie wypisania metod związanych!  
    """  
  
    def __str__(self):  
        return '<Instancja klasy %s, adres %s:\n%s>' % (  
            self.__class__._name_, # nazwa klasy  
            id(self), # adres  
            self.__attrnames()) # lista nazwa=wartość  
  
    def __attrnames(self):  
        result = ''  
        for attr in dir(self):  
            if attr[:2] == '__' and attr[-2:] == '__':  
                result += '\tname %s=>\n' % attr # Lista nazw atrybutów  
            else: # Pomijamy nazwy wewnętrzne  
                result += '\tname %s=%s\n' % (attr, getattr(self, attr))  
        return result
```

Warto zwrócić uwagę, że pomijamy nazwy typu `__X__`. Większość z takich nazw dotyczy atrybutów używanych w sposób niejawnym (wewnętrznie), przez co raczej nie będą interesujące w zastosowaniach, do których napisaliśmy naszą klasę. W tej wersji musimy też użyć funkcji `getattr()`, która korzysta z mechanizmu dziedziczenia, a więc pozwala uzyskać dostęp do nazw zdefiniowanych w klasach nadrzędnych, a niektóre z nazw zwracanych przez funkcję `dir()` nie są zapisane w instancji.

Aby sprawdzić w działaniu naszą nową klasę, zmodyfikujemy odpowiednio moduł `testmixin.py`:

```
class Sub(Super, ListInherited): # Wmieszana nazwa __str__
```

Wynik działania modułu będzie się różnił dla różnych wersji Pythona. W 2.6 uzyskamy wynik przedstawiony na poniższym listingu. Należy zwrócić uwagę na efekt działania mechanizmu zniekształcania nazw (wynik został przycięty, aby zajmował mniej miejsca).

```
C:\misc> c:\python26\python testmixin.py <Instancja klasy Sub, adres 18880832:  
nazwa __ListInherited__attrnames=<bound method Sub.__attrnames of <__main__  
↳.Sub instance at 0x01201940>>  
nazwa __doc__=<>  
nazwa __init__=<>  
nazwa __module__=<>  
nazwa __str__=<>  
nazwa data1=spam  
nazwa data2=eggs
```

```

nazwa data3=42
nazwa ham=<bound method Sub.ham of <__main__.Sub instance at 0x01201940>>
nazwa spam=<bound method Sub.spam of <__main__.Sub instance at 0x01201940>>
>

W Pythonie 3.0 skrypt wypisze więcej nazw, ponieważ wszystkie klasy „nowego typu” dziedziczą nazwy po klasie object (więcej szczegółów na ten temat przedstawię w rozdziale 31.). Sporo tych nazw jest dziedziczonych po klasie object, zatem w listingu pominąłem część z nich, ale polecam zapoznanie się z ich pełną listą.

C:\misc> c:\python30\python testmixin.py
<Instancja klasy Sub, adres 15427184:
    nazwa __ListInherited_attrnames=<bound method Sub.__attrnames of <__main__
    ↵.Sub object at 0x00EB6670>>
    nazwa __class__=<>
    nazwa __delattr__=<>
    nazwa __dict__=<>
    nazwa __doc__=<>
    nazwa __eq__=<>
    ...pominięta część wyniku...
    nazwa __repr__=<>
    nazwa __setattr__=<>
    nazwa __sizeof__=<>
    nazwa __str__=<>
    nazwa __subclasshook__=<>
    nazwa __weakref__=<>
    nazwa data1=spam
    nazwa data2=eggs
    nazwa data3=42
    nazwa ham=<bound method Sub.ham of <__main__.Sub object at 0x00EB6670>>
    nazwa spam=<bound method Sub.spam of <__main__.Sub object at 0x00EB6670>>
>

```

Jedna przestroga: w związku z tym, że wypisujemy również odziedziczone atrybuty, zamiast przeciążać metodę `__repr__()`, musimy przeciążać `__print__()`. W przeciwnym razie ten kod spowodowałby *zapętlanie*: wypisywanie wartości metody wywołuje metodę `__repr__()` klasy metody. Innymi słowy, jeśli metoda `__repr__()` zdefiniowana w klasie spróbuje wypisać metodę, spowoduje to ponowne wywołanie metody `__repr__()` tej klasy. Różnica jest subtelna, lecz istotna. Jeśli ktoś chciałby się przekonać, wystarczy zmienić nazwę metody `__str__` na `__repr__`. Jeśli ktoś jest zmuszony do wykorzystania metody `__repr__()` w takim kontekście, może uniknąć tego typu pułapek, kontrolując typ wypisywanych wartości atrybutów i porównując je ze stałą `types.MethodType`. Dzięki temu można pominać metody lub obsłużyć je w specjalny sposób, unikając cyklicznego wywołania metody `__repr__()`.

Wypisywanie atrybutów dla każdego obiektu w drzewie klas

Zaprogramujmy ostatnie już rozszerzenie klasy wypisującej atrybuty. W obecnej postaci lister nie informuje o tym, z której klasy w hierarchii dziedziczenia pochodzi dany argument. Jak widzieliśmy w przykładzie `classtree.py` z rozdziału 28., w kodzie mamy możliwość przechodzenia w hierarchii klas. Poniższy listing prezentuje klasę mieszaną wykorzystującą tę technikę do wyświetlania atrybutów pogrupowanych po klasach, w których zostały zdefiniowane. To w efekcie daje szkic drzewa dziedziczenia, wyświetlając atrybuty każdej z klas. Implementacja polega na odczytaniu atrybutu `__class__` instancji, wskazującego jej klasę, z której odczytywany jest atrybut `__bases__` zawierający listę klas nadrzędnych. Rekurencyjne przeglądanie listy `__bases__` każdej z tych klas pozwala przejrzeć całe drzewo, a przeglądanie listy `__dict__` każdej klasy daje wgląd w listę jej atrybutów.

```

# Plik lister.py, ciąg dalszy class ListTree:
"""
Klasa mieszana definiująca metodę __str__() służącą do zwracania
tekstowej reprezentacji instancji w postaci drzewa dziedziczenia
i atrybutów definiowanych na każdym poziomie. Wywołanie tej metody jest
automatyczne w efekcie wywołania na instancji funkcji print() lub
str(). Klasa wykorzystuje nazwy pseudoprzywatne __X w celu uniknięcia
konfliktu nazw w klasach potomnych. Do rekurencyjnego przeglądania
klas nadrzędnych wykorzystuje wyrażenia generatorów oraz str.format()
do czytelniego formatowania wyników.
"""

def __str__(self):
    self.__visited = {}
    return '<Instancja klasy {0}, adres {1}:\n{2}{3}>'.format(
        self.__class__._name__,
        id(self),
        self.__attrnames(self, 0),
        self.__listclass(self.__class__, 4))

def __listclass(self, aClass, indent):
    dots = '.' * indent
    if aClass in self.__visited:
        return '\n{0}<Klasa {1}:, adres {2}: (patrz wyżej)>\n'.format(
            dots,
            aClass._name__,
            id(aClass))
    else:
        self.__visited[aClass] = True
        genabove = (self.__listclass(c, indent+4) for c in aClass.__bases__)
        return '\n{0}<Klasa {1}, adres {2}:\n{3}{4}>\n'.format(
            dots,
            aClass._name__,
            id(aClass),
            self.__attrnames(aClass, indent),
            ''.join(genabove),
            dots)

def __attrnames(self, obj, indent):
    spaces = ' ' * (indent + 4)
    result = ''
    for attr in sorted(obj.__dict__):
        if attr.startswith('__') and attr.endswith('__'):
            result += spaces + '{0}=>\n'.format(attr)
        else:
            result += spaces + '{0}={1}\n'.format(attr, getattr(obj, attr))
    return result

```

Warto zwrócić uwagę na *wyrażenie generatora* wykorzystujące rekurencyjne wywołania klas nadrzędnych poprzez zagnieżdżone wywołania metody `join()` ciągu znaków. W tym przypadku wykorzystaliśmy formatowanie ciągów znaków wprowadzone w Pythonie 2.6 i 3.0 zamiast użycia tradycyjnego operatora podstawiania `%`. Dzięki temu operacje podstawiania są bardziej zrozumiałe, ponieważ przy dużej liczbie podstawień, jak w naszym przykładzie, kod staje się znacznie czytelniejszy. Innymi słowy, ten sam efekt można uzyskać, wywołując jedno z poniższych wyrażeń — my wybraliśmy tę drugą formę:

```

return '<Instancja klasy %s, adres %s:\n%s%s>' % (...)      # Wyrażenie
return '<Instancja klasy {0}, adres {1}:\n{2}{3}>'.format(...) # Metoda

```

W celu przetestowania nowej klasy zmodyfikujmy nasz moduł testujący: `class Sub(Super,`

Po wywołaniu modułu w Pythonie 2.6 zobaczymy następujący wynik:

```
C:\misc> c:\python26\python testmixin.py <Instancja klasy Sub, adres 23421736:
__ListTree__visited={}
data1=spam
data2=eggs
data3=42

....<Klasa Sub, adres 23368496:
__doc__=<>
__init__=<>
__module__=<>
spam=<unbound method Sub.spam>

.....<Klasa Super, adres 23368544:
__doc__=<>
__init__=<>
__module__=<>
ham=<unbound method Super.ham>
.....>

.....<Klasa ListTree, adres 23367968:
__ListTree__attrnames=<unbound method ListTree.__attrnames>
__ListTree__listclass=<unbound method ListTree.__listclass>
__doc__=<>
__module__=<>
__str__=<>
.....>
....>
>
```

W tym wyniku widać dokładnie *niezwiązane* metody, ponieważ odczytujemy je z klas, nie z instancji. Widzimy również efekt zniekształcania nazw na atrybucie `__visited`, dzięki czemu istnieje większa szansa, że nazwa ta nie zostanie nadpisana w klasach potomnych.

Po uruchomieniu tego modułu w Pythonie 3.0 ponownie uzyskamy znacznie większą liczbę atrybutów i klas nadzędnych. Jak pamiętamy, w Pythonie 3.0 metody niezwiązane są zwykłymi *funkcjami*. Część wyniku pominąłem w celu zaoszczędzenia miejsca; warto uruchomić ten kod samodzielnie, aby przeanalizować efekt działania w całości.

```
C:\misc> c:\python30\python testmixin.py <Instancja klasy Sub, adres 15427600:
__ListTree__visited={}
data1=spam
data2=eggs
data3=42

....<Klasa Sub, adres 15002232:
__doc__=<>
__init__=<>
__module__=<>
spam=<function spam at 0x00EB98A0>

.....<Klasa Super, adres 15014760:
__dict__=<>
__doc__=<>
__init__=<>
__module__=<>
__weakref__=<>
ham=<function ham at 0x00EB96F0>
```

```

.....<Klasa object, adres 505114624:
    __class__=<>
    __delattr__=<>
    __doc__=<>
    __eq__=<>
    ...pominietý fragment wyniku...
    __repr__=<>
    __setattr__=<>
    __sizeof__=<>
    __str__=<>
    __subclasshook__=<>
.....
.....
....>
....>

.....<Klasa ListTree, adres 15014312:
    _ListTree__attrnames=<function __attrnames at 0x00EB9B70>
    _ListTree__listclass=<function __listclass at 0x00EB9B28>
    __dict__=<>
    __doc__=<>
    __module__=<>
    __str__=<>
    __weakref__=<>

<Klasa object:, adres 505114624: (patrz wyżej)>
.....
....>
....>
>
```

W implementacji wykorzystaliśmy kod zapobiegający wielokrotnemu wypisaniu informacji na temat tej samej klasy, dlatego użyliśmy identyfikatora obiektu, który posłużył nam jako klucz w słowniku. W rozdziale 24. użyliśmy słownika w podobnej roli — zabezpieczenia przed powtórzeniami: obiekty klas mogą być kluczami w słowniku, dzięki czemu kod jest znacznie prostszy. Podobny efekt dałoby uzycie zbiorów (`set`):

W tym przypadku również staraliśmy się unikać obiektów wewnętrznych, pomijając wartości dla nazw typu `_X_`. Gdyby pominąć warunek sprawdzający te nazwy, zobaczylibyśmy w wyniku pełne wartości tych atrybutów. Poniższy listing przedstawia wynik działania skryptu w Pythonie 2.6 po usunięciu warunku pomijającego nazwy `_X_` (wynik jest znacznie większy, a w Pythonie 3.0 efekt byłby jeszcze gorszy):

```
C:\misc> c:\python26\python testmixin.py ...pominietý fragment wyniku...

.....<Klasa ListTree, adres 18849632:
    _ListTree__attrnames=<unbound method ListTree.__attrnames>
    _ListTree__listclass=<unbound method ListTree.__listclass>
    __doc__=
    Klasa mieszana definiująca metodę __str__() służącą do zwracania tekstowej reprezentacji egzemplarza w postaci drzewa dziedziczenia i atrybutów definiowanych na każdym poziomie. Wywołanie tej metody jest automatyczne w efekcie wywołania na egzemplarzu funkcji print() lub str(). Klasa wykorzystuje nazwy pseudoprivatne _X w celu uniknięcia konfliktu nazw w klasach potomnych. Do rekurencyjnego przeglądania klas nadzędnych wykorzystuje wyrażenia generatorów oraz str.format() do czytelnego formatowania wyników.

    __module__=lister
    __str__=<unbound method ListTree.__str__>
.....
....>
....>
>
```

Ciekawe efekty daje wzmieszanie metod naszej klasy razem z bardziej rozbudowaną klasą, jak `Button` z modułu `tkinter`. Należy pamiętać, aby klasa `ListTree` była zadeklarowana jako pierwsza na liście klas nadzędnych, by została użyta zdefiniowana w niej metoda `__str__()`. Klasa `Button` również posiada własną implementację metody `__str__()`, więc gdyby to ona znalazła się po lewej stronie, zostałaby użyta ta implementacja. Poniższy listing generuje bardzo duży wynik (18 tysięcy znaków), zatem Czytelnikom zainteresowanym pełnym wynikiem sugeruję samodzielne wywołanie tego kodu (użytkownicy Pythona 2.6 zamiast nazwy `tkinter` powinni użyć `Tkinter`).

```
>>> from lister import ListTree
>>> from tkinter import Button
>>> class MyButton(ListTree, Button): pass      # Obie klasy definiują metodę __str__()
>>> B = MyButton(text='mielonka')
>>> open('savetree.txt', 'w').write(str(B))      # Zapis wyniku do pliku
18247
>>> print(B)                                     # Wypisanie wyniku
<Instancja klasy MyButton, adres 23423696:
    _ListTree_visited={}
    _name=23423696
    _tclCommands=[]
    ...pominęty duży fragment wyniku...
>
```

Oczywiście w tym temacie można wywalczyć znacznie więcej (na przykład od razu przychodzi do głowy graficzna reprezentacja drzewa dziedziczenia z użyciem jakiejś biblioteki GUI), ale dalszą rozbudowę klas `Lister` pozostawię jako zadanie domowe. W dalszej części książki uzupełnimy nieco ten przykład o wypisywanie w nawiasach nazw klas nadzędnych.

Celem tego ćwiczenia było wykazanie, że w programowaniu zorientowanym obiektowo chodzi przede wszystkim o ponowne użycie kodu, a klasy mieszane są doskonałą ilustracją tej właściwości. Jak większość koncepcji programowania obiektowego, wielokrotne dziedziczenie może być użytecznym narzędziem, jeśli jest użyte prawidłowo. W praktyce jednak należy pamiętać, że to zaawansowana cecha języka i nieuważne lub nieuzasadnione jej stosowanie może szybko doprowadzić do niepotrzebnej komplikacji. Wróćmy do tego problemu na końcu następnego rozdziału w ramach przestrogi dla programisty. W tym rozdziale omówimy też nowy model klas w Pythonie, w którym zmieniono kolejność wyszukiwania nazw w przypadku wielokrotnego dziedziczenia.

Klasy są obiektami — uniwersalne fabryki obiektów

W niektórych sytuacjach jesteśmy zmuszeni tworzyć obiekty na żądanie i nie da się z góry przewidzieć, jakiej klasy mają być te obiekty. W takim przypadku przydaje się wzorzec projektowy znany pod nazwą fabryki. Dzięki dużej elastyczności Pythona fabryki mogą przyjmować różne formy.

Ponieważ klasy są obiektami, łatwo jest je przekazywać w programach czy przechowywać w strukturach danych. Można również przekazać klasy do funkcji generujących dowolne rodzaje obiektów. Takie funkcje nazywane są czasami w kręgach projektowania zorientowanego obiektowo *fabrykami* (ang. *factory*). Są one dużym wyzwaniem w językach z silnymi typami, takich jak C++, jednak w Pythonie są trywialne do zaimplementowania. Funkcja `apply` oraz nowsza



Obsługa slotów: Klasы `ListInstance` i `ListTree` wykorzystują słowniki atrybutów, przez co nie obsługują atrybutów zapisanych w `slotach`. Sloty to stosunkowo nowa i dość rzadko stosowana opcja w programowaniu obiektowym w Pythonie, której przyjrzymy się bliżej w następnym rozdziale. Technika ta wykorzystuje deklarację atrybutów klas w atrzybucie klasy `__slots__`. Gdyby na przykład w module `test-mixin.py` przypisać `__slots__ = ['data1']` w klasie `Super` oraz `__slots__ = ['data3']` w klasie `Sub`, wspomniane dwie klasy modułu `lister` wypiszą jedynie atrybut `data1` i `data3`, ale jako atrybuty, odpowiednio, klas `Super` i `Sub` z użyciem specjalnego formatu wartości (z technicznego punktu widzenia są one deskryptorami na poziomie klas).

W celu lepszej obsługi mechanizmu slotów w klasach modułu `lister` należy zmodyfikować pętle przeglądające słownik `__dict__`, aby również przeglądały `__slots__` (wykorzystując kod prezentowany w kolejnym rozdziale), a do odczytu wykorzystywać funkcję `getattr()` zamiast wartości zapisanych w słowniku `__dict__` (`ListTree` już wykorzystuje tę technikę). Instancje dziedziczą `__slots__` po klasie najniższego poziomu, należy więc dodatkowo zadbać o to, aby odpowiednio obsługiwać sytuację, gdy atrybut `__slots__` występuje w kilku klasach nadzędnych (`ListTree` w bieżącej wersji już wyświetla je jako atrybuty klas). Klasa `ListInherited` nie wymaga dodatkowych zmian, ponieważ funkcja `dir()` automatycznie obsługuje nazwy zdefiniowane w atrybutach `__dict__` oraz `__slots__`.

Alternatywne podejście mogłoby polegać na obsłudze slotów w aktualny sposób, z pominięciem obsługi rzadkich, zaawansowanych cech języka. Sloty i „zwykłe” atrybuty instancji są inaczej obsługiwane przez Pythona. Więcej szczegółów na temat użycia slotów poznamy w kolejnym rozdziale, celowo pominąłem ich analizę, aby nie uprzedzać faktów (pomijając tę uwagę). Nie jest to podejście zalecane w rzeczywistym programowaniu, ale dopuszczalne na potrzeby książki.

składnia dla niej alternatywna — składnia wywołania omówiona w rozdziale 18., pozwala w jednym kroku wywołać dowolną klasę z dowolną liczbą argumentów konstruktora w celu utworzenia dowolnej instancji.³

```
def factory(aClass, *args):                      # Krotka argumentów o zmiennej liczbie
    return aClass(*args)                          # Wywołanie konstruktora klasy aClass 1 (lub apply w 2.6)

class Spam:
    def doit(self, message):
        print message

class Person:
    def __init__(self, name, job):
        self.name = name
        self.job = job

object1 = factory(Spam)                         # Utworzenie obiektu Spam
object2 = factory(Person, "Guido", "guru")      # Utworzenie obiektu Person
```

³ Tak naprawdę taka składnia może posłużyć do wywołania dowolnego obiektu wywoływalnego, w tym funkcje, klasy oraz metody. W taki sposób funkcja `factory` może także wykonać dowolny obiekt wywoływalny, nie tylko klasę (pomimo nazwy argumentu). W rozdziale 18. poznaliśmy też dostępną w Pythonie 2.6 alternatywę dla wywołania `aClass(*args)`: funkcję wbudowaną `apply(aClass, args)`, która w Pythonie 3.0 została usunięta z powodu nadmiarowości i ograniczeń.

W powyższym kodzie definiujemy funkcję generatora obiektów o nazwie `factory`. Oczekuje ona przekazania obiektu klasy (wystarczy dowolna klasa) wraz z jednym lub większą liczbą argumentów dla konstruktora klasy. Ta funkcja do wywołania konstruktora klasy używa specjalnej składni wykorzystującej dowolną listę parametrów wywołania (`*args`).

Pozostała część przykładu definiuje po prostu dwie klasy i generuje ich instancje, przekazując je do funkcji `factory`. Jest to jedyna funkcja fabryki, jaką kiedykolwiek będziemy musieli pisać w Pythonie — działa ona dla każdej klasy i dowolnych argumentów konstruktora.

Jedynym możliwym ulepszeniem, na jakie warto zwrócić uwagę, jest to, że by obsługiwać w wywołaniu konstruktora argumenty będące słowami kluczowymi, fabryka może zebrać je za pomocą argumentu `**args` i przekazać jako trzeci argument do funkcji `apply`.

```
def factory(aClass, *args, **kwargs):      # + słownik argumentów ze słowami kluczowymi
    return apply(aClass, args, kwargs)       # Wywołanie aClass
```

Powinniśmy już wiedzieć, że w Pythonie wszystko jest obiektem, w tym klasy, które w językach takich, jak C++ są tylko danymi wejściowymi kompilatora. Jak jednak wspomniano na początku tej części książki, jedynie obiekty pochodzące z klas są w Pythonie obiektami z dziedziny programowania zorientowanego obiektowo.

Do czego służą fabryki?

Do czego zatem mogą nam się przydać fabryki (poza dostarczeniem wymówki do zilustrowania obiektów klas w niniejszej książce)? Niestety, trudno jest pokazać zastosowanie tego wzorca projektowego bez zamieszczenia większej ilości kodu, niż mamy na to miejsce. Ogólnie rzecz biorąc, fabryki pozwalają na odizolowanie kodu od szczegółów konfigurowanej dynamicznie konstrukcji obiektu.

Przypomnijmy sobie przykład z klasą `processor`, zaprezentowany z abstrakcyjnego punktu widzenia w rozdziale 25. oraz w tym rozdziale jako przykład związku typu „ma”. Przyjmował on obiekty typu `reader` oraz `writer` do przetwarzania dowolnych strumieni danych. Oryginalna wersja tego przykładu ręcznie przekazywała instancje wyspecjalizowanych klas, takich jak `FileWriter` oraz `SocketReader`, w celu dostosowania przetwarzanych strumieni danych do własnych potrzeb. Później przekazywaliśmy zapisane na stałe w kodzie obiekty pliku, strumienia oraz formatera. W bardziej dynamicznym scenariuszu do konfigurowania strumienia mogłyby posłużyć narzędzia zewnętrzne, takie jak pliki konfiguracyjne czy graficzne interfejsy użytkownika.

W takim dynamicznym świecie być może nie uda nam się zapisać obiektów interfejsów strumieni na stałe w kodzie skryptu, jednak zamiast tego możemy je utworzyć w czasie wykonywania zgodnie z zawartością pliku konfiguracyjnego.

Plik może na przykład podawać po prostu łańcuch znaków nazwy klasy strumienia, jaka ma być zaimportowana z modułu, wraz z opcjonalnym argumentem wywołania konstruktora. Mogą się tutaj przydać mechanizmy fabrykujące, ponieważ pozwolą nam pobrać oraz przekazać klasy, które nie są zapisane na stałe w naszym programie. Klasy te mogą nawet wcale nie istnieć w momencie pisania naszego kodu.

```
classname = ...odczytane z pliku konfiguracyjnego...
classarg = ...odczytane z pliku konfiguracyjnego...
import streamtypes                                # Kod można dostosować do własnych potrzeb
```

```
aclasse = getattr(streamtypes, classname)      # Pobrane z modułu
reader = factory(aclasse, classarg)            # Lub aclasse(classarg)
processor(reader, ...)
```

W powyższym kodzie wbudowana funkcja `getattr` jest ponownie wykorzystana do pobrania atrybutu modułu po podaniu łańcucha znaków nazwy (przypomina to kod `obiekt.atrybut`, ale `atrybut` jest tutaj łańcuchem znaków). Ponieważ ten fragment kodu zakłada istnienie jednego argumentu konstruktora, nie potrzebuje właściwie `factory` czy `apply` — moglibyśmy utworzyć instancję za pomocą kodu `aclasse(classarg)`. Może się to jednak przydać w obecności nieznanej listy argumentów, a ogólny wzorzec tworzenia kodu fabryki może poprawić elastyczność kodu.

Inne zagadnienia związane z projektowaniem

W tym rozdziale poznaliśmy wiele podstawowych zagadnień programowania zorientowanego obiektowo w Pythonie: dziedziczenie, kompozycję, delegację, wielokrotne dziedziczenie, metody związane i fabryki. Jednak jeśli chodzi o zagadnienia związane z projektowaniem zorientowanym obiektowo, to ledwie wierzchołek góry lodowej. Innych zagadnień przydatnych w projektowaniu warto szukać w pozostałych rozdziałach książki:

- klasy abstrakcyjne (rozdział 28.),
- dekoratory (rozdziały 31. i 38.),
- klasy typów (rozdział 31.),
- metody statyczne i metody klas (rozdział 31.),
- atrybuty zarządzane (rozdział 37.),
- metaklasy (rozdziały 31. i 39.).

Jeśli jednak chodzi o zagadnienia ogólne ściślej związane z samym projektowaniem zorientowanym obiektowo, polecam literaturę poświęconą tej tematyce. Wzorce projektowe są bardzo ważnym elementem programowania zorientowanego obiektowo i w Pythonie ich stosowanie bywa często bardziej naturalne niż w innych językach, lecz nie są one dla Pythona specyficzne.

Podsumowanie rozdziału

W niniejszym rozdziale zamieściliśmy przykłady często stosowanych sposobów wykorzystywania oraz łączenia klas w celu zoptymalizowania możliwości ich ponownego użycia i faktoryzacji. Często jest to uznawane za zagadnienia z zakresu projektowania, które są niezależne od języka programowania (choć Python może uprościć ich implementację). Omawialiśmy *delegację* (opakowanie obiektów w klasy pośredniczące), *kompozycję* (kontrolowanie obiektów osadzonych), *dziedziczenie* (nabywanie zachowania z innych klas), a także pewne bardziej ezoteryczne koncepcje, takie jak dziedziczenie wielokrotne, metody z wiązaniem oraz fabryki.

W kolejnym rozdziale zakończymy naszą pracę z klasami i programowaniem zorientowanym obiektowo, badając bardziej zaawansowane zagadnienia związane z klasami. Część tego materiału może być bardziej interesująca dla twórców narzędzi niż programistów aplikacji, jednak mimo to zasługuje na przejrzenie przez osoby, które chcą się zajmować programowaniem zorientowanym obiektowo w Pythonie. Najpierw jednak pora na krótki quiz.

Sprawdź swoją wiedzę — quiz

1. Czym jest dziedziczenie wielokrotne?
2. Czym jest delegacja?
3. Czym jest kompozycja?
4. Czym są metody związane?
5. Do czego wykorzystywane są atrybuty pseudoprzywatne?

Sprawdź swoją wiedzę — odpowiedzi

1. Dziedziczenie wielokrotne ma miejsce, kiedy klasa dziedziczy po więcej niż jednej klasie nadzędnej. Przydaje się to do mieszania ze sobą kilku pakietów kodu opartego na klasach. Przy wyszukiwaniu atrybutów obowiązuje kolejność przeszukiwania nazw od lewej do prawej względem kolejności deklaracji dziedziczenia.
2. Delegacja obejmuje opakowanie obiektu w klasę pośredniczącą, dodającą dodatkowe zachowanie i przekazującą inne operacje do opakowanego obiektu. Klasa pośrednicząca (proxy) zachowuje interfejs opakowanego obiektu.
3. Kompozycja jest techniką, w której klasa kontrolera osadza i kieruje kilkoma obiektemi, a także udostępnia własny interfejs. Jest to sposób tworzenia większych struktur za pomocą klas.
4. Metody z wiązaniem łączą instancję oraz funkcję metody. Można je wywoływać bez przekazywania obiektu instancji w sposób jawny, ponieważ oryginalna instancja nadal jest dostępna.
5. Atrybuty pseudoprzywatne (`_x`, czyli takie, których nazwy rozpoczynają się dwoma znakami podkreśnika) są wykorzystywane do ograniczania zasięgu nazw do klasy, w której są zdefiniowane. Dotyczy to atrybutów klasy, jak zdefiniowane w niej metody, oraz atrybutów instancji `self`, przypisanych w ramach klasy. Nazwy te są przekształcane w taki sposób, aby zawierały przedrostek nazwy klasy, co powoduje, że stają się unikalne.

Zaawansowane zagadnienia związane z klasami

Niniejszy rozdział kończy nasze omówienie programowania zorientowanego obiektowo, prezentując kilka bardziej zaawansowanych zagadnień związanych z klasami. Omówimy tworzenie klas podstawowych dla typów wbudowanych, atrybuty pseudoprivatne, klasy w nowym stylu, metody statyczne, dekoratory funkcji i wiele innych.

Jak widzieliśmy, model programowania zorientowanego obiektowo w Pythonie jest bardzo prosty, a część zagadnień zaprezentowanych w niniejszym rozdziale jest na tyle zaawansowana i opcjonalna, że możemy nie spotkać się z nimi zbyt często w naszej karierze twórcy aplikacji Pythona. W trosce o kompletność materiału zakończymy nasze omawianie klas krótkim spojrzeniem na bardziej skomplikowane narzędzia przeznaczone do zaawansowanego programowania zorientowanego obiektowo.

Ponieważ jest to ostatni rozdział tej części książki, jak zawsze zakończymy go podrozdziałem dotyczącym pułapek związanych z klasami, a także częścią z ćwiczeniami dotyczącymi omawianych w tej części książki zagadnień. Zachęcam do wykonania tych ćwiczeń w celu utrwalenia wiadomości dotyczących klas. Sugeruję również pracę lub przestudiowanie większych projektów opartych na programowaniu zorientowanym obiektowo w Pythonie w celu uzupełnienia podanych w książce informacji. Jak zawsze w przypadku informatyki zalety programowania zorientowanego obiektowo stają się widoczne wraz z wykonywaniem praktycznych zadań.



Uwaga na temat treści: Ten rozdział zawiera opisy bardziej zaawansowanych zagadnień, jednak część z nich jest zbyt skomplikowana, aby je tu solidnie omówić. Takie tematy, jak właściwości, deskryptory, dekoratory i metaklasy są tu ledwie wspomniane, a ich pełna analiza znalazła się w końcowej części książki.

Rozszerzanie typów wbudowanych

Poza implementowaniem nowych obiektów klasy są czasami wykorzystywane do rozszerzania funkcjonalności typów wbudowanych Pythona w celu dodania obsługi bardziej egzotycznych struktur danych. By na przykład dodać do list metody wstawiania oraz usuwania kolejek, możemy napisać kod klas opakowujących (osadzających) obiekty listy i eksportować metody

wstawiania oraz usuwania przetwarzające listę w specjalny sposób, podobnie jak technika delegacji omówiona w rozdziale 30. Od Pythona 2.2 można również wykorzystać dziedziczenie do specjalizowania typów wbudowanych. W kolejnych dwóch podrozdziałach zaprezentujemy działanie obu tych technik.

Rozszerzanie typów za pomocą osadzania

Przypomnijmy sobie funkcje działające na zbiorach, które napisaliśmy w rozdziałach 16. i 18. książki. Oto jak wyglądają, kiedy się je przywróci do życia w postaci klas Pythona. Poniższy przykład (*setwrapper.py*) implementuje nowy typ obiektu zbioru, przenosząc część funkcji zbiorów do metod i dodając pewne podstawowe metody przeciążania operatorów. Klasa ta w dużej mierze opakowuje po prostu listę Pythona za pomocą dodatkowych operacji na zbiorach. Ponieważ jest klasą, obsługuje również wiele instancji oraz możliwość dostosowania do własnych potrzeb za pomocą dziedziczenia w klasach podrzędnych.

W odróżnieniu od funkcji prezentowanych wcześniej, do których były przekazywane listy wartości, klasy użyte w tym przykładzie tworzą obiekty typu zbiorowego o zadanych z góry danych i zachowaniu.

```
class Set:
    def __init__(self, value = []):
        self.data = []
        self.concat(value)

    def intersect(self, other):
        res = []
        for x in self.data:
            if x in other:
                res.append(x)
        return Set(res)

    def union(self, other):
        res = self.data[:]
        for x in other:
            if not x in res:
                res.append(x)
        return Set(res)

    def concat(self, value):
        for x in value:
            if not x in self.data:
                self.data.append(x)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, key):
        return self.data[key]

    def __and__(self, other):
        return self.intersect(other)

    def __or__(self, other):
        return self.union(other)

    def __repr__(self):
        return 'Zbiór:' + repr(self.data)
```

Konstruktor
Zarządza listą
other to dowolna sekwencja
self to podmiot
Wybór wspólnych elementów
Zwrócenie nowej instancji Set
other to dowolna sekwencja
Kopia mojej listy
Dodanie elementów w other
value: lista, Set...
Usuwa duplikaty
len(self)
self[i]
self & other
self | other
Wyświetlenie za pomocą print

Powyższa klasa jest używana jak każda inna: tworzymy instancję, wywołujemy metody i stosujemy zdefiniowane operatory.

```

x = Set([1, 3, 5, 7])
print(x.union(Set([1, 4, 7])))
# Wypisuje Set:[1, 3, 5, 7, 4]
print(x | Set([1, 4, 6]))
# Wypisuje Set:[1, 3, 5, 7, 4, 6]

```

Przeciążenie indeksowania pozwala instancjom naszej klasy `Set` na udawanie prawdziwych list. Ponieważ nad tą klasą i jej rozszerzeniem będziemy pracować w ćwiczeniu na końcu rozdziału, nie powiem nic więcej na temat kodu aż do dodatku B.

Rozszerzanie typów za pomocą klas podrzędnych

Od Pythona 2.2 wszystkie typy wbudowane można bezpośrednio rozszerzać za pomocą klas podrzędnych. Funkcje konwersji typów, takie jak `list`, `str`, `dict` oraz `tuple`, stały się nazwami typów wbudowanych. Choć jest to niewidoczne dla naszego skryptu, wywołanie konwersji typu (na przykład `list('mielonka')`) jest teraz tak naprawdę wywołaniem konstruktora obiektu typu.

Zmiana ta pozwala rozszerzać lub dostosowywać do własnych potrzeb zachowanie typów wbudowanych za pomocą zdefiniowanych przez użytkownika instrukcji `class`. By dostosować typ wbudowany do własnych potrzeb, wystarczy utworzyć klasę podrzędną z wykorzystaniem nowej nazwy typu. Instancje klas nadrzędnych typów można wykorzystywać w każdym miejscu, w którym mógłby się pojawić oryginalny typ wbudowany. Założymy na przykład, że mamy problem z przyzwyczajeniem się do faktu, iż wartości przesunięcia w indeksach list Pythona zaczynają się od 0, a nie od 1. Nie ma się co martwić — zawsze możemy utworzyć własną klasę dostosowującą tę właściwość list do naszych upodobań. Plik `typesubclass.py` pokazuje, jak można to zrobić.

```

# Klasa podrzędna wbudowanego typu (klasy) listy
# Odwzorowanie 1..N na 0..N-1; wywołanie z powrotem wbudowanej wersji

class MyList(list):
    def __getitem__(self, offset):
        print('indeksowanie %s w pozycji %s' % (self, offset))
        return list.__getitem__(self, offset - 1)

    if __name__ == '__main__':
        print(list('abc'))
        x = MyList('abc')           # Metoda __init__ odziedziczona po liście
        print(x)                   # Metoda __repr__ odziedziczona po liście

        print(x[1])                # MyList.__getitem__
        print(x[3])                # Dostosowuje metodę klasy nadrzędnej listy

        x.append('mielonka'); print x  # Atrybuty z klasy nadrzędnej listy
        x.reverse(); print x

```

W powyższym pliku klasa podrzędna `MyList` rozszerza metodę indeksowania listy wbudowanej `__getitem__` jedynie w taki sposób, by odwzorować indeksy od 1 do N na wymagane 0 do N-1. Tak naprawdę zmniejsza wartość otrzymanego indeksu i wywołuje z powrotem wersję indeksowania z klasy nadrzędnej, jednak w zupełności wystarczy to do wykonania tego zadania.

```

% python typesubclass.py
['a', 'b', 'c']
['a', 'b', 'c']
(indeksowanie ['a', 'b', 'c'] w pozycji 1)
a
(indeksowanie ['a', 'b', 'c'] w pozycji 3)

```

```
c
['a', 'b', 'c', 'mielonka']
['mielonka', 'c', 'b', 'a']
```

Powyższe dane wyjściowe obejmują również tekst śledzenia, który klasa wyświetla przy indeksowaniu. Czy taka modyfikacja indeksowania jest dobrym pomysłem to inna sprawa — użytkowników klasy `MyList` może bardzo zdziwić takie odejście od zachowania sekwencji Pythona. To, że możemy dostosować typy wbudowane w ten sposób, daje nam jednak narzędzie o dużych możliwościach.

Powyższy wzorzec kodu umożliwia nam na przykład alternatywne sposoby tworzenia zbiorów, które mogą być klasą podrzędną wbudowanego typu list zamiast samodzielna klasą zarządzającą osadzonym obiektem listy. Jak dowiedzieliśmy się w rozdziale 5., Python oferuje wbudowany typ zbiorowy oraz składnię definiowania literałów i złożień, pozwalającą na tworzenie nowych zbiorów. Jednak samodzielna implementacja klasy implementującej zbiory to nadal doskonały sposób na nauczenie się technik tworzenia klas potomnych.

Poniższa klasa utworzona w pliku `set subclass.py` dostosowuje listy w taki sposób, by dodać do nich jedynie metody oraz operatory powiązane z przetwarzaniem zbiorów. Ponieważ pozostałe zachowanie dziedziczone jest po wbudowanej klasie nadrzędnej list, umożliwia nam to utworzenie krótszej i prostszej alternatywy.

```
class Set(list):
    def __init__(self, value = []):
        list.__init__([])
        self.concat(value)
                                # Konstruktor
                                # Dostosowuje listę do własnych potrzeb
                                # Kopiuje zmienne wartości domyślne

    def intersect(self, other):
        res = []
        for x in self:
            if x in other:
                res.append(x)
        return Set(res)
                                # other to dowolna sekwencja
                                # self to podmiot
                                # Wybór wspólnych elementów
                                # Zwrócenie nowej instancji Set

    def union(self, other):
        res = Set(self)
        res.concat(other)
        return res
                                # other to dowolna sekwencja
                                # Kopia mojej listy

    def concat(self, value):
        for x in value:
            if not x in self:
                self.append(x)
                                # value: lista, Set...
                                # Usuwa duplikaty

    def __and__(self, other): return self.intersect(other)
    def __or__(self, other): return self.union(other)
    def __repr__(self):     return 'Zbiór:' + list.__repr__(self)

if __name__ == '__main__':
    x = Set([1,3,5,7])
    y = Set([2,1,4,5,6])
    print(x, y, len(x))
    print(x.intersect(y), y.union(x))
    print(x & y, x | y)
    x.reverse(); print(x)
```

Poniżej znajdują się dane wyjściowe kodu testującego znajdującego się na końcu pliku. Ponieważ tworzenie klas podrzędnych dla typów wbudowanych jest opcją zaawansowaną, pominę dalsze szczegóły, jednak zapraszam do prześledzenia wyników w celu przestudiowania tego zachowania kodu.

```
% python setsubclass.py
Zbiór:[1, 3, 5, 7], Zbiór:[2, 1, 4, 5, 6], 4
Zbiór:[1, 5], Zbiór:[2, 1, 4, 5, 6, 3, 7]
Zbiór:[1, 5], Zbiór:[1, 3, 5, 7, 2, 4, 6]
Zbiór:[7, 5, 3, 1]
```

Istnieją bardziej wydajne sposoby implementowania zbiorów w Pythonie za pomocą słowników, zastępujące liniowe przeszukiwanie w pokazanych wyżej implementacjach zbiorów operacją indeksowania słowników (mieszania), co działa o wiele szybciej. Więcej informacji na ten temat można znaleźć w książce *Programming Python*. Osoby zainteresowane zbiorami mogą również przyjrzeć się wbudowanemu typowi obiektu `set`, który omawialiśmy w rozdziale 5. Typ ten udostępnia działania na zbiorach jako narzędzi wbudowane. Implementacje zbiorów są zabawnym eksperymentem, jednak nie są one w dzisiejszym Pythonie wymagane.

Kolejny przykład klas podrzędnych dla typów można zobaczyć na podstawie nowego typu `bool` w Pythonie 2.3 oraz późniejszej wersji. Jak wspomnieliśmy wcześniej, `bool` jest klasą podrzędną typu `int` mającą dwie instancje (`True` oraz `False`), które zachowują się jak liczby całkowite 1 oraz 0, jednak dziedziczą własne metody reprezentacjiłańcuchów znaków, które wyświetlają ich nazwy.

Klasy w nowym stylu

W Pythonie 2.2 wprowadzono nowy rodzaj klas, zwany klasami „w nowym stylu”. Klassy omawiane dotychczas w książce są nazywane „klasycznymi”, kiedy porównuje się je z nowym typem. W Pythonie 3.0 migracja do klas w nowym stylu została zakończona, ale w przypadku wersji Pythona z serii 2.X nadal mamy do czynienia z podziałem na klasyczny model klas i model klas w nowym stylu.

- Od Pythona 3.0 wszystkie klasy są automatycznie tworzone jako klasy w nowym stylu, niezależnie od tego, czy dziedziczą po klasie `object`, czy nie. Po prostu wszystkie klasy dziedziczą po klasie `object`, nawet jeśli nie posiadają jawniej deklaracji takiego dziedziczenia, przez co wszystkie obiekty są instancjami klasy `object`.
- W Pythonie 2.6 i wcześniejszych klas w nowym stylu muszą jawnie dziedziczyć po klasie `object` (lub po innym typie wbudowanym).

Dzięki temu, że w 3.0 wszystkie klasy są automatycznie klasami w nowym stylu, opisywane przez nas funkcje, szczególnie dla klas w nowym stylu, są normą. Zdecydowałem się jednak, że te cechy opiszę osobno — z myślą o użytkownikach Pythona w wersjach 2.X — aby zaznaczyć, że w celu ich udostępnienia klasom należy zadeklarować dziedziczenie po klasie `object`.

Innymi słowy, gdy użytkownicy Pythona 3.0 znajdą odwołanie do „klas w nowym stylu”, oznacza to dla nich funkcję standardowo dostępną w klasach. Dla użytkowników Pythona 2.6 są to opcjonalne rozszerzenia mechanizmu klas.

W Pythonie 2.6 i wcześniejszych jedyna składniowa różnica deklaracji klas w nowym stylu polega na tym, że deklaruje się jej klasę nadziedzinną w postaci typu wbudowanego (na przykład `list`) lub specjalnego typu `object`. Nowa nazwa wbudowana `object` udostępniona została w taki sposób, by mogła służyć jako klasa nadziedzina dla klas w nowym stylu, jeśli żaden inny typ wbudowany nie nadaje się do zastosowania w określonym przypadku.

```
class newstyle(object):
...normalny kod...
```

Każda klasa pochodząca od `object` lub innego typu wbudowanego jest automatycznie traktowana jak klasa w nowym stylu. Dopóki typ wbudowany będzie się znajdował w którejś klasie nadzędnej, nowa klasa będzie traktowana jak klasa w nowym stylu. Klasy niepochodzące od typów wbudowanych uznawane są za klasyczne.

Klasy w nowym stylu nieznacznie się różnią od odmiany klasycznej, a różnice te są całkowicie nieistotne dla zdecydowanej większości użytkowników Pythona. Co więcej, model klasyczny, który jest częścią Pythona od jakichś piętnastu lat, nadal działa dokładnie tak samo, jak opisano to wcześniej.

Klasy w nowym stylu są prawie całkowicie zgodne wstecznie z typem klasycznym, jeśli chodzi o składnię oraz zachowanie, dodają jedynie pewne zaawansowane nowe opcje. Ponieważ jednak modyfikują jeden specjalny przypadek dziedziczenia, musiały być wprowadzone jako osobne narzędzie, by zapobiec ich negatywnemu wpływowi na istniejący kod, który oparty jest na wcześniejszym zachowaniu. Na przykład subtelne różnice w algorytmie wyszukiwania nazw w tzw. diamentowym wzorcu dziedziczenia oraz mechanizm wyszukiwania metod specjalnych dla operacji wbudowanych, jak `__getattr__`, mogą spowodować, że kod napisany dla Pythona 2.6 nie będzie działał w 3.0 zgodnie z oczekiwaniemi.

Następne dwa punkty zawierają przegląd nowości wprowadzonych w klasach w nowym stylu. Oczywiście różnice mają znaczenie dla użytkowników Pythona 2.X, natomiast ci, którzy na co dzień pracują z Pythonem 3.0, mogą ten opis potraktować jako listę zaawansowanych właściwości klas.

Nowości w klasach w nowym stylu

Klasy w nowym stylu różnią się od klasycznych pod kilkoma względami, niektóre z nich są subtelne, ale mogą powodować niezgodność z istniejącym kodem Pythona 2.X. Poniżej omawiamy najistotniejsze z tych różnic.

Połączenie klas i typów

Klasy są typami, a typy są klasami. W rzeczywistości stały się one synonimami. Funkcja wbudowana `type(I)`wróci klasę, z której utworzono obiekt `I`, nie generyczny typ, i najczęściej jest to ta sama klasa, którą zawiera `I.__class__`. Co więcej, klasy są instancjami klasy `type`; można również tworzyć klasy potomne klasy `type`, co pozwala na dostosowanie do własnych potrzeb mechanizmu tworzenia klas. Wszystkie klasy dziedziczą po klasie `object` i to samo dotyczy klasy `type`.

Kolejność wyszukiwania nazw w drzewie dziedziczenia

Algorytm wyszukiwania nazw w dziedziczeniu diamentowym został nieco zmodyfikowany.

W uproszczeniu: nazwy są wyszukiwane według zasady „najpierw poziomo od lewej do prawej, potem pionowo”.

Wyszukiwanie atrybutów dla funkcji wbudowanych

Metody `__getattr__` i `__getattribute__` nie są już wywoływane dla atrybutów wykorzystywanych przez operacje wbudowane. Oznacza to, że nie są one wykorzystywane do odczytu nazw dla metod specjalnych `__X__`. Wyszukiwanie tych metod odbywa się w klasach, nie w instancjach.

Nowe zaawansowane narzędzia

Klasy w nowym stylu oferują zestaw nowych narzędzi, jak sloty, właściwości, deskryptory i metodę `__getattribute__`. Większość z tych narzędzi ma bardzo specjalizowane zastosowania, przede wszystkim do konstruowania narzędzi programistycznych.

Ostatnią z wymienionych wyżej nowości omówiliśmy pokrótko w rozdziale 27. Wiedzę tę pogłębimy w rozdziale 37. przy okazji omawiania zarządzania atrybutami oraz w rozdziale 38. przy okazji dekoratorów prywatności. Pierwsza i druga z wymienionych wyżej nowości może powodować niezgodność z kodem napisanym dla Pythona 2.X, zatem omówimy je teraz, zanim przejdziemy do omawiania nowych funkcji wprowadzonych w klasach w nowym stylu.

Zmiany w modelu typów

W klasach w nowym stylu różnica między typami a klasami została zupełnie zatarta. Klasy stały się typami: obiekt `type` generuje klasy będące jego instancjami, natomiast klasy generują instancje swoich typów. W rzeczywistości nie ma żadnej różnicy między typami wbudowanymi, jak listy i ciągi znaków, a klasami zdefiniowanymi przez użytkownika. Dzięki temu możemy tworzyć klasy potomne typów wbudowanych, co zademonstrowaliśmy wcześniej w tym rozdziale — utworzenie klasy dziedziczącej po typie wbudowanym powoduje, że taka klasa jest klasą w nowym stylu, zatem staje się automatycznie typem zdefiniowanym przez użytkownika.

Oprócz możliwości tworzenia klas potomnych typów wbudowanych różnice między klasami typu klasycznego a klasami w nowym stylu stają się widoczne podczas kontroli typów. W klasycznych klasach Pythona 2.6 typem instancji klasy jest `instance`, natomiast typy obiektów wbudowanych są już bardziej specjalizowane.

```
C:\misc> c:\python26\python
>>> class C(object): pass                                # Klasyczne klasy Pythona 2.6
...
>>> I = C()
>>> type(I)                                              # Typ instancji
<class '__main__.C'>
>>> I.__class__
<class '__main__.C'>

>>> type(C)                                              # Klasa są typami tworzonimi przez użytkownika
<type 'type'>
>>> C.__class__
<type 'type'>

>>> type([1, 2, 3])                                       # Klasa wbudowane działają tak samo
<type 'list'>
>>> type(list)
<type 'type'>
>>> list.__class__
<type 'type'>
```

Jednak w przypadku klas w nowym stylu w 2.6 typem instancji klasy jest jej klasa, ponieważ klasy są w rzeczywistości typami zdefiniowanymi przez użytkownika — typem instancji jest jej klasa, a typ klasy zdefiniowanej przez użytkownika jest taki sam jak typ wbudowanego obiektu `type`. Klasy posiadają również atrybut `__class__`, ponieważ same są instancjami klasy `type`.

```
C:\misc> c:\python26\python
>>> class C(object): pass                                # Klasa w nowym stylu w 2.6
...
...
```

```

>>> I = C()
>>> type(I)                                # Typem instancji jest jej klasa
<class '__main__.C'>
>>> I.__class__
<class '__main__.C'>

>>> type(C)                                # Klasa jest typem, a typ jest klasą
<type 'type'>
>>> C.__class__
<type 'type'>

>>> type([1, 2, 3])                         # Klasa i typy wbudowane działają tak samo
<type 'list'>
>>> type(list)
<type 'type'>
>>> list.__class__
<type 'type'>

```

Ta sama zasada obowiązuje w przypadku wszystkich klas w Pythonie 3.0, ponieważ tutaj wszystkie klasy są klasami w nowym stylu, nawet jeśli nie posiadają żadnych jawnie zadeklarowanych klas nadzędnych. W rzeczywistości rozróżnienie między klasami zdefiniowanymi przez użytkownika a typami wbudowanymi w 3.0 zupełnie przestaje istnieć.

```

C:\misc> c:\python30\python
>>> class C: pass                         # W 3.0 wszystkie klasy są w nowym stylu
...
>>> I = C()
>>> type(I)                                # Typem instancji jest jej klasa
<class '__main__.C'>
>>> I.__class__
<class '__main__.C'>

>>> type(C)                                # Klasa jest typem, a typ jest klasą
<class 'type'>
>>> C.__class__
<class 'type'>

>>> type([1, 2, 3])                         # Klasa i typy wbudowane działają tak samo
<class 'list'>
>>> type(list)
<class 'type'>
>>> list.__class__
<class 'type'>

```

Jak widać, w 3.0 klasy są typami, ale typy są również klasami. Z technicznego punktu widzenia każda klasa jest generowana przez *metaklasę*, która z kolei również jest klasą. Najczęściej jest to klasa type albo jej klasa potomna z metodami przeciążonymi w celu modyfikacji zachowania tworzonych klas. Ta zmiana może spowodować problemy z kompatybilnością kodu wykorzystującego kontrolę typów, ale jest istotna dla twórców narzędzi dla programistów. Więcej informacji na temat metaklas przedstawię w dalszej części tego rozdziału oraz omówię je bardziej szczegółowo w rozdziale 39.

Konsekwencje z perspektywy kontroli typów

Oprócz możliwości modyfikowania typów wbudowanych oraz mechanizmu metaklas połączenie klas z typami w klasach w nowym stylu może nieść zagrożenia w postaci efektów ubocznych w kodzie wykorzystującym jawną kontrolę typów. Na przykład w Pythonie 3.0

typy instancji klas zdefiniowanych przez użytkownika można porównywać bezpośrednio i porównania te dadzą użyteczne informacje, tak samo jak w przypadku obiektów wbudowanych. Właściwość ta bierze się z faktu, że klasy są obecnie typami, a typ instancji jest jej klasą.

```
C:\misc> c:\python30\python
>>> class C: pass
...
>>> class D: pass
...
>>> c = C()
>>> d = D()
>>> type(c) == type(d)                                # 3.0: porównanie klas instancji
False

>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)

>>> c1, c2 = C(), C()
>>> type(c1) == type(c2)
True
```

W przypadku klasycznych klas Pythona 2.6 i wcześniejszych porównywanie typu instancji jest praktycznie bezużyteczne, ponieważ wszystkie instancje mają ten sam typ: `instance`. Aby efektywnie porównać typy, należy zastosować wartość atrybutu `__class__` instancji (w celu zachowania kompatybilności w 3.0 ta technika również zadziała, ale w tym przypadku jej stosowanie nie jest już niezbędne).

```
C:\misc> c:\python26\python
>>> class C: pass
...
>>> class D: pass
...
>>> c = C()
>>> d = D()
>>> type(c) == type(d)                                # 2.6: wszystkie klasyczne instancje są tego samego typu
True
>>> c.__class__ == d.__class__                         # Należy jawnie porównywać klasy
False

>>> type(c), type(d)
(<type 'instance'>, <type 'instance'>)
>>> c.__class__, d.__class__
(<class __main__.C at 0x024585A0>, <class __main__.D at 0x024588D0>)
```

Jak można się spodziewać, klasy w nowym stylu w 2.6 działają dokładnie tak samo jak w 3.0 — porównanie typów instancji automatycznie powoduje porównanie ich klas.

```
C:\misc> c:\python26\python
>>> class C(object): pass
...
>>> class D(object): pass
...
>>> c = C()
>>> d = D()
>>> type(c) == type(d)                                # 2.6 w nowym stylu: działają tak samo jak w 3.0
False
>>> type(c), type(d)
(<class '__main__.C'>, <class '__main__.D'>)
>>> c.__class__, d.__class__
(<class '__main__.C'>, <class '__main__.D'>)
```

Jednak należy pamiętać, o czym wspominałem w tej książce wielokrotnie, że bezpośrednie porównywanie typów instancji jest z reguły niewłaściwym podejściem w programach w Pythonie — powinniśmy wykorzystywać interfejsy realizowane przez obiekty, nie ich typy. W rzadkich sytuacjach, gdy jesteśmy zmuszeni do jawnego kontrolowania typów, powinniśmy raczej stosować funkcję wbudowaną `isinstance`. Jednak warto znać model typów zaimplementowany w Pythonie, ponieważ to pozwoli lepiej zrozumieć model klas.

Wszystkie obiekty dziedziczą po klasie `object`

Jedną z najważniejszych cech modelu klas w nowym stylu jest to, że wszystkie klasy bezpośrednio lub pośrednio dziedziczą po klasie `object`, a z faktu, że typy są również klasami, wynika, że każdy obiekt bezpośrednio lub pośrednio dziedziczy po klasie wbudowanej `object`. Weźmy na przykład następujący kod w Pythonie 3.0 (aby kod zadziałał w Pythonie 2.6, należy jawnie zadeklarować klasę nadzczną `object`).

```
>>> class C: pass
...
>>> X = C()

>>> type(X)
<class '__main__.C'>
>>> type(C)
<class 'type'>
```

Typ jest klasą instancji

Podobnie jak poprzednio, typem instancji klasy jest jej klasa, a typem klasy jest klasa `type`. Należy również pamiętać, że zarówno egzemplarz, jak i klasa dziedziczą po klasie `object`, ponieważ to właściwość wszystkich obiektów w klasach w nowym stylu.

```
>>> isinstance(X, object)
True
>>> isinstance(C, object)           # Klasa zawsze dziedziczą po object
True
```

Ta sama zasada obowiązuje w stosunku do klas wbudowanych, jak listy czy ciągi znaków, ponieważ w modelu klas w nowym stylu typy są klasami, a instancje typów wbudowanych również dziedziczą po klasie wbudowanej `object`.

```
>>> type('spam')
<class 'str'>
>>> type(str)
<class 'type'>

>>> isinstance('spam', object)      # To samo dotyczy klas wbudowanych
True
>>> isinstance(str, object)
True
```

W rzeczywistości klasa `type` również dziedziczy po klasie `object`, a klasa `object` dziedziczy po klasie `type` i choć obydwie są niezależnymi obiektami, to ta cykliczna zależność pozwala ująć w jedną całość tę najważniejszą cechę klas w nowym stylu: typy są klasami i służą do generowania klas.

```
>>> type(type)                   # Wszystkie klasy są typami, a typy są klasami
<class 'type'>
>>> type(object)
<class 'type'>

>>> isinstance(type, object)       # Wszystkie klasy dziedziczą po object, nawet klasa type
True
```

```
>>> isinstance(object, type)           # Typy tworzą klasy, a type jest klasą
True
>>> type is object
False
```

W praktyce ten model powoduje, że mamy do czynienia z mniejszą liczbą przypadków szczególnych, niż ma to miejsce w klasycznym modelu klas, a ponadto pozwala na pisanie kodu wykorzystującego funkcje wbudowanej klasy `object`. Szczegółowe omówienie możliwości wynikających z tego faktu znajduje się w dalszej części książki, tymczasem przejdźmy do innych zmian wprowadzonych w klasach w nowym stylu.

Zmiany w dziedziczeniu diamentowym

Jedną z widocznych zmian w klasach w nowym stylu jest nieco odmienny algorytm wyszukiwania nazw w tak zwanym wielodziedziczeniu *diamentowym*, to znaczy takim, w którym klasa dziedziczy po kilku klasach, mających wspólnego przodka. Dziedziczenie diamentowe jest zaawansowanym wzorcem projektowym, rzadko stosowanym w Pythonie w sposób bezpośredni i do tej pory przez nas nieomawianym, zatem nadszedł czas, aby nieco zgłębić ten temat.

W przypadku klas typu *klasycznego* procedura wyszukiwania nazw działa w pierwszej kolejności w pionie, a następnie od lewej do prawej. Innymi słowy, drzewo dziedziczenia jest najpierw przeszukiwane w głąb, z zachowaniem trzymania się lewej strony, a dopiero jeśli nazwa nie zostanie odszukana, procedura wraca na dół drzewa i klasy są przeszukiwane od lewej do prawej. W klasach *nowego stylu* procedura wyszukiwania najpierw poszukuje wszerz w klasach nadzędnych pierwszego poziomu, od lewej do prawej, a następnie przechodzi piętro wyżej. Algorytm wyszukiwania nazw jest nieco bardziej skomplikowany, ale tyle szczegółów powinno wystarczyć programistom.

Dzięki tej zmianie klasy na niższym poziomie dziedziczenia mogą przesłaniać nazwy zdefiniowane w klasach nadzędnych, niezależnie od tego, na której pozycji zostały zadeklarowane na liście klas nadzędnych. Co więcej, reguła wielokrotnego dziedziczenia pozwala uniknąć wielokrotnego przeszukiwania tej samej klasy nadzędnej, jeśli znajduje się ona na liście dziedziczenia większej liczby klas w drzewie.

Przykład dziedziczenia diamentowego

Aby zilustrować działanie dziedziczenia diamentowego, posłużymy się prostym schematem. Zaczniemy od klas typu klasycznego. Klasa `D` dziedziczy po klasach `B` i `C`, które mają wspólnego przodka — klasę `A`.

```
>>> class A:
    attr = 1          # Klasyczne klasy (Python 2.6)

>>> class B(A):
    pass            # B i C dziedziczą po A

>>> class C(A):
    attr = 2

>>> class D(B, C):
    pass            # Sprawdza w A, a następnie w C

>>> x = D()
>>> x.attr          # Kolejność przeszukiwania: x, D, B, A
1
```

W tym przypadku poszukiwana nazwa `attr` została znaleziona w klasie `A`, ponieważ w klasach typu klasycznego algorytm wyszukiwania nazw przeszukuje drzewo wzwyż, a dopiero potem w prawo, przez co Python przeszuka klasy w kolejności `D, B, A`, a na końcu `C`. Oczywiście po odszukaniu `attr` w klasie `A` przeszukiwanie się kończy, przez co nigdy nie dociera do `C`.

W klasach w nowym stylu dziedziczących po klasie `object`, czyli we wszystkich klasach w 3.0, algorytm wyszukiwania nazw jest inny. Python najpierw przeszuka klasę `D`, potem `B`, następnie `C`, a na końcu `A`. W tym przypadku wyszukiwanie zakończy się na `C`.

```
>>> class A(object):
    attr = 1          # Nowy styl (dziedziczenie po "object" niewymagane w 3.0)

>>> class B(A):
    pass

>>> class C(A):
    attr = 2

>>> class D(B, C):
    pass            # Sprawdza w C, a następnie w A

>>> x = D()
>>> x.attr        # Kolejność przeszukiwania: x, D, B, C
2
```

Ta zmiana procedury wyszukiwania nazw w drzewie dziedziczenia jest oparta na założeniu, że jeśli wmieszamy nazwy zdefiniowane w klasie `C` na niższym poziomie drzewa, chcielibyśmy, aby miały wyższy priorytet niż nazwy zdefiniowane w klasie `A`. Przyjmuje się, że jeśli ktoś zdefiniował klasę `C` z nazwami przesłaniającymi nazwy klas `A`, zrobił to po to, by używać nazw z klasy `C`. Tak się dzieje zawsze w sytuacji pojedynczego dziedziczenia, ale już przy wielodziedziczeniu diamentowym w przypadku klasycznego modelu klas tak nie musi być: wszystko zależy od tego, na której pozycji na liście klas nadzędnych w definicji klasy `D` umieścimy klasę `C`.

W praktyce tak skonstruowane dziedziczenie, w którym nazwy klasy `C` przesłaniają nazwy klasy `A`, po której dziedziczą, oznacza intencję programisty, aby były użyte nazwy klasy `C`. W przeciwnym razie użycie klasy `C` jest właściwie bezzasadne w kontekście dziedziczenia diamentowego — nie ma wpływu na nazwy zdefiniowane w klasie `A`, zatem z klasy `C` zostaną użyte tylko te nazwy, które są dla niej unikalne.

Jawne rozwiązywanie konfliktów

Oczywiście problemem założeń jest to, że zakładają one różne rzeczy. Jeśli taka modyfikacja kolejności wyszukiwania wydaje nam się zbyt subtelna, by ją zapamiętać, lub jeśli chcemy mieć większą kontrolę nad procesem wyszukiwania, możemy zawsze wymusić wybór atrybutu z dowolnego miejsca drzewa, przypisując lub w inny sposób wymieniając ten, który chcemy, w miejscu, w którym klasy są mieszane.

```
>>> class A:
    attr = 1          # Model klasyczny

>>> class B(A):
    pass

>>> class C(A):
    attr = 2
```

```

>>> class D(B, C):
    attr = C.attr
                                # Wybór C, na prawo

>>> x = D()
>>> x.attr
                                # Działa jak klasy w nowy stylu (wszystkie klasy w 3.0)
2

```

W powyższym kodzie drzewo klas z modelu klasycznego emuluje kolejność wyszukiwania klas w nowym stylu. Przypisanie do atrybutu w klasie `D` wybiera jego wersję z klasą `C`, odwracając tym samym normalną ścieżkę wyszukiwania dziedziczenia (`D.attr` będzie się znajdować najniżej w drzewie). Klasy w nowym stylu mogą w podobny sposób emulować model klasyczny, wybierając atrybut z góry w miejscu, w którym klasy mieszają się ze sobą.

```

>>> class A(object):
    attr = 1
                                # Nowy styl

>>> class B(A):
    pass

>>> class C(A):
    attr = 2

>>> class D(B,C):
    attr = B.attr
                                # Wybór A.attr, z góry

>>> x = D( )
>>> x.attr
                                # Działa jak model klasyczny
1

```

Jeśli mamy ochotę zawsze rozwiązywać konflikty w ten sposób, możemy w dużym stopniu zignorować różnice w kolejności wyszukiwania i nie polegać na założeniach dotyczących tego, co mieliśmy na myśli, tworząc kod klas.

Oczywiście atrybuty pobrane w ten sposób mogą również być funkcjami metod — metody są normalnymi obiektami, które można przypisywać.

```

>>> class A:
...     def meth(s): print ('A.meth')

>>> class C(A):
...     def meth(s): print ('C.meth')

>>> class B(A):
...     pass

>>> class D(B,C): pass
                                # Wykorzystanie domyślnej kolejności wyszukiwania
>>> x = D( )
                                # Różni się dla typów klas
>>> x.meth()
                                # Domyślnie kolejność klasyczna
A.meth

>>> class D(B,C): meth = C.meth
                                # Wybór metody klasy C: nowy styl
>>> x = D( )
>>> x.meth()
                                # Wybór metody klasy B: model klasyczny
C.meth

>>> class D(B,C): meth = B.meth
                                # Wybór metody klasy B: model klasyczny
>>> x = D( )
>>> x.meth()
                                # Wybór metody klasy C: nowy styl
A.meth

```

W powyższym kodzie wybieramy metody, w jawnym sposobie przypisując wartości do zmiennych znajdujących się niżej w drzewie. Możemy również wywołać pożądaną klasę w sposób jawnym. W praktyce ten wzorzec może zyskać na popularności, w szczególności dla elementów takich, jak konstruktory.

```
class D(B,C):
    def meth(self): # Redefiniowanie niżej
        ...
        C.meth(self) # Wybór metody klasy C przez wywołanie
```

Taki wybór przez przypisanie lub wywołanie w miejscu mieszania może w rezultacie odizolować nasz kod od różnic w typach klas. Jawnie rozwiązywanie konfliktów w ten sposób zapewnia, że nasz kod nie będzie się zachowywał inaczej w różnych przyszłych wersjach Pythona (poza, być może, potrzebą dziedziczenia klas od typów wbudowanych lub klasy `object` w przypadku nowego stylu w 2.6).



Nawet bez rozbieżności między modelem klasycznym a nowym stylem, technika ta może się czasem przydać w scenariuszach dziedziczenia wielokrotnego. Jeśli chcemy uzyskać część klasy nadzędnej z lewej strony oraz część klasy nadzędnej z prawej strony, musimy przekazać Pythonowi, który z atrybutów o tych samych nazwach ma wybrać, wykorzystując do tego jawnie przypisania w klasach podzrędnego. Powrócimy do tego zagadnienia w podrozdziale poświęconym pułapkom związanym z klasami pod koniec tego rozdziału.

Warto również zauważyć, iż wzorce wielokrotnego dziedziczenia po jednej klasie nadzędnej mogą w niektórych przypadkach być bardziej problematyczne, niż pokazano to tutaj (co na przykład stanie się, jeśli zarówno `B`, jak i `C` mają wymagane konstruktory wywołujące `A`?). Tego typu sytuacje są rzadkością w Pythonie, zatem uznajmy to zagadnienie za wykraczające poza tematykę tej książki (zainteresowanych odsyłam do funkcji wbudowanej `super`: oprócz udostępniania klas nadzędnych w drzewach z pojedynczym dziedziczeniem `super` oferuje tryb rozstrzygania konfliktów nazw w drzewach z wielokrotnym dziedziczeniem).

Zakres zmian kolejności wyszukiwania

Podsumowując, wzorzec dziedziczenia diamentowego jest przeszukiwany w inny sposób w klasach typu klasycznego i w klasach w nowym stylu i ta zmiana nie zachowuje zgodności wstecz. Należy jednak pamiętać, że ma ona wpływ na wielodziedziczenie typu diamentowego i klasy w nowym stylu w większości przypadków będą działały dokładnie tak samo jak klasy typu klasycznego. Co więcej, niewykluczone jest, że problem różnic algorytmu wyszukiwania ma znaczenie bardziej teoretyczne. W końcu aż do Pythona 2.2 nikt nie zainteresował się błędną koncepcją zastosowaną w algorytmie wyszukiwania, a na wprowadzenie go w standardzie czekaliśmy aż do Pythona 3.0, co sugeruje, że niewiele jest kodu Pythona, w którym ten problem może wystąpić.

Niemniej jednak należy pamiętać, że choć niewielu programistów w sposób jawnym i świadomym korzysta z diamentowego wzorca dziedziczenia, to *wszystkie* przypadki wielodziedziczenia w Pythonie 3.0 są implementacją właśnie tego wzorca. Dzieje się tak z tego powodu, że wszystkie klasy dziedziczą po klasie `object`, która stanowi odpowiednik klasy `A` z naszego powyższego przykładu. Z tego powodu nowy algorytm wyszukiwania nazw w modelu klas w nowym stylu

nie tylko wprowadza modyfikację semantyki działania dziedziczenia, lecz również optymalizację wydajności — dzięki uniknięciu wielokrotnego wyszukiwania nazwy w tej samej klasie występującej w drzewie wielokrotnie.

Co nie mniej istotne, domyślnie wykorzystana klasa nadrzędna `object` dostarcza wszystkim obiektom wielu metod wykorzystywanych przez operacje wbudowane, jak `_str_` czy `_repr_`. Aby zapoznać się z listą metod oferowanych przez klasę `object`, wystarczy wywołać `dir(object)`. Gdyby nie wprowadzono zmiany algorytmu wyszukiwania, w klasach w nowym stylu metody te nie mogłyby być efektywnie przesłonięte, chyba że klasy je redefiniujące zostałyby zadeklarowane na pierwszym miejscu na liście klas nadrzędnych. Innymi słowy, zmiana algorytmu wyszukiwania nazw w dziedziczeniu diamentowym to wręcz warunek konieczny efektywnego funkcjonowania klas w nowym stylu.

Bardziej praktyczny przykład wykorzystania metod klasy `object` w dziedziczeniu w Pythonie 3.0 można znaleźć w definicji klasy `ListTree` z modułu `lister.py` w poprzednim rozdziale oraz w przykładzie `classtree.py` w rozdziale 28.

Nowości w klasach w nowym stylu

Oprócz zmian opisanych w poprzednim punkcie (które są, szczerze mówiąc, zbyt akademickie i zaawansowane, aby mogły mieć szczególnie znaczenie dla większości Czytelników) klasy w nowym stylu oferują kilka nowych możliwości o bardziej bezpośrednich i praktycznych zastosowaniach. Kolejne punkty zawierają przegląd tych nowości, które są dostępne w klasach w nowym stylu w Pythonie 2.6 oraz we wszystkich klasach w 3.0.

Sloty

Klasy w nowym stylu mają możliwość ograniczenia listy nazw, które można wykorzystać w instancjach. Służy do tego specjalny atrybut `_slots_`, któremu przypisuje się listę dostępnych nazw. Funkcja ta służy kontroli dostępności nazw, jak również optymalizacji szybkości działania i zużycia pamięci programu.

Ten specjalny atrybut z reguły definiuje się na początku definicji klasy — w instancjach mogą być użyte wyłącznie te atrybuty, których nazwy znajdują się na liście `_slots_`. Jednak pozostałe zasady obowiązujące w Pythonie nadal pozostają w mocy: atrybutom instancji należy przypisać wartość, zanim będzie można z nich cokolwiek odczytać.

```
>>> class limiter(object):
...     __slots__ = ['age', 'name', 'job']
...
>>> x = limiter()
>>> x.age                                         # Przed użyciem należy przypisać wartość
AttributeError: age

>>> x.age = 40
>>> x.age
40
>>> x.ape = 1000                                    # Nielegalne wywołanie: nazwa niezdefiniowana w __slots__
AttributeError: 'limiter' object has no attribute 'ape'
```

Sloty można postrzegać jako pewnego rodzaju wyłom w dynamicznej naturze Pythona: programista może zabronić przypisania nowych nazw obiektom. Jednak tę funkcję można wykorzystać

jako zabezpieczenie przed literówkami (wykryta zostanie już próba przypisania do atrybutu, którego nazwa nie znalazła się w `_slots_`, co pozwoli wychwycić błędy na wczesnym etapie testowania) oraz jako mechanizm optymalizujący. Utworzenie każdej nowej instancji powoduje alokację słownika przestrzeni nazw, co w przypadku dużej liczby instancji może być kosztowne czasowo i pamięciowo i czego można uniknąć, jeśli zechcemy wykorzystać tylko niewielką część atrybutów. Miejsce zajmowane w pamięci i szybkość wykonania można zoptymalizować, deklarując w klasie atrybut `_slots_`.

Sloty a kod generyczny

W rzeczywistości niektóre instancje wykorzystujące sloty w ogóle nie posiadają atrybutu `_dict_`, co może komplikować programy narzędziowe (również przykładowy kod z tej książki). Narzędzia, które uzyskują dostęp do atrybutów na podstawie ich nazw, muszą być zaimplementowane z użyciem bardziej neutralnych rozwiązań, jak funkcje wbudowane `getattr`, `setattr` i `dir`, które pośrednio wykorzystują `_dict_` lub `_slots_`, ale w sposób niewidoczny dla programisty. W niektórych przypadkach kod może być zmuszony do przeszukania każdego z tych atrybutów specjalnych.

Jeśli na przykład zostały użyte sloty, instancje nie będą miały słownika atrybutów, a Python do zaalokowania slotów dla atrybutów wykorzysta *deskryptory* klas, opisane szerzej w rozdziale 37. Atrybut `_slots_` ogranicza listę nazw atrybutów instancji, ale do odczytu i zapisu tych atrybutów nadal można wykorzystać generyczne narzędzia. Oto przykład dla Pythona 3.0 (oraz 2.6 przy użyciu klas w nowym stylu):

```
>>> class C:
...     ['a', 'b']                                     # __slots__ oznacza, że nie ma __dict__
...
>>> X = C()
>>> X.a = 1
>>> X.a
1
>>> X.__dict__
AttributeError: 'C' object has no attribute '__dict__'
>>> getattr(X, 'a')
1
>>> setattr(X, 'b', 2)                           # Ale getattr() i setattr() będą działać
>>> X.b
2
>>> 'a' in dir(X)                                # dir() również znajduje atrybuty w slotach
True
>>> 'b' in dir(X)
True
```

Bez słownika przestrzeni nazw nie jest możliwe przypisanie nowych nazw atrybutów w instancji, jeśli nie są zadeklarowane na liście slotów:

```
>>> class D:
...     __slots__ = ['a', 'b']
...     def __init__(self): self.d = 4      # Nie można dodawać nazw niezdefiniowanych w __dict__
...
>>> X = D()
AttributeError: 'D' object has no attribute 'd'
```

Dodatkowe atrybuty można jednak obsłużyć, deklarując nazwę `_dict_` w atrybucie `_slots_`, co pozwoli na wykorzystanie w instancjach słownika przestrzeni nazw. W tym przypadku wykorzystane będą *obydwia* mechanizmy zapisu atrybutów, ale generyczne narzędzia, takie jak `getattr`, będą je traktować jako jednolity zestaw atrybutów.

```

>>> class D:
...     __slots__ = ['a', 'b', '__dict__'] # Deklarujemy __dict__ jako jeden ze slotów
...     c = 3                                # Atrybuty klasy działają bez zmian
...     def __init__(self): self.d = 4          # d zostaje zapisane w __dict__ a w __slots__
...
>>> X = D()
>>> X.d
4
>>> X.__dict__                               # Niektóre obiekty posiadają __dict__ oraz __slots__
{'d': 4}                                     # getattr() odczyta wartość atrybutu dowolnego rodzaju
>>> X.__slots__                            ['a', 'b', '__dict__']
>>> X.c
3
>>> X.a                                     # Atrybutów nie można odczytać, jeśli nie zostały przypisane
AttributeError: a
>>> X.a = 1
>>> setattr(X, 'a',), getattr(X, 'c'), getattr(X, 'd')
(1, 3, 4)

```

Jednak kod pobierający atrybuty w sposób generyczny musi obsługiwać ręcznie obydwa mechanizmy, ponieważ funkcja `dir` zwraca również nazwy atrybutów odziedziczonych (poniższy przykład wykorzystuje iterator słownikowy do odczytu listy kluczy):

```

>>> for attr in list(X.__dict__) + X.__slots__:
...     print(attr, '=>', getattr(X, attr))

d => 4
a => 1
b => 2
__dict__ => {'d': 4}

```

Zarówno `__dict__`, jak i `__slots__` może nie występować w instancji, zatem bardziej prawidłowa implementacja będzie wyglądała następująco:

```

>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
...     print(attr, '=>', getattr(X, attr))

d => 4
a => 1
b => 2
__dict__ => {'d': 4}

```

Wiele slotów w klasach nadrzędnych

Należy zwrócić uwagę, że powyższy kod obsługuje wyłącznie nazwy zdefiniowane w `__slots__` na *najniższym* poziomie drzewa dziedziczenia. Jeśli klasy na wyższych poziomach dziedziczenia mają zdefiniowane własne atrybuty `__slots__`, narzędzia programistyczne muszą implementować własne procedury dostępu do atrybutów (to znaczy interpretować sloty jako atrybuty klas, nie instancji).

Deklaracje slotów mogą występować w wielu klasach w drzewie dziedziczenia, ale podlegają pewnym ograniczeniom, które mogą wydać się mało sensowne dla osób nieznających szczegółów implementacji slotów, będących w rzeczywistości deskryptorami klas (jest to narzędzie, któremu przyjrzymy się szczegółowo w dalszej części książki).

- Jeśli klasa dziedziczy po klasie nieposiadającej atrybutu `__slots__`, atrybut `__slots__` klasy nadzędnej będzie zawsze dostępny, przez co atrybut `__slots__` w klasie podrzędnej nie będzie miał zastosowania.

- Jeśli klasa definiuje te same nazwy slotów co klasa nadziedziona, to sloty klasy nadziedznej będą dostępne poprzez deskryptor klasy.
- Deklaracja `__slots__` dotyczy wyłącznie klasy, w której występuje, klasy podziedzne będą posiadały `__dict__`, chyba że również definiują `__slots__`.

Wracając do problemu wypisywania atrybutów instancji, sloty w klasach na różnych poziomach drzewa dziedziczenia zmuszają nas do ręcznego przechodzenia w tym drzewie, wykorzystania funkcji wbudowanej `dir` lub całkowicie odrębnego traktowania nazw zdefiniowanych w `__slots__`.

```
>>> class E:
...     __slots__ = ['c', 'd']                      # Klasa nadziedziona posiada sloty
...
>>> class D(E):
...     __slots__ = ['a', '__dict__']                 # Podobnie jej klasa potomna
...
>>> X = D()
>>> X.a = 1; X.b = 2; X.c = 3                  # Instancia jest połączeniem tych klas
>>> X.a, X.c
(1, 3)

>>> E.__slots__                                # Ale sloty nie są dziedziczone z klas nadziedzonych
['c', 'd']
>>> D.__slots__                                # Instancia dziedziczy sloty wyłącznie z jej klas
['a', '__dict__']
>>> X.__slots__                                # I posiada własny słownik __attr__
['a', '__dict__']
>>> X.__dict__                                 # Brakuje slotów z klasą nadziedznej!
{'b': 2}

>>> for attr in list(getattr(X, '__dict__', [])) + getattr(X, '__slots__', []):
...     print(attr, '=>', getattr(X, attr))
...
b => 2                                         # Brakuje slotów z klasą nadziedznej!
a => 1
__dict__ => {'b': 2}

>>> dir(X)                                     # dir() wyświetla wszystkie nazwy
[...pominieć część wyniku... 'a', 'b', 'c', 'd']
```

Jeśli możemy sobie pozwolić na takie uogólnienie, sloty najlepiej traktować jako atrybuty klasy, zamiast próbować wrzucać je do jednego worka ze zwykłymi atrybutami instancji. Więcej informacji na temat slotów można znaleźć w standardowym podręczniku Pythona. Ponadto w rozdziale 38., przy okazji implementacji prywatnego dekoratora, zademonstrujemy wykorzystanie zarówno `__slots__`, jak i `__dict__`.

Omówienie problemów z obsługą slotów, jakie mogą napotkać programy, można znaleźć przy okazji opisu modułu `lister.py` w punkcie omawiającym wielokrotne dziedziczenie w poprzednim rozdziale (uwaga na temat slotów w kontekście prezentowanego kodu). W tego typu generycznym narzędziu odczytującym atrybuty instancji w celu prawidłowego obsłużenia slotów należy jawnie zastosować dodatkowy kod albo zaimplementować uogólniony algorytm uwzględniający atrybuty zapisane w slotach.

Właściwości klas

Mechanizm znany jako *właściwości* udostępnia kolejny sposób definiowania w klasach w nowym stylu wywoływanych automatycznie metod służących do dostępu lub przypisania do atrybutów instancji. Opcja ta jest alternatywą wielu wykorzystywanych obecnie zastosowań metod przeciążania `__getattr__` oraz `__setattr__`, które omawialiśmy w rozdziale 29. Właściwości mają podobny efekt do tych dwóch metod, jednak mają dodatkowe wywołanie metody zarezerwowane jedynie dla dostępu do zmiennych, które wymagają dynamicznego obliczenia. Właściwości (oraz sloty) oparte są na nowej koncepcji deskryptorów atrybutów, która jest zbyt zaawansowana, byśmy ją tutaj omawiali.

Mówiąc w skrócie, właściwości są rodzajem obiektu przypisanego do nazwy atrybutu klasy. Są generowane przez wywołanie wbudowanej funkcji `property` z trzema metodami (programi obsługą dla operacji pobierania, ustawiania oraz usuwania), a także łańcuchem znaków dokumentacji. Jeśli jakiś argument zostanie przekazany jako `None` lub pominięty, operacja ta nie jest obsługiwana. Właściwości są zazwyczaj przypisywane na najwyższym poziomie instrukcji `class` (na przykład `name = property(...)`). Jeśli się je przypisze w taki sposób, dostęp do samego atrybutu klasy (na przykład `obj.name`) są automatycznie przekierowywane do jednej z metod akcesorów przekazanej do funkcji `property`. Metoda `__getattr__` pozwala na przykład klasom przechwytywać referencje do niezdefiniowanych atrybutów.

```
>>> class classic:
...     def __getattr__(self, name):
...         if name == 'age':
...             return 40
...         else:
...             raise AttributeError
...
>>> x = classic()
>>> x.age                                     # Wykonuje __getattr__
40
>>> x.name                                    # Wykonuje __getattr__
AttributeError
```

Poniżej znajduje się ten sam przykład, jednak tym razem zostały wykorzystane właściwości (właściwości działają z dowolnymi klasami, ale w przypadku Pythona 2.6 wymagają jawnej deklaracji dziedziczenia po klasie `object`).

```
>>> class newprops(object):
...     def getage(self):
...         return 40
...     age = property(getage, None, None, None) # Operacje get, set, del, dokumentacja
...
>>> x = newprops()
>>> x.age                                      # Wykonuje getage
40
>>> x.name                                     # Normalne pobranie
AttributeError: newprops instance has no attribute 'name'
```

Dla niektórych zadań programistycznych właściwości mogą być mniej skomplikowane oraz szybsze do wykonania od tradycyjnych technik. Kiedy na przykład dodamy obsługę przypisania atrybutów, właściwości staną się bardziej atrakcyjne — do wpisania mamy mniej kodu, a żadne dodatkowe wywołania metod nie są konieczne dla przypisania do atrybutów, których nie chcemy obliczać dynamicznie.

```

>>> class newprops(object):
...     def getage(self):
...         return 40
...     def setage(self, value):
...         print('ustawienie wieku:', value)
...         self._age = value
...     age = property(getage, setage, None, None)
...
>>> x = newprops( )
>>> x.age
40
# Wykonuje getage
>>> x.age = 42
ustawienie wieku: 42
# Wykonuje setage
>>> x._age
# Normalne pobranie; nie ma wywołania getage
42
# Normalne przypisanie; nie ma wywołania setage
>>> x.job = 'instruktor'
# Normalne przypisanie; nie ma wywołania setage
>>> x.job
# Normalne pobranie; nie ma wywołania getage
'instruktor'

```

Odpowiednik tego kodu w modelu klasycznym wymaga kilku dodatkowych wywołań przy pisaniu atrybutów i do ich obsługi wymaga zastosowania słownika (a w klasach w nowym stylu wywołania metody `__setattr__` klasy nadrzędnej) w celu uniknięcia pętli.

```

>>> class classic:
...     def __getattr__(self, name):          # Przy niezdefiniowanej referencji
...         if name == 'age':
...             return 40
...         else:
...             raise AttributeError
...     def __setattr__(self, name, value):    # Przy wszystkich przypisaniach
...         print('ustawienie: ', name, value)
...         if name == 'age':
...             self.__dict__['_age'] = value
...         else:
...             self.__dict__[name] = value
...
>>> x = classic()
>>> x.age
40
# Wykonuje __getattr__
>>> x.age = 41
ustawienie: age 41
# Wykonuje __setattr__
>>> x._age
# Zdefiniowane; nie ma wywołania __getattr__
41
# Wykonuje znów __setattr__
>>> x.job = 'instruktor'
# Wykonuje znów __setattr__
# Zdefiniowane; nie ma wywołania __getattr__

```

Właściwości wydają się w tym przypadku lepszym rozwiązańiem. Jednak istnieją sytuacje, gdy lepszym rozwiązaniem są metody `__getattr__` oraz `__setattr__`, na przykład w celu zaimplementowania bardziej dynamicznych czy uniwersalnych interfejsów od tych, które mogą być zaimplementowane przez właściwości. W wielu przypadkach nie da się na etapie tworzenia kodu klasy ustalić zbioru atrybutów, jaki ma być obsługiwany, może on nawet nie istnieć w jakiejkolwiek namacalnej formie (na przykład kiedy deleguje się referencje do dowolnych metod do obiektów opakowanych czy osadzonych w sposób uniwersalny). W takich przypadkach ogólny mechanizm obsługi atrybutów `__getattr__` czy `__setattr__` z przekazaną nazwą atrybutu może być lepszym wyborem. Ponieważ ten mechanizm potrafi sobie również radzić z prostszymi przypadkami, właściwości są w dużej mierze rozszerzeniem opcjonalnym.

Więcej szczegółów na temat wykorzystania obydwu metod dynamicznej obsługi atrybutów można znaleźć w rozdziale 37. w ostatniej części książki. Znajdziemy tam między innymi zastosowanie składni dekoratora *funkcji*, która zostanie wprowadzona w dalszej części niniejszego rozdziału.

Przeciążanie nazw — `__getattribute__` i deskryptory

Metoda przeciążania `__getattribute__`, dostępna jedynie dla klas w nowym stylu, pozwala klasom na przechwytywanie *wszystkich* referencji do atrybutów, nie tylko referencji niezdefiniowanych (tak, jak `__getattr__`). Jej użycie jest również znacznie bardziej skomplikowane w porównaniu do `__getattr__` oraz `__setattr__` (jest podatna na zapętleńia, podobnie do `__setattr__`, ale w inny sposób).

Oprócz metod przeciążania właściwości i operatorów Python oferuje koncepcję *deskryptorów* atrybutów. Są to klasy wyposażone w metody `__set__` i `__get__` przypisane atrybutom klasy i dziedziczone przez instancje. Klasy te przechwytyują żądania odczytu i zapisu atrybutu. Deskryptory są w pewnym sensie uogólnioną koncepcją właściwości, a raczej właściwości stanowią uproszczony sposób definicji specyficznego typu deskryptora — mówiąc dokładniej: takiego deskryptora, który uruchamia wskazane funkcje w reakcji na próby dostępu. Deskryptory są również wykorzystywane do implementacji slotów, które omówiliśmy wcześniej.

Właściwości, metoda `__getattribute__` oraz deskryptory są zagadnieniami dość zaawansowanymi, na razie więc odłożymy ich analizę. Szczegóły dotyczące tych technik oraz pogłębienie tematu właściwości można znaleźć w rozdziale 37. w ostatniej części książki.

Metaklasy

Większość zmian i dodatkowych funkcji klas w nowym stylu wpasowuje się w koncepcję typów będących klasami, o której wspominaliśmy wcześniej w tym rozdziale, ponieważ ta koncepcja oraz klasy w nowym stylu w ogólności zostały wprowadzone w połączeniu z wyeliminowaniem sytuacji istnienia dwóch modeli klas od Pythona 2.2 wzwyż. Jak mieliśmy okazję się przekonać od Pythona 3.0 to połączenie jest kompletne: klasy są teraz typami i typy klasami.

Wraz z tymi zmianami Python rozwinał się jako spójny protokół kodowania *metaklas*, które są klasami potomnymi klasy `type` i służą do obsługi procesu tworzenia klas. W tym celu posiadają mechanizmy zarządzania i modyfikacji semantyki obiektów klas. Są zaawansowanym zagadnieniem, które dla większości programistów Pythona stanowi zagadnienie opcjonalne, zatem na razie nie będziemy się zagłębiać w jego szczegóły. Do metaklas wrócimy na chwilę w dalszej części rozdziału przy okazji omawiania dekoratorów klas, a szczegółowo zajmiemy się nimi w rozdziale 39., na zakończenie książki.

Metody statyczne oraz metody klasy

W wersjach Pythona starszych od 2.2 funkcji metod klas nie można było nigdy wywoływać bez instancji — *metody statyczne* działają podobnie jak zdefiniowane w klasie funkcje nieprzyjmujące instancji, natomiast *metody klas* przyjmują obiekt klasy zamiast obiektu instancji. Te dwa typy metod powstały równolegle do wprowadzenia klas w nowym stylu, ale działają również w klasycznym modelu klas.

Aby aktywować te specjalne tryby działania metod, należy w klasie wywołać dedykowane funkcje wbudowane `staticmethod` i `classmethod` lub wykorzystać składnię dekoratora, o czym powiemy więcej w dalszej części rozdziału. W Pythonie 3.0 metody wywoływane z klasy bez podawania instancji nie wymagają deklaracji `staticmethod`, ale takie metody wywoywane z instancji już takiej deklaracji wymagają.

Do czego potrzeujemy metod specjalnych?

Jak mieliśmy okazję się dowiedzieć, metoda klasy z reguły w swoim pierwszym argumentem wymaga podania instancji, która jest wykorzystywana jako podmiot wykonywanej metody. Istnieją jednak dwa sposoby modyfikacji tego domyślnego działania. Zanim wyjaśniamy, na czym polegają, należy wyjaśnić, do czego to może być przydatne.

Czasem programiści potrzebują przetwarzać dane związane z klasami, nie z instancjami. Weźmy na przykład śledzenie liczby instancji utworzonych z klasy lub zarządzanie listą instancji klasy istniejących aktualnie w pamięci. Informacje tego typu i ich przetwarzanie to zadanie dla klasy, nie jej instancji. A dokładniej: informacje tego typu są najczęściej zapisywane w obiekcie klasy i przetwarzane bez użycia instancji.

Do zadań tego typu najczęściej wystarczą zwykłe funkcje zakodowane poza klasą: funkcje mają dostęp do atrybutów klasy za pośrednictwem jej nazwy, dzięki czemu mają też dostęp do danych zapisanych bezpośrednio w klasie i nie potrzebują tworzyć instancji. Jednak lepiej jest powiązać kod tego typu z klasą, której dotyczy, pozwalając na powtórne jego użycie przy wykorzystaniu dziedziczenia. Kod ten najlepiej jest zaimplementować w postaci metod klasy. Aby jednak takie metody zadziałyły zgodnie z oczekiwaniemi, nie powinny wymagać przekazywania instancji w argumentem `self`.

Python implementuje takie mechanizmy w formie tak zwanych *metod statycznych* — zwykłych funkcji nieobsługujących argumentu `self`, zdefiniowanych w ciele klasy i zaprojektowanych z myślą o współpracy z obiektem klasy, a nie — jak zwykłe metody — z obiektem instancji. Dane zapisane w klasie są dostępne z wszystkich instancji i nie są powiązane z żadną z nich.

Python obsługuje również metody klas, choć technika ta jest rzadziej stosowana. *Metody klas* to takie metody, którym zamiast instancji przekazywana jest klasa, niezależnie od tego, czy zostały wywołane z klasy, czy z instancji. Metody tego typu mają dostęp do danych klasy za pośrednictwem klasy przekazanej w atrybutie. Zwykłe metody (nazywane obecnie *metodami instancji*) otrzymują w pierwszym atrybutie instancję, z której zostały wywołane. W przypadku metod statycznych i metod klas tak się nie dzieje.

Metody statyczne w 2.6 i 3.0

Koncepcja metod statycznych nie różni się w Pythonie 2.6 i 3.0, ale wymagania w stosunku do jej implementacji zmieniły się nieco w Pythonie 3.0. Niniejsza książka omawia obie wersje Pythona, zatem należy wyjaśnić różnice w obydwu modelach, zanim przejdziemy do przykładowego kodu.

W rzeczywistości omawianie tego zagadnienia rozpoczęliśmy już w poprzednim rozdziale, przy okazji metod niezwiązańych. Jak pamiętamy, Python 2.6 i 3.0 metodom wywoływanym z instancji zawsze przekazują tę instancję. Jednak w Pythonie 3.0 metody wywoływane z klas są traktowane inaczej niż w 2.6:

- w Pythonie 2.6 metoda w kontekście klasy jest traktowana jak *metoda niezwiązana*, której nie można wywołać, nie przekazując jej instancji w sposób jawny,
- w Pythonie 3.0 metoda w kontekście klasy jest traktowana jak *zwykła funkcja*, którą można wywołać bez podawania instancji.

Innymi słowy, metody klas w Pythonie 2.6 wymagają przekazania im instancji, niezależnie od tego, czy są wywoływane z instancji, czy z klasy. Natomiast w Pythonie 3.0 instancję musimy przekazać metodzie jedynie wówczas, gdy jej wymaga — metody, które nie mają zadeklarowanego argumentu `self`, mogą być wywoływane z klasy bez przekazywania instancji. Oznacza to, że wersja 3.0 pozwala na wykorzystanie w klasie zwykłych funkcji, pod warunkiem że nie oczekują argumentu instancji i że im takiego argumentu nie będziemy podawać. W efekcie mamy następującą sytuację:

- w Pythonie 2.6 metodę zawsze musimy deklarować jako statyczną, jeśli chcemy ją wykonywać bez instancji, niezależnie, czy jest wywoływana z klasy, czy z instancji,
- w Pythonie 3.0 metod nie musimy deklarować jako statyczne, jeśli są wywoływane wyłącznie z klas, ale musimy to zrobić, jeśli chcemy je wywoływać z instancji.

Załóżmy na przykład, że chcemy wykorzystać atrybuty klasy do zliczania tego, jak dużo instancji zostało wygenerowanych z klasy (tak jak w poniższym pliku *spam.py*). Należy pamiętać, że atrybuty klasy są współdzielone przez wszystkie instancje. Dodatkowo mamy metodę, która wypisuje ten licznik. Jak pamiętamy, atrybuty klas są współdzielone przez wszystkie instancje. Z tego powodu zapisanie atrybutu w samej klasie daje pewność, że jego wartość będzie dostępna we wszystkich instancjach.

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Liczba utworzonych instancji: ", Spam.numInstances)
```

Metoda `printNumInstances` została napisana w taki sposób, aby dane przechowywała w klasie, nie w instancji, ponieważ wynik dotyczy *wszystkich* instancji. Z tego powodu chcemy mieć możliwość uruchamiania jej bez przekazywania instancji. Co więcej, nie chcemy tworzyć nowej instancji do odczytu licznika, ponieważ to spowodowałoby zmianę wartości tego licznika. Innymi słowy, potrzebujemy metody statycznej niewykorzystującej obiektu `self`.

Prezentowany kod jest prawidłowy w niektórych wersjach Pythona, ale nie działa w innych; znaczenie ma również sposób użycia tej klasy, czyli to, czy metoda `printNumInstances` jest wywoływana z klasy, czy z instancji. W Pythonie 2.6 (a w zasadzie ogólnie w serii 2.X) wywołania metod nieposiadających argumentu `self` nie będą działały, niezależnie od tego, czy wywołamy je z instancji, czy z klasy (część komunikatu o błędzie została pominięta).

```
C:\misc> c:\python26\python
>>> from spam import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()

>>> Spam.printNumInstances()
# W 2.6 metod niezwiązanych nie można wywoływać z klasy
# Metody oczekują obiektu self
TypeError: unbound method printNumInstances() must be called with Spam instance as
first argument (got nothing instead)
>>> a.printNumInstances()
TypeError: printNumInstances() takes no arguments (1 given)
```

Problemem jest to, że metody klas bez wiązania nie są w 2.6 do końca tym samym co proste funkcje. Mimo że w deklaracji `def` nie ma argumentów, metoda nadal oczekuje przekazania instancji, ponieważ funkcja jest związana z klasą. W Pythonie 3.0 (i nowszych) wywołanie metod bez `self` zadziała w przypadku wywołania z klasy, ale w przypadku wywołania z instancji zakończy się błędem.

```
C:\misc> c:\python30\python
>>> from spam import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()

>>> Spam.printNumInstances()                                # W 3.0 można wywoływać metody z klas
Liczba utworzonych instancji: 3                          # Jednak wywołania metod z instancji przekazują self

>>> a.printNumInstances()                                 # Różnica w 3.0
TypeError: printNumInstances() takes no arguments (1 given)
```

Wywołanie metod nieobsługujących `self`, jak `printNumInstances`, nie zadziała w Pythonie 2.6, ale w 3.0 już tak. Z drugiej strony, tego typu wywołania z instancji nie będą działały niezależnie od wersji Pythona, ponieważ *instancja* jest automatycznie przekazywana do metody, nawet jeśli ta metoda nie posiada argumentu, w którym mogłaby ją przyjąć.

```
Spam.printNumInstances()                                  # Nie działa w 2.6, działa w 3.0
instance.printNumInstances()                            # Nie działa w 2.6 i 3.0
```

Jeśli ktoś zdecyduje się na wykorzystanie Pythona 3.0 i metody nieobsługujące `self` będzie wywoływała wyłącznie z klasy, to w zasadzie ma gotowy mechanizm metod statycznych. Jeśli jednak zdecyduje się na wykorzystanie takich metod w 2.6 lub potrzebuje wywoływać je z instancji w 2.6 i 3.0, będzie potrzebował dodatkowego mechanizmu adaptującego metody do takiego zastosowania albo powinien oznaczyć je jako specjalne. Przyjrzyjmy się każdemu z tych sposobów.

Alternatywy dla metod statycznych

Oprócz zadeklarowania metody jako specjalnej (nieprzyjmującej argumentu `self`) programista ma do dyspozycji jeszcze kilka narzędzi pozwalających uzyskać zbliżony efekt. Jeśli potrzebujemy funkcji, które będą wykonywać działania na obiektach klas bez przekazywania instancji, najprostszym rozwiązaniem jest po prostu wykorzystanie zwykłych funkcji zakodowanych niezależnie od definicji klasy. Dzięki temu w wywołaniu nie jest automatycznie przekazywana instancja. Poniższy przykład, będący modyfikacją modułu `spam.py`, działa bez przeskódeł w 3.0 i 2.6 (choć w 2.6 wyświetli dodatkowe nawiasy w efekcie wywołania instrukcji `print`).

```
def printNumInstances():
    print("Liczba utworzonych instancji: ", Spam.numInstances)

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

>>> import spam
>>> a = spam.Spam()
>>> b = spam.Spam()
>>> c = spam.Spam()
>>> spam.printNumInstances()                               # Funkcja jest poderwana od klasy i nie można
Liczba utworzonych instancji: 3                         # nią manipulować w ramach dziedziczenia
>>> spam.Spam.numInstances
3
```

Nazwa klasy jest dostępna dla zwykłej funkcji jako nazwa globalna, zatem kod zadziała bez przeszkodeń. Warto zwrócić uwagę, że nazwa funkcji staje się nazwą globalną, ale tylko w ramach modułu, w którym jest zdefiniowana, zatem nie będzie kolidować z nazwami zdefiniowanymi w innych plikach programu.

Zanim w Pythonie pojawiły się metody statyczne, właśnie taki był domyślny sposób realizacji tego typu zadań. Python oferuje moduły jako mechanizm kontroli przestrzeni nazw, zatem można by dyskutować, czy w ogóle jest sens umieszczać metody klas w definicji klas. Proste funkcje zaimplementowane w prezentowany sposób będą realizowały te same zadania co metody klas, a będą związane z klasami dzięki umieszczeniu ich definicji w tym samym module.

Niestety, takie podejście ma jednak sporo wad. Po pierwsze, w przestrzeni nazw modułu pojawia się osobna nazwa używana do przetwarzania tylko jednej klasy. Po drugie, funkcja jest tylko pośrednio powiązana z klasą, a jej definicja może znajdować się w zupełnie innym miejscu kodu. Co gorsza, proste funkcje nie mogą być przekazywane i przesłaniane przez dziedziczenie, ponieważ istnieją poza przestrzenią nazw klas, zatem klasy potomne nie będą brane pod uwagę przez funkcję i jedynym rozwiązaniem tego problemu jest ponowna definicja funkcji dla każdej klasy potomnej.

Można spróbować rozszerzyć ten kod w sposób bardziej przenośny, wykorzystując zwykłą metodę instancji i dbając o to, aby zawsze wywoływać ją z instancji.

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances(self):
        print("Liczba utworzonych instancji: ", Spam.numInstances)

>>> from spam import Spam
>>> a, b, c = Spam(), Spam(), Spam()
>>> a.printNumInstances()
Number of instances created: 3
>>> Spam.printNumInstances(a)
Number of instances created: 3
>>> Spam().printNumInstances()           # odczyt licznika modyfikuje ten licznik!
Liczba utworzonych instancji: 4
```

Niestety, jak wspomniałem wcześniej, tego typu podejście nie sprawdza się, jeśli nie mamy pod ręką gotowej instancji, ponieważ utworzenie nowej spowoduje zmianę licznika instancji zapisanego w klasie, co demonstruje ostatnie wywołanie przykładu. Lepsze rozwiązanie polega na oznaczeniu metody w klasie jako specjalnego przypadku, nieoczekującego obiektu instancji. Szczegóły poniżej.

Używanie metod statycznych i metod klas

W najnowszych wersjach Pythona istnieje jeszcze jedna opcja pozwalająca tworzyć proste funkcje skojarzone z klasami, które mogą być wywoływane z klasy lub instancji. Od Pythona 2.2 możemy tworzyć klasy wyposażone w metody statyczne i metody klas. Żadna z tych specjalnych odmian metod nie wymaga przekazania instancji. Metody tych typów muszą być oznaczone za pomocą wywołania funkcji wbudowanej `staticmethod` lub `classmethod` (wspomniałem o tym przy okazji omawiania klas w nowym stylu). Obie z tych funkcji oznaczają metodę jako specjalną, to znaczy niewymagającą przekazania instancji w przypadku metod statycznych oraz wymagającą przekazania klasy w przypadku metod klas.

```

class Methods:
    def imeth(self, x):          # Zwykła metoda instancji: otrzymuje self
        print(self, x)

    def smeth(x):                # Metoda statyczna: instancja nie jest przekazywana
        print(x)

    def cmeth(cls, x):           # Metoda klasy: otrzymuje klasę, nie instancję
        print(cls, x)

smeth = staticmethod(smeth)      # Przekształcenie smeth w metodę statyczną
cmeth = classmethod(cmeth)       # Przekształcenie cmeth w metodę klasy

```

Warto zwrócić uwagę, w jaki sposób dwa ostatnie wyrażenia zmieniają przypisania metod smeth i cmeth. Atrybuty są tworzone i modyfikowane przez przypisania w ciele klasy, zatem tego typu przypisania są w pełni standardowe i pozwalają zmienić pierwotną definicję metod.

Z technicznego punktu widzenia Python pozwala na stosowanie trzech typów metod: *metod instancji*, *metod statycznych* i *metod klas*. Dodatkowo w Pythonie 3.0 umożliwiono stosowanie zwykłych funkcji w ciele klasy, które pełnią rolę metod statycznych w przypadku wywołania z klasy i nie wymagają stosowania specjalnych funkcji przekształcających semantykę metod.

Metody instancji to domyślna, najczęściej stosowana (również w tej książce) forma metod. Metoda instancji musi być wywołana z przekazaniem obiektu instancji w pierwszym argumentem. Jeśli metoda instancji jest wywołana z instancji, Python automatycznie przekaże do niej właśnie tę instancję, jeśli zostanie wywołana z klasy, programista musi ręcznie przekazać instancję (dla uproszczenia pominąłem część importów):

```

>>> obj = Methods()           # Utworzenie instancji

>>> obj.imeth(1)             # Zwykła metoda, wywoływana z instancji
<__main__.Methods object...> 1 # Wywoływana jako imeth(obj, 1)

>>> Methods.imeth(obj, 2)     # Zwykła metoda, wywoływana z klasy
<__main__.Methods object...> 2 # Instancja przekazywana bezpośrednio

```

Metody statyczne wywołuje się bez przekazywania obiektu instancji. W przeciwieństwie do zwykłych funkcji zdefiniowanych poza klasą nazwy metod statycznych są lokalne w przestrzeni nazw klasy i mogą być wykorzystywane w klasach potomnych. Metody, w których nie zdefiniowano argumentu instancji, mogą w 3.0 być wywoływane z klasy, w 2.6 wymagana jest dodatkowa deklaracja staticmethod. W 3.0 staticmethod pozwala wywoływać metody statyczne również z instancji. Innymi słowy, metody statyczne w 2.6 wymagają deklaracji staticmethod, dzięki czemu mogą być wywoływane bez przekazywania instancji z klasy i z instancji. W Pythonie 3.0 metody statyczne wywoływane z klasy nie wymagają wywołania staticmethod, ale w celu ich wywoływanego z instancji już takie wywołanie jest wymagane.

```

>>> Methods.smeth(3)          # Metoda statyczna, wywoływana z klasy
3                                # Instancja nie jest przekazywana ani oczekiwana

>>> obj.smeth(4)              # Metoda statyczna, wywoływana z instancji
4                                # Instancja nie jest przekazywana

```

Metody klas działają podobnie do zwykłych metod, ale Python zamiast instancji przekazuje do nich klasę, niezależnie od tego, czy zostały wywołane z klasy, czy z instancji.

```

>>> Methods.cmeth(5)          # Metoda klasy, wywoływana z klasą
<class '__main__.Methods'> 5 # Wywoływana jako cmeth(Methods, 5)

>>> obj.cmeth(6)              # Metoda klasy, wywoływana z instancją
<class '__main__.Methods'> 6 # Wywoływana jako cmeth(Methods, 6)

```

Zliczanie instancji z użyciem metod statycznych

Wyposażeni w funkcję wbudowaną `staticmethod` możemy pokusić się o ponowną implementację klasy zliczającej swoje instancje. Funkcja `staticmethod` oznacza metodę jako specjalną, dlatego Python nie przekaże jej obiektu instancji.

```
class Spam:  
    numInstances = 0  
    def __init__(self):  
        Spam.numInstances += 1  
    def printNumInstances():  
        print("Liczba instancji:", Spam.numInstances)  
    printNumInstances = staticmethod(printNumInstances)
```

Dzięki zadeklarowaniu metody jako statycznej można ją wywołać z klasy lub z dowolnej instancji i uzyskamy dokładnie taki sam efekt, zarówno w Pythonie 2.6, jak i 3.0.

```
>>> a = Spam()  
>>> b = Spam()  
>>> c = Spam()  
>>> Spam.printNumInstances()          # Wywoływana jak zwykła funkcja  
Number of instances: 3  
>>> a.printNumInstances()            # Instancja nie jest przekazywana  
Number of instances: 3
```

Porównując to z wydzieleniem metody `printNumInstances` poza ciało klasy (co uczyniliśmy wcześniej), użyliśmy specjalnej funkcji `staticmethod`. Dzięki temu nazwa metody pozostała lokalna w klasie (nie będzie powodować konfliktów z innymi nazwami w module), logika klasy jest zdefiniowana w niej samej, a klasy potomne mogą bez przeszkód korzystać z tej metody lub *przesłonić* ją własną, wykorzystując wszelkie zalety dziedziczenia. Zalety te demonstruje poniższy listing.

```
class Sub(Spam):  
    def printNumInstances():           # Przeciążenie metody statycznej  
        print("Coś ekstra...")          # Ale z wywołaniem oryginału  
        Spam.printNumInstances()  
    printNumInstances = staticmethod(printNumInstances)  
  
>>> a = Sub()  
>>> b = Sub()  
>>> a.printNumInstances()          # Wywołanie z instancji klasy potomnej  
Coś ekstra...  
Liczba instancji: 2  
>>> Sub.printNumInstances()        # Wywołanie z samej klasy potomnej  
Coś ekstra...  
Liczba instancji: 2  
>>> Spam.printNumInstances()  
Liczba instancji: 2
```

Klasy potomne mogą dziedziczyć metodę statyczną, mogą ją też przeddefiniować. Metodę taką wywołuje się bez przekazywania instancji, niezależnie od tego, w którym miejscu drzewa dziedziczenia została zdefiniowana.

```
>>> class Other(Spam): pass         # Dziedziczenie metody statycznej  
  
>>> c = Other()  
>>> c.printNumInstances()  
Liczba instancji: 3
```

Zliczanie instancji z metodami klas

Co interesujące, tę samą funkcję zliczania instancji można zaimplementować z użyciem *metod klas*. Poniższy listing daje dokładnie takie same wyniki jak wersja z użyciem metody statycznej, z tą różnicą, że metoda klasy otrzymuje w swoim pierwszym argumentem obiekt klasy, z której została wywołana. Dzięki temu w kodzie klasy nie trzeba kodować nazwy klasy, co czyni ją bardziej modularną.

```
class Spam:  
    numInstances = 0  
    def __init__(self):  
        Spam.numInstances += 1  
    def printNumInstances(cls):  
        print("Liczba instancji:", cls.numInstances)  
    printNumInstances = classmethod(printNumInstances)
```

Ta klasa jest używana w taki sam sposób jak poprzednie wersje, ale jej metoda `printNumInstances`, niezależnie od tego, czy została wywołana z klasy, czy z instancji, otrzymuje w pierwszym argumentem obiekt klasy.

```
>>> a, b = Spam(), Spam()  
>>> a.printNumInstances() # Klasa przekazana w pierwszym argumencie  
Liczba instancji: 2  
>>> Spam.printNumInstances() # Klasa przekazana w pierwszym argumencie  
Liczba instancji: 2
```

W przypadku używania metod klas należy zapamiętać, że zawsze przekazywana jest im klasa znajdująca się na *najniższym* poziomie drzewa dziedziczenia. Ta właściwość może mieć pewne subtelne konsekwencje w przypadku prób modyfikacji danych przekazanej klasy. Weźmy na przykład moduł `test.py`, który zmodyfikujemy w taki sposób, aby metoda `Spam.printNumInstances` wyświetlała również wartość argumentu `cls`.

```
class Spam:  
    numInstances = 0  
    def __init__(self):  
        Spam.numInstances += 1  
    def printNumInstances(cls):  
        print("Liczba instancji:", cls.numInstances, cls)  
    printNumInstances = classmethod(printNumInstances)  
  
class Sub(Spam):  
    def printNumInstances(cls):  
        print("Coś ekstra...", cls) # Przeciążenie metody klasy  
        # Ale z wywołaniem oryginału  
        Spam.printNumInstances()  
    printNumInstances = classmethod(printNumInstances)  
  
class Other(Spam): pass # Dziedziczenie metody klasy
```

Przy wywołaniu metody klasy przekazywana jest klasa najbliższego poziomu, nawet w przypadku klas potomnych, które nie mają zdefiniowanych metod klasy.

```
>>> x, y = Sub(), Spam()  
>>> x.printNumInstances() # Wywołanie z instancji klasy potomnej  
Extra stuff... <class 'test.Sub'>  
Number of instances: 2 <class 'test.Spam'>  
>>> Sub.printNumInstances() # Wywołanie z samej klasy potomnej  
Coś ekstra... <class 'test.Sub'>  
Liczba instancji: 2 <class 'test.Spam'>  
>>> y.printNumInstances()  
Liczba instancji: 2 <class 'test.Spam'>
```

W pierwszym wywołaniu metoda klasy jest wywoływana z instancji klasy Sub, Python przekazuje do niej obiekt klasy Sub. W tym przypadku wszystko działa, jak należy, ponieważ metoda w klasie Sub wywołuje metodę klasy nadzędnej (Spam), dzięki czemu metoda o tej samej nazwie w Spam otrzyma w pierwszym argumencie obiekt klasy Spam. Sprawdźmy jednak, co się stanie, gdy metodę klasy wywołamy z instancji klasy potomnej:

```
>>> z = Other()
>>> z.printNumInstances()
Liczba instancji: 3 <class 'test.Other'>
```

Ostatnie wywołanie przekaże obiekt klasy Other do metody klasy Spam. W naszym przypadku nadal wynik będzie zgodny z oczekiwaniem, ponieważ *odczytywany* licznik jest zdefiniowany w Spam i zostanie odnaleziony w tej klasie. Gdyby jednak metoda usiłowała *zapisać* dane w klasie przekazanej jej w pierwszym argumencie, zapis zostałby wykonany do klasy Other, nie Spam. Z tego powodu na potrzeby przypadków tego typu warto w metodzie klasy Spam jawnie zadeklarować nazwę klasy, w której będą wykonywane zapisy danych, unikając wykorzystania obiektu przekazanego w argumencie wywołania metody klasy.

Zliczanie instancji dla każdej z klas z użyciem metod klas

Z faktu, że metody klas otrzymują obiekt klasy na z najniższego poziomu w drzewie dziedziczenia wynikają następujące właściwości:

- metody *statyczne* i wykorzystanie jawnych nazw klas są lepszą strategią zapisywania danych wspólnych dla wszystkich klas z drzewa.
- metody *klas* są lepszą strategią jeśli dane mają być gromadzone dla każdej klasy indywidualnie.

Przykładem problemu, do którego lepiej nadają się metody klas może być zliczanie instancji *każdej klasy* z drzewa dziedziczenia. W poniższym listingu w klasie najwyższego poziomu zdefiniowano metodę klasy zapisującą informację stanu, która ma być zapisana dla każdej klasy niezależnie. Jest to zachowanie analogiczne do działania metod instancji, które zapisują dane niezależnie w każdej instancji.

```
class Spam:
    numInstances = 0
    def count(cls):
        cls.numInstances += 1
    def __init__(self):
        self.count()
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self):
        Spam.__init__(self)
    # Przeciążenie metody __init__

class Other(Spam):
    numInstances = 0
    # Odziedziczenie metody __init__
    >>> x = Spam()
    >>> y1, y2 = Sub(), Sub()
    >>> z1, z2, z3 = Other(), Other(), Other()
    >>> x.numInstances, y1.numInstances, z1.numInstances
(1, 2, 3)
    >>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```

Metody klas i metody instancji mają dodatkowe zaawansowane role, których nie będziemy omawiać w tym miejscu. Wskazówki dotyczące dalszej lektury w celu pogłębienia wiedzy na ten temat można znaleźć pod koniec rozdziału. W najnowszych wersjach Pythona deklarowanie metod statycznych i metod klas stało się jeszcze prostsze dzięki wprowadzeniu dekoratorów funkcji. Jest to prosty sposób wykonania funkcji na innej funkcji, którego możliwe zastosowania znacznie wykraczają poza deklarację metod statycznych i metod klas. Dekoratory mogą również być w Pythonie 2.6 i 3.0 wykorzystywane do modyfikowania zachowania klas, na przykład do inicjalizacji licznika numInstances. Przykład takiego użycia przedstawia następny punkt rozdziału.

Dekoratory i metaklasy — część 1.

Ponieważ opisana w poprzednim podrozdziale technika z wywołaniem funkcji staticmethod wydawała się niektórym użytkownikom zbyt toporna, dodano nową opcję, która może ją uprościć. *Dekoratory funkcji* (ang. *function decorator*) udostępniają możliwość określania specjalnych trybów operacji dla funkcji, opakowując je w dodatkową warstwę logiki zaimplementowanej jako kolejna funkcja.

Dekoratory funkcji okazują się narzędziami uniwersalnymi — przydają się do dodawania do funkcji dodatkowej logiki, możliwości wykraczają znacznie poza deklarację metody jako statycznej. Można je na przykład wykorzystać do rozszerzenia funkcji za pomocą kodu logującego ich wywołania czy sprawdzającego typy argumentów przekazanych w czasie debugowania. W pewien sposób dekoratory funkcji są podobne do opisanego w rozdziale 30. wzorca projektowego *delegacji*, jednak zaprojektowano je z myślą o rozszerzeniu określonego wywołania funkcji lub metody, a nie całego interfejsu obiektu.

Python udostępnia pewne wbudowane dekoratory dla operacji takich, jak oznaczanie metod statycznych, jednak programiści mogą również samodzielnie tworzyć dowolne dekoratory. Choć nie są one ściśle powiązane z klasami, dekoratory funkcji zdefiniowane przez użytkownika często zapisywane są jako klasy w celu zachowania oryginalnych funkcji wraz z innymi danymi, takimi jak informacje o stanie. W Pythonie 2.6 i 3.0 pojawiło się kolejne rozszerzenie mechanizmu dekoratorów, które teraz można powiązać z całym modelem. Rola dekoratorów klas częściowo pokrywa się z tym, do czego służą *metaklasy*.

Podstawowe informacje o dekoratorach funkcji

Z punktu widzenia składni dekorator funkcji jest rodzajem deklaracji w czasie wykonywania dotyczącej funkcji, która następuje po nim. Dekorator funkcji zapisywany jest w wierszu tuż przed instrukcją def definiującą funkcję lub metodę i składa się z symbolu @, po którym następuje coś, co nazywamy *metafunkcją* — funkcja (lub inny obiekt wywoływalny), która zarządza inną funkcją. Metody statyczne można na przykład tworzyć za pomocą poniższej składni dekoratora.

```
class C:  
    @staticmethod  
    def meth():  
        ...  
        # składnia dekoratora
```

Wewnętrznie składnia ta ma ten sam efekt co poniższa (przekazanie funkcji przez dekorator oraz przypisanie wyniku z powrotem do oryginalnej nazwy).

```

class C:
    def meth():
        ...
    meth = staticmethod(meth)           # Ponowne wiązanie nazwy

```

Dekorator wiąże nazwę metody z wynikiem wywołania dekoratora. W efekcie wywołanie metody powoduje niejawne wcześniejsze wywołanie dekoratora. Dekorator może zwrócić dowolny obiekt, dzięki czemu dekoratory pozwalają przywiązać dodatkową logikę do każdego wywołania metody. Funkcja dekoratora może zwrócić oryginalną funkcję albo nowy obiekt, który ma zapisaną oryginalną funkcję, wywoływaną po uruchomieniu dodatkowej logiki implementowanej w dekoratorze.

Dzięki składni dekoratorów naszą implementację metod statycznych możemy zapisać w następujący sposób. Kod działa tak samo w Pythonie 2.6 i 3.0 (dekorator `classmethod` stosuje się w ten sam sposób).

```

class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1

    @staticmethod
    def printNumInstances():
        print("Liczba utworzonych instancji: ", Spam.numInstances)

a = Spam()
b = Spam()
c = Spam()
Spam.printNumInstances()          # Tera działają wywołania z klas i z instancji!
a.printNumInstances()            # Obydwa wypiszą "Liczba utworzonych instancji: 3"

```

Należy pamiętać, że `staticmethod` jest funkcją wbudowaną. Można jej użyć ze składnią dekoratorów, ponieważ funkcja ta jako swój parametr przyjmuje obiekt uruchamiany (*callable*). Każdej funkcji o takiej właściwości można użyć jako dekoratora, również funkcji napisanej przez użytkownika, co zademonstruje następny punkt.

Przykład dekoratora

Python oferuje spory wybór funkcji wbudowanych, których można używać w charakterze dekoratorów, możemy też pisać własne dekoratory. Z powodu ich wszechstronnego zastosowania dekoratorom poświęcimy cały rozdział w następnej części książki. W tym miejscu posłużymy się prostym przykładem dekoratora w działaniu.

Jak pamiętamy z rozdziału 29., metoda przeciążania operatorów `__call__` implementuje interfejs wywołania funkcji dla instancji klas. Poniższy kod wykorzystuje to w celu zdefiniowania klasy zapisującej funkcję dekoratora w instancji i przechwytującego wywołania do oryginalnej nazwy. Ponieważ jest to klasa, ma również informacje o stanie (licznik wykonanych wywołań).

```

class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args):
        self.calls += 1
        print('call %s to %s' % (self.calls, self.func.__name__))
        self.func(*args)

```

```

@tracer
def spam(a, b, c):
    print(a, b, c)

spam(1, 2, 3)          # Równoważne wywołaniu spam = tracer(spam)
spam('a', 'b', 'c')    # Opakowanie funkcji spam w obiekt dekoratora
spam(4, 5, 6)          # Naprawdę wywołuje obiekt opakowujący tracer
                       # Wywołuje __call__ w klasie
                       # __call__ dodaje logikę i wykonuje oryginalny obiekt

```

Ponieważ funkcja `spam` jest wykonywana przez dekorator `tracer`, kiedy wywołana zostaje oryginalna nazwa `spam`, tak naprawdę uruchomiona zostaje metoda `__call__` z klasy. Metoda ta odlicza i loguje wywołania, a następnie wywołuje oryginalną opakowaną funkcję. Warto zwrócić uwagę na wykorzystanie składni argumentów `*nazwa` w celu spakowania i rozpakowania przekazanych argumentów. Z tego powodu ten dekorator może zostać wykorzystany do opakowania dowolnej funkcji z dowolną liczbą argumentów.

Rezultat jest taki, że do oryginalnej funkcji `spam` dodana zostaje warstwa logiki. Poniżej widać dane wyjściowe tego skryptu — pierwszy wiersz pochodzi z klasy `tracer`, drugi z funkcji `spam`.

```

wywołanie 1 do spam
1 2 3
wywołanie 2 do spam
a b c
wywołanie 3 do spam
4 5 6

```

By lepiej zrozumieć ten mechanizm, warto prześledzić kod tego przykładu. Prezentowany dekorator działa dla każdej funkcji przyjmującej parametry pozycyjne, ale nie zwraca *wyników* dekorowanej funkcji, nie obsługuje *argumentów kluczowych* i nie może być użyty do dekorowania *metod* klas (w skrócie: w przypadku metod do metody `__call__` dekoratora zostanie przekazana tylko instancja śledząca). Jak zobaczymy w części VIII, istnieje wiele sposobów tworzenia dekoratorów funkcji, między innymi zagnieżdżone instrukcje `def`. Niektóre z tych technik lepiej nadają się do obsługi metod niż prezentowana w powyższym przykładzie.

Dekoratory klas i metaklasy

Dekoratory funkcji okazały się tak użyteczne, że w Pythonie 2.6 i 3.0 rozszerzono ich możliwości o współpracę z klasami. *Dekoratory klas* mają działanie zbliżone do dekoratorów funkcji, z tą różnicą, że są uruchamiane na końcu instrukcji `class`, wiążąc wynik swojego wykonania z nazwą klasy. Dzięki temu dekoratory klas mogą być wykorzystane do zarządzania klasami po ich utworzeniu, jak również do opakowywania logiki klas w dodatkową logikę, na przykład do zarządzaniainstancjami. Składnia użycia dekoratora klas jest następująca:

```

def decorator(aClass):
    ...
@decorator
class C:
    ...

```

Powyższy kod jest równoważny następującemu:

```

def decorator(aClass):
    ...
class C:
    ...
C = decorator(C)

```

Dekorator klas może modyfikować klasę, może też zwracać obiekt pośredniczący, który będzie przechwytywał wywołania konstruktora instancji. Za pomocą takiego dekoratora moglibyśmy w następujący sposób zaimplementować kod zaprezentowany w punkcie „Zliczanie instancji dla każdej z klas z użyciem metod klas”:

```
def count(aClass):
    aClass.numInstances = 0
    return aClass                      # Zwracamy samą klasę, nieobiekt opakowujący

@count
class Spam: ...                      # Równoważne wywołaniu Spam = count(Spam)

@count
class Sub(Spam): ...                  # numInstances = 0 nie jest potrzebne

@count
class Other(Spam): ...
```

Metaklasy są inną formą zaawansowanej mechaniki modyfikującej działanie klas. Ich funkcje przenikają się w pewnym zakresie z funkcjami dekoratorów klas. Metaklasa tworzy alternatywny model przejmujący procedurę konstrukcji obiektu instancji, przekazując ją do klasy potomnej klasy type.

```
class Meta(type):
    def __new__(meta, classname, supers, classdict):
        ...
class C(metaclass=Meta):
    ...

W Pythonie 2.6 stosuje się odmienną składnię: zamiast parametru słownikowego metaclass w deklaracji klasy należy w jej ciele zadeklarować atrybut klasy o nazwie __metaclass__.
```

```
class C:
```

```
    __metaclass__ = Meta
```

```
    ...
```

Metaklasy najczęściej implementują dwie metody klasy type: __new__ oraz __init__, przejmując kontrolę nad procedurą tworzenia i inicjalizacji nowej instancji klasy. W efekcie, podobnie jak ma to miejsce w przypadku dekoratorów klas, otrzymujemy nową logikę uruchamianą automatycznie na etapie tworzenia klasy. Obydwa mechanizmy służą do modyfikowania działania klas, mogą też zwracać dowolne obiekty zastępujące klasy. Ten interfejs daje praktycznie nieograniczone możliwości w zakresie zarządzania klasami.

Dalsza lektura

Oczywiście dekoratory i metaklasy są zagadnieniami znacznie bardziej zaawansowanymi, niż udało mi się je tu przedstawić. Są pomyślane jako mechanizm bardzo ogólny, ale ich głównymi odbiorcami są twórcy narzędzi programistycznych, nie autorzy aplikacji. Z tego powodu szczegóły na ich temat odłożymy do ostatniej części książki:

- rozdział 37. demonstruje zasady tworzenia właściwości z wykorzystaniem składni dekoratora funkcji,
- rozdział 38. zawiera dalsze szczegóły dotyczące dekoratorów, w tym bardziej zaawansowane przykłady,
- rozdział 39. omawia metaklasy i pogłębia temat zarządzania klasami iinstancjami.

Wspomniane wyżej rozdziały omawiają bardzo zaawansowane zagadnienia, ale zawierają też dość szczegółowe przykłady zastosowań Pythona.

Pułapki związane z klasami

Większość problemów z klasami sprowadza się zazwyczaj do kwestii związanych z przestrzeniami nazw (co ma sens, biorąc pod uwagę to, że klasy są po prostu przestrzeniami nazw z kilkoma dodatkowymi sztuczkami). Część zagadnień omawianych w niniejszym podręczniku to raczej studia przypadku zaawansowanego zastosowania klas niż problemy, a jedna lub dwie z tych pułapek zostały wygładzone przez nowsze wydania Pythona.

Modyfikacja atrybutów klas może mieć efekty uboczne

Teoretycznie klasy (oraz instancje klas) są obiektami *zmiennymi*. Podobnie jak wbudowane listy oraz słowniki, mogą być modyfikowane w miejscu przez przypisanie do ich atrybutów. Tak samo jak w przypadku list oraz słowników, oznacza to, że modyfikacja obiektu klasy lub instancji może mieć wpływ na większą liczbę referencji do nich.

Zazwyczaj tego właśnie oczekujemy (w taki sposób obiekty w ogóle zmieniają swój stan), jednak świadomość tego faktu jest kwestią kluczową, kiedy modyfikuje się atrybuty klas. Ponieważ wszystkie instancje wygenerowane z klasy współdzielą przestrzeń nazw klasy, wszelkie zmiany na poziomie klasy są odzwierciedlane we wszystkich instancjach, o ile nie mają one własnych wersji zmodyfikowanych atrybutów klas.

Ponieważ klasy, moduły oraz instancje są tylko obiektami z przestrzeniami nazw atrybutów, normalnie możemy modyfikować ich atrybuty w czasie wykonywania za pomocą przypisania. Rozważmy na przykład poniższą klasę. Wewnątrz ciała klasy przypisanie do zmiennej generuje atrybut `X.a`, istniejący w obiekcie klasy w czasie wykonywania, który zostanie odziedziczony przez wszystkie instancje klasy `X`.

```
>>> class X:  
...     a = 1  
...  
>>> I = X()  
>>> I.a  
1  
>>> X.a  
1
```

Jak na razie wszystko jest w porządku — sytuacja jest zupełnie normalna. Zauważmy jednak, co się stanie, kiedy zmodyfikujemy atrybut klasy w sposób dynamiczny poza instrukcją `class` — modyfikuje to atrybut także w każdej instancji dziedziczącej po tej klasie. Co więcej, nowe instancje utworzone z tej klasy podczas tej sesji czy wykonywania programu otrzymują wartość ustawioną dynamicznie, bez względu na to, co mówi kod źródłowy klasy.

```
>>> X.a = 2  
>>> I.a  
2  
>>> J = X()  
>>> J.a  
2
```

```
# Może zmienić coś więcej niż tylko X  
# I również się zmienia  
# J dziedziczy po X wartości w czasie wykonywania  
# (przypisanie do J.a zmienia a w J, a nie w X lub I)
```

Czy jest to przydatna opcja, czy też niebezpieczna pułapka? Sami musimy to ocenić. Jak mieliśmy okazję przekonać się w rozdziale 26., możemy tak naprawdę wykonać swoją pracę, modyfikując atrybuty klasy i nie tworząc nigdy żadnej instancji. Technika ta może symulować „rekordy” lub „struktury” z innych języków programowania. Jako powtórkę rozważmy poniższy, dość niezwykły, ale całkowicie poprawny program.

```
class X: pass  
class Y: pass  
  
X.a = 1  
X.b = 2  
X.c = 3  
Y.a = X.a + X.b + X.c  
  
for X.i in range(Y.a): print X.i
```

Utworzenie kilku przestrzeni nazw atrybutów
Wykorzystanie atrybutów klas jako zmiennych
Nigdzie nie ma instancji
Wyświetla 0.5

W powyższym kodzie klasy X oraz Y działają jak moduły bez plików — przestrzeń nazw służące do przechowywania zmiennych, których konfliktu wolelibyśmy uniknąć. Jest to całkowicie poprawna sztuczka programistyczna w Pythonie, która jednak jest mniej właściwa, kiedy stosuje się ją do klas utworzonych przez inne osoby. Nie zawsze możemy być pewni, że modyfikowane atrybuty klas nie są krytycznym elementem wewnętrznego zachowania klasy. Jeśli chcemy symulować struktury z języka C, być może lepiej będzie modyfikować instancje niż klasy, ponieważ w ten sposób ma to wpływ jedynie na jeden obiekt.

```
class Record: pass  
X = Record()  
X.name = 'robert'  
X.job = 'Twórca pizzy'
```

Modyfikowanie mutowalnych atrybutów klas również może mieć efekty uboczne

Ta pułapka to w zasadzie rozbudowana wersja poprzedniej. Atrybuty klas są współdzielone przez wszystkie instancje. Jeśli atrybut klasy zawiera referencję do obiektu mutowalnego, modyfikacja tego obiektu z poziomu instancji będzie miała wpływ na wszystkie instancje klasy.

```
>>> class C:  
...     shared = []          # Atrybut klasy  
...     def __init__(self):  
...         self.perobj = []    # Atrybut instancji  
...  
>>> x = C()                # Dwie instancje  
>>> y = C()                # Atrybuty klasy są współdzielone  
>>> y.shared, y.perobj  
([], [])  
  
>>> x.shared.append('spam') # Również modyfikuje zachowanie instancji y!  
>>> x.perobj.append('spam') # Modyfikuje zachowanie instancji x  
>>> x.shared, x.perobj  
(['spam'], ['spam'])  
  
>>> y.shared, y.perobj  
(['spam'], [])              # y widzi zmiany wprowadzone w x  
>>> C.shared  
# Zapisane w klasie i współdzielone  
['spam']
```

Ten efekt nie różni się zbytnio od zjawisk, które poznaliśmy w tej książce: obiekty mutowalne są współdzielone przez zwykłe zmienne, zmienne globalne są współdzielone przez funkcje, obiekty zdefiniowane w modułach są współdzielone przez moduły je importujące, mutowalne argumenty funkcji są współdzielone przez kod wywołujący i kod wywoływany. Wszystko to są przykłady wielokrotnych referencji do obiektów mutowalnych i wszystkie one powodują efekty uboczne, jeśli obiekt mutowalny zostanie zmodyfikowany w jednej z referencji. W tym przypadku mówimy o atrybutach klas dziedziczonych przez instancje, ale nadal chodzi o to samo zjawisko. Efekt będzie zależny od tego, czy modyfikowany jest obiekt wskazany przez referencję, czy sama referencja:

```
x.shared.append('spam') # Modyfikuje współdzielony obiekt przypisany atrybutowi klasy  
x.shared = 'spam'        # Modyfikuje lub tworzy nowy atrybut instancji przez przypisanie
```

Należy pamiętać, że omawiane efekty uboczne nie wynikają z błędów w implementacji języka. Są to subtelności jego semantyki, których należy mieć świadomość, ale które mogą mieć zupełnie legalne zastosowania w programach w Pythonie.

Dziedziczenie wielokrotne — kolejność ma znaczenie

Może to być oczywiste, ale warto to podkreślić jeszcze raz. Jeśli korzystamy z dziedziczenia wielokrotnego, kolejność podawania klas nadrzędnych w nagłówku instrukcji `class` może być kwestią kluczową. Python zawsze przeszukuje klasy nadrzędne od lewej do prawej strony, zgodnie z ich kolejnością w wierszu nagłówka.

Weźmy przykład wielokrotnego dziedziczenia przedstawiony w rozdziale 30. Założymy, że dodatkowo zaimplementowaliśmy metodę `__str__` w klasie `Super`.

```
class ListTree:  
    def __str__(self): ...  
  
class Super:  
    def __str__(self): ...  
  
class Sub(ListTree, Super):      # Wymuszały użycie metody __str__ z klasy ListTree  
    x = Sub()                      # Mechanizm dziedziczenia znajdzie metodę w ListTree zanim przeszuka Super
```

Z której klasy będziemy dziedziczyć: `ListTree` czy `Super`? Wyszukiwanie nazw odbywa się od lewej do prawej, zatem używana będzie metoda tej klasy, która jest zadeklarowana wcześniej (bardziej po lewej) w nagłówku klasy `Sub`. Założymy, że chcemy użyć metody `__str__` z klasy `ListTree`, dlatego zadeklarujemy ją jako pierwszą (w przedstawionym w rozdziale 30. przykładzie klasa `tkinter.Button` miała zdefiniowaną własną metodę `__str__`).

Założymy jednak, że `Super` i `ListTree` posiadają inne atrybuty o takich samych nazwach, których chcielibyśmy użyć. Jeśli chcemy użyć jednej nazwy z `Super`, a innej z `ListTree`, nie pomoże zmiana kolejności deklaracji klas nadrzędnych w klasie `Sub`. Możemy jednak ręcznie zadeklarować każdy z tych atrybutów w klasie `Sub`:

```
class ListTree:  
    def __str__(self): ...  
    def other(self): ...  
  
class Super:  
    def __str__(self): ...  
    def other(self): ...
```

```

class Sub(ListTree, Super):
    other = Super.other
    def __init__(self):
        ...
x = Sub()                                     # Mechanizm dziedziczenia przeszuka klasę Sub przed ListTree/Super

```

W powyższym kodzie przypisanie do zmiennej `other` wewnątrz klasy `Sub` tworzy `Sub`.
`→other` — referencję z powrotem do obiektu `Super.other`. Ponieważ zmienna ta znajduje się
niżej w drzewie, `Sub.other` w rezultacie ukrywa `ListTree.other`, czyli atrybut, który znalazłoby
normalne wyszukiwanie dziedziczenia. I podobnie, gdybyśmy podali `Super` jako pierwszą
w nagłówku klasy, tak by wybrać atrybut `other` z tej właśnie klasy, musielibyśmy ręcznie wybrać
metodę z klasy `ListTree`.

```

class Sub(Super, Lister):
    __repr__ = Lister.__repr__                  # Pobranie other z klasy Super dzięki kolejności
                                                # Jawnym wyborem Lister.__repr__

```

Dziedziczenie wielokrotne jest narzędziem zaawansowanym. Nawet jeśli rozumiemy poprzedni
akapit, lepiej jest korzystać z dziedziczenia wielokrotnego oszczędnie i ostrożnie. W przeciwnym
razie znaczenie nazwy w naszym kodzie może zależeć od kolejności, w jakiej klasy są
mieszane w dowolnie odległej klasie podzielonej. Inny przykład zaprezentowanej tu techniki
wymuszania atrybutów można znaleźć w omówieniu jawnego rozwiązywania konfliktów
w podrozdziale „Klasy w nowym stylu” zamieszczonym wcześniej w niniejszym rozdziale.

Z reguły dziedziczenie wielokrotne działa najlepiej, kiedy nasze klasy mieszane są na tyle
samodzielne, na ile jest to możliwe. Ponieważ mogą być wykorzystywane w wielu różnych
kontekstach, nie powinny czynić żadnych założeń dotyczących zmiennych znajdujących się
w innych klasach drzewa. Omówiona w rozdziale 30. opcja atrybutów pseudoprivaćnych
`__X` może pomóc uczynić nazwy, na których klasa polega, bardziej lokalnymi, a także ograniczyć
zmienne dodawane przez klasy mieszane. W tym przykładzie, jeśli na przykład klasa
`ListTree` chce eksportować jedynie własną metodę `__str__`, drugą swoją metodę może nazwać
`__other`, by uniknąć konfliktów z innymi klasami.

Metody, klasy oraz zakresy zagnieżdżone

Ta pułapka zniknęła w Pythonie 2.2 wraz z wprowadzeniem zakresów funkcji zagnieżdżonych, jednak przedstawiam ją tutaj ze względu na perspektywę historyczną, a także dla osób pracujących ze starszymi wersjami Pythona. Demonstruje ona również, co dzieje się z nowymi regułami zakresu funkcji zagnieżdżonych, kiedy jedną z warstw zagnieżdżenia jest klasa.

Klasy wprowadzają zakresy lokalne, podobnie jak funkcje, dlatego takie samo zachowanie może mieć miejsce w ciele instrukcji `class`. Co więcej, metody są funkcjami jeszcze bardziej zagnieżdżonymi, dlatego mają do nich zastosowanie te same kwestie. Można się pomylić, zwłaszcza wtedy, gdy zagnieżdżane są klasy.

W poniższym przykładzie (plik `nester.py`) funkcja `generate` zwraca instancję zagnieżdżonej klasy `Spam`. Wewnątrz jej kodu nazwa klasy `Spam` przypisywana jest w zakresie lokalnym funkcji `generate`. W wersjach Pythona poprzedzających 2.2 wewnątrz funkcji `method` klasy nazwa klasy `Spam` nie była widoczna — funkcja `method` miała dostęp jedynie do swojego zakresu lokalnego, do modułu otaczającego funkcję `generate` oraz nazw wbudowanych.

```

def generate():                                # Nie zadziała w wersjach wcześniejszych od 2.2
    class Spam:
        count = 1

```

```

def method(self):
    print(Spam.count)
return Spam()                                     # Nazwa Spam nie jest widoczna:
                                                    # nie jest lokalna (def), globalna (moduł), ani wbudowana

generate().method()

C:\python\examples> python nester.py
...pominięty fragment komunikatu o błędzie...

Print(Spam.count)                                # nie jest lokalna (def), globalna (moduł), ani wbudowana
NameError: Spam

```

Przykład ten działa w Pythonie 2.2 oraz nowszych wersjach, ponieważ zakresy lokalne wszystkich instrukcji `def` zawierających funkcję są automatycznie widoczne dla `def` zagnieżdżonych (w tym zagnieżdżonych `def` metod, jak w powyższym kodzie). Nie działa jednak w wersjach wcześniejszych od 2.2 (poniżej znajduje się kilka propozycji rozwiązania tego problemu).

Warto zauważyć, że nawet w wersji 2.2 instrukcje `def` nie widzą zakresu lokalnego klasy zawierającej — widzą jedynie zakresy lokalne zawierających instrukcji `def`. Z tego powodu metody muszą przechodzić przez instancję `self` lub nazwę klasy, by móc się odnieść do metod oraz innych atrybutów zdefiniowanych w obejmującej je instrukcji `class`. Kod metody musiał na przykład używać `self.count` lub `Spam.count` zamiast po prostu `count`.

Jeśli korzystamy z wersji Pythona starszej od 2.2, istnieje kilka sposobów zmuszenia poprzedniego przykładu do działania. Jednym z najprostszych jest przeniesienie nazwy `Spam` do zakresu modułu zawierającego funkcję za pomocą deklaracji `global`. Ponieważ metoda widzi zmienne globalne zawierającego ją modułu, referencje do `Spam` będą w ten sposób działały.

```

def generate():
    global Spam
    class Spam:
        count = 1
        def method(self):
            print Spam.count
    return Spam()

generate().method()                               # Wyświetla 1
                                                    # Zmuszenie Spam do istnienia w zakresie modułu
                                                    # Działa: w zakresie globalnym (modułu)

```

Lepszym rozwiązaniem byłaby zmiana struktury kodu w taki sposób, by klasa `Spam` była zdefiniowana na najwyższym poziomie modułu przez jej poziom zagnieżdżenia, zamiast używać do tego deklaracji `global`. Zagnieżdżona funkcja `method` oraz funkcja najwyższego poziomu `generate` odnajdą następnie klasę `Spam` w swoich zakresach globalnych.

```

def generate():
    return Spam()

class Spam:                                       # Zdefiniowana na najwyższym poziomie modułu
    count = 1
    def method(self):
        print(Spam.count)                         # Działa: w zakresie globalnym (modułu)

generate().method()

```

Tak naprawdę takie rozwiązanie jest zalecane we wszystkich wydaniach Pythona — kod jest prostszy, jeśli unikamy zagnieżdżania klas oraz funkcji.

Jeśli chcemy jednak, by wszystko było skomplikowane i podchwytliwe, możemy się również całkowicie pozbyć referencji do klasy `Spam` w `method`, wykorzystując specjalny atrybut `__class__` zwracający obiekt klasy instancji.

```

def generate():
    class Spam:
        count = 1
        def method(self):
            print(self.__class__.count)      # Działa: kwalifikacja, by dotrzeć do klasy
    return Spam()

generate().method()

```

Klasy wykorzystujące delegację w 3.0 — `__getattr__` i funkcje wbudowane

Z tym zagadnieniem spotkaliśmy się w rozdziale 27. przy okazji omawiania podstaw użycia klas oraz w rozdziale 30. przy omawianiu delegacji. Klasy wykorzystujące metodę przeciążającą operator `__getattr__` w celu delegowania operacji pobierania atrybutu do obiektu opakowującego nie będą działać w 3.0, chyba że przeciążenie operatora zostanie zdefiniowane również w klasie opakowującej. W Pythonie 3.0 (i w 2.6 w przypadku użycia klas w nowym stylu) nazwy metod przeciążających operatory nie są wyszukiwane z użyciem wbudowanych mechanizmów wyszukiwania nazw. Na przykład wyszukiwanie metody `__str__` używanej do wyświetlania obiektu nigdy nie spowoduje wywołania metody `__getattr__`. Zamiast tego Python 3.0 wyszukuje nazwy tego typu bezpośrednio w klasach, całkowicie pomijając mechanizmy wyszukiwania nazw w instancjach. Aby wymusić pożądane zachowanie, metody specjalne muszą być zdefiniowane w klasach opakowujących bezpośrednio, za pomocą narzędzi pomocniczych lub przez definicje w klasach nadzędnych. Więcej szczegółów na temat tych technik poznamy w rozdziałach 37. i 38.

Przesadne opakowywanie

Kiedy się je dobrze stosuje, możliwości ponownego wykorzystania kodu w programowaniu zorientowanym obiektywem potrafią znacznie skrócić czas programowania. Czasami jednak potencjału abstrakcyjnego programowania zorientowanego obiektywem można nadużyć, tak że kod staje się trudny do zrozumienia. By zrozumieć, co robi jakaś operacja, czasami konieczne staje się przeszukanie wielu klas.

Pracowałem kiedyś w firmie programującej w języku C++, która stosowała tysiące klas (część z nich była generowana automatycznie) i do piętnastu poziomów dziedziczenia. Rozszerzanie wywołań metod w tak skomplikowanych systemach było często monumentalnym zadaniem — nawet dla najbardziej podstawowych operacji konieczne było sprawdzanie wielu klas. Logika systemu była tak naprawdę tak głęboko opakowana, że zrozumienie fragmentu kodu wymagało w niektórych przypadkach całych dni przekopywania się przez powiązane z nim pliki.

Tutaj również ma zastosowanie najważniejsza reguła programowania w Pythonie — nie należy czegoś komplikować, o ile nie jest to naprawdę konieczne. Opakowanie kodu w większą liczbę warstw klas w taki sposób, by stał się on niezrozumiały, zawsze jest złym pomysłem. Abstrakcja jest podstawą polimorfizmu oraz hermetyzacji i może być bardzo skutecznym narzędziem, kiedy się ją dobrze wykorzystuje. Ułatwimy sobie jednak debugowanie i utrzymywanie kodu, jeśli interfejsy naszych klas będą intuicyjne, będącymi unikalną nadmiernej abstrakcją kodu i utrzymywając hierarchie klas krótkie oraz płaskie, o ile nie mamy istotnego powodu, by robić inaczej.

Podsumowanie rozdziału

Niniejszy rozdział przedstawił kilka zaawansowanych zagadnień związanych z klasami, w tym tworzenie klas podrzędnych typów wbudowanych, klasy w nowym stylu, metody statyczne oraz dekoratory funkcji. Większość tych elementów to opcjonalne rozszerzenia modelu programowania zorientowanego obiektywem w Pythonie, jednak mogą one stać się bardziej użyteczne w miarę pisania większych programów zorientowanych obiektywem. Jak wspomniałem wcześniej, nasza dyskusja dotycząca bardziej zaawansowanych narzędzi programowania obiektywego będzie kontynuowana w ostatniej części książki. Tam znajdują się omówienia zaawansowanych technik, jak właściwości, deskryptory, dekoratory i metaklasy.

Jest to koniec szóstej części książki poświęconej klasom, dlatego na końcu rozdziału znajduje się zbiór ćwiczeń praktycznych — należy koniecznie się z nimi zapoznać w celu zdobycia doświadczenia w pisaniu kodu prawdziwych klas. W kolejnym rozdziale zacznijmy przyglądać się ostatniemu podstawowemu elementowi języka, czyli wyjątkom. Wyjątki są w Pythonie mechanizmem służącym do komunikowania błędów oraz innych warunków kodu. Jest to stosunkowo lekki temat, jednak zachowałem go na koniec, ponieważ wyjątki powinny obecnie być zapisywane jako klasy. Zanim jednak przejdziemy do tego zagadnienia, warto zająć się quizem podsumowującym niniejszy rozdział oraz ćwiczeniami kończącymi tę część książki.

Sprawdź swoją wiedzę — quiz

1. Należy podać dwa sposoby rozszerzania wbudowanego typu obiektu.
2. Do czego wykorzystywane są dekoratory funkcji?
3. W jaki sposób tworzy się kod klas w nowym stylu?
4. Czym różnią się od siebie klasy z modelu klasycznego oraz klasy w nowym stylu?
5. Czym różnią się od siebie metody normalne i statyczne?
6. Do ilu powinniśmy zliczyć przed rzuceniem Świętego Granatu Ręcznego?

Sprawdź swoją wiedzę — odpowiedzi

1. Możemy osadzić wbudowany obiekt w klasie opakowującej lub bezpośrednio utworzyć jego klasę podrzędną. To drugie rozwiązanie zazwyczaj jest prostsze, gdyż automatycznie dziedziczona jest większość oryginalnego zachowania.
2. Dekoratory funkcji są zazwyczaj wykorzystywane w celu dodania do istniejącej funkcji nowej warstwy logiki, która jest wykonywana za każdym wywołaniem funkcji. Mogą być używane do logowania lub zliczania wywołań funkcji czy sprawdzania jej typów argumentów. Są również wykorzystywane do „deklarowania” metod statycznych — prostych funkcji w klasach, do których nie przekazuje się instancji.
3. Klasy w nowym stylu tworzy się poprzez dziedziczenie po wbudowanej klasie `object` (lub dowolnym innym typie wbudowanym). W Pythonie 3.0 wszystkie klasy są tworzone w nowym stylu. W 2.6 klasy jawnie dziedziczące po typie wbudowanym lub klasie `object` są klasami w nowym stylu, pozostałe są klasami „klasycznymi”.

4. Klasy w nowym stylu inaczej przeszukują drzewa z wielokrotnym dziedziczeniem po jednej klasie nadzędnej (model diamentowy) — najpierw szukają na szerokość zamiast na głębokość (w góre). Klasy w nowym stylu zwracają inne wyniki funkcji wbudowanej `type` dla instancji i klas i nie wykorzystują uogólnionych metod pobierania atrybutów (jak `__getattr__`) dla operacji wbudowanych, obsługują również zbiór dodatkowych, zaawansowanych narzędzi, w tym właściwości oraz listę atrybutów instancji `__slots__`.
5. Zwykłe metody (instancji) otrzymują argument `self` (domniemaną instancję), natomiast w przypadku metod statycznych tak nie jest. Metody statyczne są prostymi funkcjami zagnieżdżonymi w obiekcie klasy. By metoda stała się statyczna, trzeba ją przekształcić za pomocą specjalnej funkcji wbudowanej lub użyć składni dekoratora.
6. Trzy sekundy. A dokładniej: „I Pan powiedział: — Wpierw wyjąć musisz Świętą Zawleczkę. Potem masz zliczyć do trzech, nie mniej, nie więcej. Trzy ma być liczbą, do której liczyć masz, i liczbą tą ma być trzy. Do czterech nie wolno ci liczyć ani do dwóch. Masz tylko policzyć do trzech. Pięć jest wykluczone. Gdy liczba trzy jako trzecia w kolejności osiągnięta zostanie, wówczas rzucić masz Święty Granat Ręczny z Antiochii w kierunku wroga, co naigrywał się z ciebie w polu widzenia twoego, a on kitę odwali”¹.

Sprawdź swoją wiedzę — ćwiczenia do części szóstej

Poniższe ćwiczenia będą wymagały samodzielnego napisania kilku klas oraz poeksperymentowania z istniejącym kodem. Problem istniejącego kodu polega oczywiście na tym, że musi on istnieć. By pracować z klasą zbioru z ćwiczenia 5., można albo pobrać kod źródłowy z Internetu (zgodnie z informacjami z Przedmowy), albo wpisać go ręcznie (jest on stosunkowo krótki). Programy te zaczynają być coraz bardziej zaawansowane, dlatego należy koniecznie sprawdzić rozwiązania znajdujące się na końcu książki w celu odnalezienia wskazówek. Można je znaleźć w podrozdziale „Część VI Klasy i programowanie zorientowane obiektowo” w dodatku B.

1. **Dziedziczenie.** Należy napisać klasę o nazwie `Adder`, która eksportuje metodę `add(self, x, y)` wyświetlającą komunikat: „Nie zaimplementowano”. Następnie należy zdefiniować dla niej dwie klasy podrzędne implementujące metodę `add`:

`ListAdder`

Z metodą `add` zwracającą konkatenację dwóch argumentów będących listami.

`DictAdder`

Z metodą `add` zwracającą nowy słownik zawierający elementy znajdujące się w obydwu słownikach argumentów (wystarczy dowolna definicja dodawania).

Należy poeksperymentować, tworząc instancje wszystkich trzech klas w sesji interaktywnej i wywołując ich metody `add`.

Następnie należy rozszerzyć klasę nadzijną `Adder`, tak by zapisywała obiekt w instancji za pomocą konstruktora (na przykład przypisując `self.data` do listy lub słownika) i przejęła operator `+` za pomocą metody `__add__`, tak by automatycznie przesyłać argumenty do metod `add` (na przykład `X + Y` ma wywoływać `X.add(X.data, Y)`). Jakie jest

¹ Cytat pochodzi z filmu *Monty Python i Święty Graal*. Polskie tłumaczenie w wersji Tomasza Beksińskiego, udostępnione w serwisie *Monty Python's Modrzew* (<http://www.modrzew.stopklatka.pl/>) — przyp. tłum.

najlepsze miejsce na umieszczenie konstruktorów oraz metod przeciążania operatorów (to znaczy w których klasach powinny się one znaleźć?) Jakie rodzaje obiektów można dodać do instancji klas?

W praktyce łatwiejsze może okazać się zapisanie kodu metod add w taki sposób, by przyjmowały one tylko jeden prawdziwy argument (na przykład `add(self, y)`) i dodawały ten argument do bieżących danych instancji (na przykład `self.data + y`). Czy ma to większy sens od przekazania dwóch argumentów do metody add? Czy klasy stają się w ten sposób bardziej zorientowane obiektywne?

2. *Przeciążanie operatorów.* Należy napisać klasę Mylist opakowującą listę Pythona. Powinna ona przeciążać większość operatorów oraz operacji listy, w tym +, indeksowanie, iterację, wycinki oraz metody listy, takie jak append oraz sort. Listę wszystkich możliwych metod, jakie można obsługiwać, można znaleźć w dokumentacji Pythona. Należy również udostępnić konstruktor dla tej klasy, który tworzy istniejącą listę (lubinstancję Mylist) i kopiuje jej komponenty do składowej instancji. Należy poeksperymentować z tą klasą w sesji interaktywnej. Rzeczy, które należy sprawdzić:
 - a) Dlaczego skopiowanie początkowej wartości jest tutaj tak istotne?
 - b) Czy można wykorzystać pusty wycinek (na przykład `start[:]`) do skopiowania początkowej wartości, jeśli jest to instancja klasy Mylist?
 - c) Czy istnieje jakiś uniwersalny sposób przekierowania wywołań metod listy do opakowanej listy?
 - d) Czy można dodać do siebie instancję klasy Mylist oraz zwykłą listę? A co w przypadku zwykłej listy i instancji Mylist?
 - e) Obiekt jakiego typu powinny zwracać operacje takie, jak + czy wycinki? Co z opercjami indeksowania?
 - f) Jeśli pracujemy w którejś z nowszych wersji Pythona (od 2.2 w góre), możemy zaimplementować ten typ klasy opakowującej, osadzając prawdziwą listę w samodzielnej klasie lub rozszerzając wbudowany typ listy za pomocą klasy podrzędnej. Które rozwiązanie jest prostsze i dlaczego?
3. *Klasy podrzędne.* Należy utworzyć klasę podrzędną dla Mylist z ćwiczenia 2. o nazwie MylistSub. Klasa ta powinna rozszerzać Mylist i wyświetlać komunikat do stdout przed wywołaniem każdej przeciążonej operacji oraz zliczać wywołania. Klasa MylistSub powinna dziedziczyć podstawowe metody zachowania po Mylist. Dodanie sekwencji do MylistSub powinno wyświetlić komunikat, dokonać inkrementacji licznika wywołań + i wykonać metodę klasy nadrzędnej. Należy również dodać nową metodę wyświetlającą liczniki operacji do stdout i poeksperymentować z tą klasą w sesji interaktywnej. Czy liczniki zliczają wywołania na instancję, czy też na klasę (dla wszystkich instancji klasy)? W jaki sposób można w kodzie zaimplementować obie wersje? Wskazówka: jest to uzależnione od tego, do którego obiektu przypisane są składowe licznika — składowe klasy są współdzielone przez instancje, natomiast składowe self są danymi poszczególnych instancji.
4. *Metody atrybutów.* Należy napisać klasę o nazwie Meta z metodami przechwytyującymi każdą kwalifikację atrybutów (zarówno pobrania, jak i przypisania) i wyświetlającymi do stdout komunikaty wymieniające argumenty. Później trzeba utworzyć instancję klasy Meta i poeksperymentować ze składnią kwalifikującą w sesji interaktywnej. Co się dzieje, jeśli próbujemy wykorzystać instancję w wyrażeniach? Należy spróbować dodać, zindek-

sować i wykonać wycinek klasy. (Uwaga: w pełni uogólnione podejście zadziała w 2.6, ale nie w 3.0 — z powodów opisanych w rozdziale 30. i przytoczonych w rozwiążaniu tego ćwiczenia).

5. *Obiekty zbiorów.* Należy poeksperymentować z klasą zbioru opisaną w podrozdziale „Rozszerzanie typów za pomocą osadzania”. Należy wykonać polecenia pozwalające na następujące rodzaje operacji:
- Należy utworzyć dwa zbiory liczb całkowitych i obliczyć ich część wspólną oraz sumę za pomocą wyrażeń z operatorami & oraz |.
 - Należy utworzyć zbiór z łańcucha znaków i poeksperymentować z indeksowaniem tego zbioru. Jakie metody klasy są wywoływanie?
 - Należy za pomocą pętli for spróbować wykonać iterację po elementach zbioru łańcucha znaków. Jakie metody wykonywane są tym razem?
 - Należy spróbować obliczyć część wspólną oraz sumę naszego zbioru łańcucha znaków oraz prostego łańcucha znaków. Czy to zadziała?
 - Teraz należy rozszerzyć zbiór, tworząc klasę podrzędną obsługującą dowolną liczbę argumentów, wykorzystując do tego formę *args. Wskazówka: wersja tych algorytmów przeznaczona dla funkcji znajduje się w rozdziale 18. Należy obliczyć części wspólne oraz sumy większej liczby argumentów za pomocą naszej klasy podrzędnej. W jaki sposób można uzyskać część wspólną trzech lub większej liczby zbiorów, biorąc pod uwagę to, że operator & ma tylko dwie strony?
 - W jaki sposób można emulować inne operacje na listach w klasie zbioru? Wskazówka: metoda `_add_` może przechwytywać konkatenację, a `_getattr_` przekazać większość wywołań metod list do opakowanej listy.
6. *Łącza drzew klas.* W podrozdziałach „Przestrzenie nazw — cała historia” w rozdziale 28. oraz „Dziedziczenie wielokrotne — klasy mieszane” w rozdziale 30. wspomniałem, że klasy mają atrybut `_bases_` zwracający krotkę obiektów ich klas nadrzędnych (tych wymienionych w nawiasach w nagłówku klasy). Należy wykorzystać atrybut `_bases_` do rozszerzenia klas zdefiniowanych w module `lister.py` (rozdział 30.) w taki sposób, by wyświetlała ona nazwy bezpośrednich klas nadrzędnych klasy instancji. W rezultacie pierwszy wiersz reprezentacji łańcuchów znaków powinien wyglądać jak poniższy (adres może być inny):

<Instancja klasy Sub(Super, Lister), adres 7841200:

7. *Kompozycja.* Należy wykonać symulację scenariusza zamawiania jedzenia w barze szybkiej obsługi, definiując cztery klasy:

Lunch

Pojemnik oraz klasa kontrolera.

Customer

Aktor kupujący jedzenie.

Employee

Aktor, u którego klient składa zamówienie.

Food

To, co kupuje klient.

Na początek w poniższym kodzie widoczne są klasy oraz metody, jakie będziemy definiować.

```
class Lunch:  
    def __init__(self)  
    def order(self, foodName)  
    def result(self)  
  
class Customer:  
    def __init__(self)  
    def placeOrder(self, foodName, employee)  
    def printFood(self)  
  
class Employee:  
    def takeOrder(self, foodName)  
  
class Food:  
    def __init__(self, name)
```

Tworzy/osadza klasy Customer oraz Employee
Rozpoczęcie symulacji zamówienia klienta Customer
Zapytanie klienta Customer, jakie ma jedzenie Food
Inicjalizacja mojego jedzenia na None
Złożenie zamówienia pracownikowi Employee
Wyświetlenie nazwy mojego jedzenia
Zwraca jedzenie Food z żądaną nazwą
Przechowanie nazwy jedzenia

Symulacja zamówienia przebiega następująco:

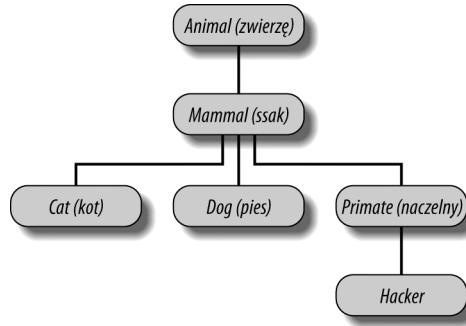
- Konstruktor klasy Lunch powinien utworzyć oraz osadzić instancję klas Customer (klient) oraz Employee (pracownik). Powinien również eksportować metodę o nazwie order. Po wywołaniu metoda order powinna poprosić klienta Customer o złożenie zamówienia, wywołując jego metodę placeOrder. Metoda placeOrder klienta powinna z kolei poprosić obiekt pracownika Employee o nowy obiekt jedzenia Food, wywołując jego metodę takeOrder.
- Obiekty jedzenia Food powinny przechowywać łańcuch znaków z nazwą jedzenia (na przykład „burrito”), przekazany z Lunch.order do Customer.placeOrder, stamtąd do Employee.takeOrder, a na końcu do konstruktora obiektu Food. Klasa najwyższego poziomu Lunch powinna również eksportować metodę o nazwie result, prosząc klienta o wyświetlenie nazwy jedzenia, jakie otrzymał od pracownika za pomocą zamówienia (można wykorzystać to do przetestowania naszej symulacji).

Należy zauważyc, że aby klasa Customer mogła wywoływać metody klasy Employee, klasa Lunch musi przekazać klasę Employee lub samą siebie do klasy Customer.

Należy poeksperymentować z tymi klasami w sesji interaktywnej, importując klasę Lunch, wywołując jej metodę order w celu wykonania interakcji, a następnie wywołując jej metodę result w celu zweryfikowania, że klient Customer otrzymał to, co zamówił. Jeśli wolimy, możemy również zapisać przypadki testowe jako kod samosprawdzający w pliku, w którym klasy zostały zdefiniowane, wykorzystując do tego sztuczkę z nazwą `__name__` modułu z rozdziału 24. W tej symulacji to klasa Customer jest aktywnym agentem. W jaki sposób zmieniłyby się klasy, gdyby to Employee był obiektem inicjalizującym interakcję pomiędzy klientem a pracownikiem?

8. Hierarchia zwierząt w zoo. Należy rozważyć drzewo klas pokazane na rysunku 31.1.

Tym razem trzeba utworzyć sześć instrukcji `class` pozwalających utworzyć model tej taksonomii za pomocą mechanizmu dziedziczenia Pythona. Następnie należy dodać metodę speak do każdej klasy, wyświetlającą unikalny komunikat, oraz metodę reply w klasie nadzędnej najwyższego poziomu Animal, która wywołuje po prostu `self.speak` w celu wywołania komunikatu specyficznego dla tej kategorii w klasie podzędnej znajdującej się niżej (pozwoli to uruchomić wyszukiwanie dziedziczenia niezależne od `self`). Na koniec należy usunąć metodę `self` z klasy Hacker, tak by przyjęła ona metodę domyślną klasy nadzędnej. W rezultacie klasy powinny działać w następujący sposób.

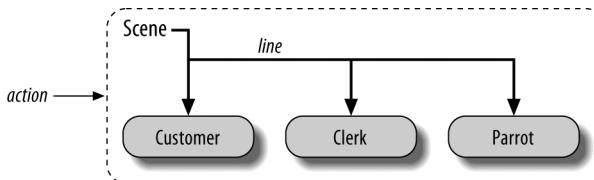


Rysunek 31.1. Hierarchia zwierząt w zoo składająca się z klas połączonych w drzewo, które jest przeszukiwane przez dziedziczenie atrybutów. Zwierzęta mają wspólną metodę reply, jednak każda klasa ma własną metodę speak wywoływaną przez reply

```

% python
>>> from zoo import Cat, Hacker
>>> spot = Cat()
>>> spot.reply()                                # Animal.reply; wywołuje Cat.speak
miau
>>> data = Hacker()
>>> data.reply()                               # Animal.reply; wywołuje Primate.speak
Witaj, świecie!
  
```

9. Skecz z martwą papugą. Rozważmy strukturę osadzania obiektów przedstawioną na rysunku 31.2.



Rysunek 31.2. Kompozyt sceny z klasą kontrolera (Scene) osadzającą instancje trzech innych klas (Customer, Clerk, Parrot) i kierującą nimi. Klasa osadzonych instancji mogą również brać udział w hierarchii dziedziczenia. Kompozycja i dziedziczenie są często tak samo użytecznymi sposobami nadawania kodowi struktury umożliwiającej jego ponowne wykorzystanie

Należy utworzyć kod Pythona implementujący tę strukturę za pomocą kompozycji. Kod obiektu Scene ma definiować metodę action i osadzać instancje klas Customer (klient), Clerk (sprzedawca) oraz Parrot (papuga) — wszystkie trzy powinny definiować metodę line wyświetlającą unikalny komunikat. Osadzone obiekty mogą dziedziczyć po wspólnej klasie nadrzędnej definiującej line i udostępniać komunikat text lub samodzielnie definiować line. W rezultacie klasy powinny działać w następujący sposób.

```

% python
>>> import parrot
>>> parrot.Scene().action()                    # Aktywacja żagnieżdżonych obiektów
customer: "To już ekspapuga!"
clerk: "nie, wcale nie..."
parrot: None
  
```

Programowanie zorientowane obiektowo według mistrzów

Kiedy prowadzę kursy z Pythona, nieodmiennie spotykam się z tym, że w połowie kursu osoby, które używały już programowania zorientowanego obiektowo w przeszłości, z łatwością nadążają za materiałem, natomiast pozostałe osoby zaczynają się nudzić (lub nawet przysypanią). Cel tej technologii nie jest dla nich oczywisty.

W książce takiej, jak ta, mam luksus dołączenia materiału takiego, jak przegląd szerokiej perspektywy z rozdziału 25. czy stopniowe wprowadzenie z rozdziału 27. Szczerze zachęcam każdego, kto zaczyna czuć, że programowanie zorientowane obiektowo to jakieś komputerowe trele-morele, by raz jeszcze powrócił do tego fragmentu książki.

Na moich kursach — by zachęcić osoby poczatkujące (a czasem uniemożliwić im drzemkę) — znany jestem z tego, że zatrzymuję się i pytam ekspertów spośród publiczności, po co używają programowania zorientowanego obiektowo. Odpowiadzi, jakich mi udzielają, mogą pomóc objaśnić nieco cel programowania zorientowanego obiektowo osobom, dla których zagadnienie to jest nowością.

Poniżej, jedynie z drobnymi upiększeniami, znajdują się powody wykorzystywania programowania zorientowanego obiektowo podawane przez moich studentów przez te wszystkie lata.

Ponowne wykorzystanie kodu

Ten powód jest prosty (i jest też najważniejszą przyczyną korzystania z programowania zorientowanego obiektowo). Dzięki obsłudze dziedziczenia klasy pozwalają nam programować za pomocą dostosowywania kodu do własnych potrzeb zamiast rozpoczynania każdego projektu od podstaw.

Hermetyzacja

Opakowywanie szczegółów implementacji za interfejsami obiektów izoluje użytkowników klas od zmian kodu.

Struktura

Klasy udostępniają nowe zakresy lokalne, co pozwala zminimalizować konflikty między nazwami zmiennych. Są również naturalnym miejscem, w którym pisze się i wyszukuje kod implementacyjny, a także gdzie zarządza się stanem obiektów.

Utrzymanie

Klasy w sposób naturalny promują faktoryzację kodu, co pozwala nam na minimalizowanie jego powtarzalności. Dzięki strukturze oraz obsłudze ponownego wykorzystania zazwyczaj zmienić musimy jedynie jedną kopię kodu.

Spójność

Klasy oraz dziedziczenie pozwalają na implementowanie wspólnych interfejsów i tym samym na taki sam wygląd oraz działanie kodu. Ułatwia to debugowanie, zrozumienie kodu oraz jego utrzymanie.

Polimorfizm

To raczej właściwość programowania zorientowanego obiektowo, a nie powód korzystania z niego, ale dzięki obsłudze uogólnienia kodu polimorfizm sprawia, że kod ten staje się bardziej elastyczny i nadający się do różnych zastosowań, dzięki czemu można go wykorzystywać wielokrotnie.

Pozostałe

I na koniec, oczywiście, jeden z powodów wykorzystywania programowania zorientowanego obiektowo podany przez pewnego studenta, który brzmi: ponieważ wygląda to dobrze w naszym CV! No dobrze, powód ten dodałem jako dowcip, jednak jeśli planujemy w dzisiejszych czasach pracować w przemyśle informatycznym, znajomość programowania zorientowanego obiektowo jest istotna.

Należy również pamiętać o tym, co napisałem na początku szóstej części książki — programowanie zorientowane obiektowo docenia się wtedy, gdy się go trochę poużywa. Należy wybrać jakiś projekt, przestudiować większe przykłady, samodzielnie wykonać ćwiczenia — zrobić cokolwiek, co zwiększy nasze obyczcie z kodem zorientowanym obiektowo. Na pewno nam się to opłaci.

Wyjątki oraz narzędzia

Podstawy wyjątków

Niniejsza część książki poświęcona jest *wyjątkom* (ang. *exception*), czyli zdarzeniom, które mogą modyfikować przebieg sterowania w programie. W Pythonie wyjątki wywoływane są自动ycznie w momencie wystąpienia błędów i mogą być wywoływane oraz przechwytywane przez nasz kod. Są przetwarzane przez cztery instrukcje omówione w tej części książki, z których pierwsza ma dwa warianty (omówione tutaj osobno), a ostatnia była rozszerzeniem opcjonalnym aż do Pythona 2.6 oraz 3.0.

`try/except`

Przechwytuje wyjątki wywołane przez Pythona lub przez nas i pozwala sobie z nimi poradzić.

`try/finally`

Wykonuje działania oczyszczające bez względu na to, czy wyjątki wystąpią, czy też nie.

`raise`

Ręczne wywołanie wyjątku w kodzie.

`assert`

Warunkowe wywołanie wyjątku w kodzie.

`with/as`

Implementuje menedżery kontekstu w Pythonie 2.6 oraz 3.0 (w wersji 2.5 opcjonalne).

Temat ten został odłożony aż do prawie samego końca książki, ponieważ do tworzenia kodu własnych wyjątków musimy znać klasy. Z kilkoma wyjątkami (celowa gra słów) — zobaczymy, że obsługa wyjątków w Pythonie jest prosta, ponieważ w dużej mierze jest ona zintegrowana z samym językiem jako kolejne narzędzie wyższego poziomu.

Po co używa się wyjątków?

Mówiąc w skrócie, wyjątki pozwalają nam wyskoczyć z dowolnie dużych części programu. Rozważmy hipotetycznego robota wytwarzającego pizzę, o którym mówiliśmy wcześniej w książce. Założmy, że pomysł ten bierzemy całkiem na poważnie i rzeczywiście zbudujemy taką maszynę. By wytworzyć pizzę, nasz kulinarny automat musi wykonać plan, który musielibyśmy zaimplementować jako program napisany w Pythonie. Musiałby on przyjąć zamówienie, przygotować ciasto, dodać składniki nadzienia, upiec ciasto i tak dalej.

Załóżmy teraz, że coś pójdzie nie tak na etapie „pieczenie ciasta”. Być może piekarnik jest uszkodzony, być może robot źle obliczył swój zasięg i spontanicznie się zapalił. Chcemy oczywiście móc szybko przeskoczyć do kodu obsługującego taki stan. Ponieważ nie mamy już nadziei na skończenie pizzy w tak niezwykłych okolicznościach, równie dobrze możemy porzucić cały plan.

Właśnie na to pozwalają nam wyjątki — możemy w jednym kroku przeskoczyć do programu obsługi wyjątku, porzucając wszystkie wywołania funkcji rozpoczęte od momentu wejścia do programu obsługi wyjątku. Kod z programu obsługi wyjątku może następnie odpowiedzieć w odpowiedni sposób na zgłoszony wyjątek (na przykład wzywając straż pożarną!).

Wyjątek można sobie wyobrazić jako typ ustrukturyzowanej instrukcji „super-goto”. *Program obsługi wyjątków* (instrukcja `try`) pozostawia znacznik i wykonuje jakiś kod. Gdzieś dalej w programie zgłaszany jest wyjątek sprawiający, że Python wraca do znacznika, porzucając wszystkie aktywne funkcje, które były wywołane po opuszczeniu znacznika. Protokół ten udostępnia spójny sposób reagowania na niezwykłe zdarzenia. Co więcej, ponieważ Python natychmiast przeskakuje do instrukcji programu obsługi, kod może być prostszy — zazwyczaj nie ma konieczności sprawdzania kodów statusu po każdym wywołaniu funkcji, która potencjalnie może zawieść.

Role wyjątków

W programach napisanych w Pythonie wyjątki są zazwyczaj wykorzystywane do różnych celów. Poniżej znajduje się spis ich najważniejszych ról.

Obsługa błędów

Python zgłasza wyjątki za każdym razem, kiedy w czasie wykonywania programu znajduje w nim błąd. Możemy przechwytywać i odpowiadać na błędy w kodzie lub ignorować zgłoszone wyjątki. Jeśli błąd jest ignorowany, do gry wkracza domyślna obsługa wyjątków Pythona — zatrzymuje ona program i wyświetla komunikat o błędzie. Jeśli takie zachowanie nam nie odpowiada, musimy zapisać w kodzie instrukcję `try`, która przechwyci wyjątek i pozwoli go obsłużyć. Po wykryciu błędu Python przeskoczy do programu obsługi `try`, a program wznowi wykonywanie po `try`.

Powiadomienia o zdarzeniach

Wyjątki można również wykorzystać do sygnalizowania poprawnych warunków bez konieczności przekazywania flag wyników w programie lub jawnego ich sprawdzania. Procedura wyszukiwania może na przykład zgłosić wyjątek dla niepowodzenia, zamiast zwracać liczbowy kod wyniku (i mieć nadzieję, że kod nigdy nie będzie miał poprawnego wyniku).

Obsługa przypadków specjalnych

Czasami jakiś warunek może występować tak rzadko, że trudno jest uzasadnić przekształcanie kodu w taki sposób, by go obsługiwał. Często możemy wyeliminować kod specjalnych przypadków, obsługując je zamiast tego w programach obsługi wyjątków na wyższych poziomach.

Działania końcowe

Jak zobaczymy, instrukcja `try/finally` pozwala nam zagwarantować, że wymagane operacje czasu zakończenia zostaną wykonane bez względu na obecność lub nieobecność wyjątków w programach.

Niezuwykły przebieg sterowania

I wreszcie, ponieważ wyjątki są rodzajem operacji „*goto*” wysokiego poziomu, możemy ich użyć jako podstawy do implementacji egzotycznego przebiegu programu. Choć na przykład Python oficjalnie nie obsługuje nawracania (ang. *backtracking*), można je w tym języku programowania zaimplementować za pomocą wyjątków, a także niewielkiej ilości logiki obsługującej służącej do rozwinięcia przypisania.¹ W Pythonie nie ma instrukcji „*goto*” (na całe szczęście!), jednak wyjątki mogą czasami pełnić te same role.

Takie typowe zastosowania zobaczymy w tej części książki. Na razie zacznijmy omawianie narzędzi przetwarzających wyjątki Pythona.

Wyjątki w skrócie

W porównaniu z innymi kluczowymi możliwościami języka, z jakimi spotkaliśmy się w książce, wyjątki są w Pythonie dość mało kłopotliwym narzędziem. Ponieważ są tak proste w użyciu, przejdźmy od razu do przykładu.

Domyślny program obsługi wyjątków

Załóżmy, że piszemy kod następującej funkcji.

```
>>> def fetcher(obj, index):
...     return obj[index]
... 
```

W funkcji tej nie ma nic szczególnego — indeksuje ona obiekt dla przekazanej wartości indeksu. W normalnej operacji zwraca ona wynik tej całkowicie poprawnej składni.

```
>>> x = 'mielonka'
>>> fetcher(x, 3)                                # Jak x[3]
'1'
```

Jeśli jednak zażądamy od funkcji zindeksowania wartości przesunięcia przekraczającej długość łańcucha znaków, w czasie próby wykonania `obj[index]` zgłoszony zostanie wyjątek. Python wykrywa indeksowanie poza granicami sekwencji i raportuje je, zgłaszając (wywołując) wbudowany wyjątek `IndexError`.

```
>>> fetcher(x, 8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Domyślny program obsługi wyjątków — interfejs powłoki

Ponieważ nasz kod nie przechwytuje tego wyjątku w sposób jawnny, przechodzi z powrotem do najwyższego poziomu programu i wywołuje *domyślny program obsługi wyjątków*, który po prostu wyświetla standardowy komunikat o błędzie. W tym miejscu książki widzieliśmy już

¹ Prawdziwe nawracanie jest zagadnieniem zaawansowanym, które nie jest częścią języka Python (nawet funkcje i wyrażenia generatorów przedstawione w rozdziale 20. nie są prawdziwym nawracaniem — po prostu odpowiadają one na żądania `next(G)`), dlatego nie będę go tutaj szerzej omawiał. Nawracanie w przybliżeniu przywraca wszystkie obliczenia przed przeskokiem. Wyjątki Pythona tego nie robią (to znaczy zmienne przypisane pomiędzy czasem wejścia do instrukcji `try` a czasem zgłoszenia wyjątku nie są przywracane do wcześniejszych wartości). Osoby zainteresowane tym zagadnieniem odsyłam do książek poświęconych sztucznej inteligencji lub językom Prolog albo Icon.

kilka standardowych komunikatów o błędach. Obejmują one zgłoszony wyjątek wraz ze *śladem stosu* (ang. *stack trace*) — listą wszystkich wierszy oraz funkcji aktywnych w momencie, kiedy nastąpił wyjątek.

Powyższy tekst komunikatu o błędzie został wyświetlony przez Pythona w wersji 3.0. W różnych wydaniach Pythona, a także typach powłoki interaktywnej może się on nieznacznie różnić. Kiedy kod wpisujemy w sesji interaktywnej w prostym interfejsie powłoki, plik to po prostu <stdin>, co oznacza standardowy strumień wejścia. W przypadku pracy w powłoce interaktywnej GUI IDLE nazwą pliku jest <pyshell> i wyświetlane są także wiersze kodu źródłowego. W żadnym z przypadków numerowanie wierszy nie jest zbyt znaczące, jeśli nie mamy pliku (ciekawsze komunikaty o błędach zobaczymy nieco później w tej części książki).

```
>>> fetcher(x, 8)                                # Domyślny program obsługi wyjątków — graficzny interfejs IDLE
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    fetcher(x, 8)
  File "<pyshell#3>", line 2, in fetcher
    return obj[index]
IndexError: string index out of range
```

W bardziej realistycznym programie uruchomionym poza sesją interaktywną po wyświetleniu komunikatu o błędzie domyślny program obsługi na najwyższym poziomie programu natychmiast kończy również program. Takie działanie ma sens dla prostych skryptów — błędy często są krytyczne i najlepsze, co możemy zrobić, kiedy wystąpią, to sprawdzenie standardowego komunikatu o błędzie.

Przechwytywanie wyjątków

Czasami jednak nie o to nam chodzi. Programy serwera zazwyczaj muszą na przykład pozostać aktywne nawet po błędach wewnętrznych. Jeśli nie chcemy stosować domyślnego zachowania wyjątków, powinniśmy opakować wywołanie w instrukcję try w celu samodzielnego ich przechwytywania.

```
>>> try:
...     fetcher(x, 8)
... except IndexError:                         # Przechwycenie i poradzenie sobie
...     print('mam wyjątek')
...
mam wyjątek
>>>
```

Teraz Python przeskakuje do naszego *programu obsługi* (bloku znajdującego się pod częścią except, który nazywa zgłoszany wyjątek) automatycznie, kiedy wyjątek jest wywoływany w czasie wykonywania bloku instrukcji try. Kiedy pracujemy w sesji interaktywnej, jak powyżej, po wykonaniu części except trafiemy z powrotem do zachęty Pythona. W bardziej realistycznym programie instrukcje try nie tylko przechwytyują błędy, ale również pomagają sobie z nimi *poradzić*.

```
>>> def catcher():
...     try:
...         fetcher(x, 8)
...     except IndexError:
...         print('mam wyjątek')
...         print('kontynuuuję')
```

```
>>> catcher()
mam wyjątek
kontynuuuję
>>>
```

Tym razem jednak po przechwyceniu i obsłużeniu wyjątku program kontynuuje wykonywanie po całej instrukcji `try`, która przechwyciła błąd — stąd otrzymujemy komunikat „kontynuuuję”. Nie widzimy standardowego komunikatu o błędzie, a program normalnie kontynuuje swoje działanie.

Zgłaszanie wyjątków

Dotychczas pozwalaliśmy Pythonowi zgłaszać wyjątki za nas, popełniając błędy (tym razem celowo!), jednak nasze skrypty mogą również same zgłaszać wyjątki. Oznacza to, że wyjątki mogą być zgłaszcane przez Pythona lub nasz program; mogą być przechwytywane lub nie. By ręcznie wywołać wyjątek, wystarczy wykonać instrukcję `raise`. Wyjątki wywoływane przez użytkownika są przechwytywane w taki sam sposób jak wywoywane przez Pythona. Poniższy kod może nie być najbardziej użytecznym programem Pythona w historii, ale dobrze ilustruje, o co nam chodzi.

```
>>> try:
...     raise IndexError
... except IndexError:
...     print('mam wyjątek')
...
mam wyjątek
```

Jak zwykle, jeśli wyjątek zgłoszony przez użytkownika nie zostanie przechwycony, jest on przekazywany do domyślnego programu obsługi wyjątków na najwyższym poziomie programu, a program kończy się ze standardowym komunikatem o błędzie.

```
>>> raise IndexError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError
```

Jak zobaczymy w kolejnym rozdziale, do wywołania wyjątku można także wykorzystać instrukcję `assert` — jest ona warunkowym odpowiednikiem `raise`, wykorzystywanym głównie na cele debugowania kodu w trakcie programowania:

```
>>> assert False, 'Nikt nie spodziewa się hiszpańskiej inkwizycji!'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Nikt nie spodziewa się hiszpańskiej inkwizycji!
```

Wyjątki zdefiniowane przez użytkownika

Instrukcja `raise` przedstawiona w poprzednim podrozdziale powoduje zgłoszenie wbudowanego wyjątku zdefiniowanego w zakresie wbudowanym Pythona. Jak dowiemy się w dalszej części książki, możemy również definiować nowe własne wyjątki, specyficzne dla naszego programu. Wyjątki zdefiniowane przez użytkowników można tworzyć za pomocą *klas dziedziczących po wbudowanej klasie wyjątków* — zazwyczaj jest to klasa o nazwie `Exception`. Wyjątki oparte na klasach pozwalają skryptom na budowanie kategorii wyjątków, dziedziczenie zachowania i dołączanie informacji o stanie.

```

>>> class Bad(Exception): # Wyjątek zdefiniowany przez użytkownika
...     pass
...
>>> def doomed(): # Zgłoszenie instancji
...     raise Bad()
...
>>> try: # Przechwycenie nazwy klasy
...     doomed()
... except Bad:
...     print('przechwycenie Bad')
...
przechwycenie Bad
>>>

```

Działania końcowe

Instrukcje `try` mogą wreszcie określać, co stanie się na końcu — zawierając bloki `finally`. Wyglądają one tak jak programy obsługi dla wyjątków `except`, jednak kombinacja `try/finally` określa działania końcowe, jakie zawsze zostaną wykonane przy wychodzeniu, bez względu na to, czy w bloku `try` wystąpi wyjątek.

```

>>> try: # Działania końcowe
...     fetcher(x, 3)
... finally:
...     print('po pobraniu')
...
'1'
po pobraniu
>>>

```

W powyższym kodzie, jeśli blok `try` zakończy się bez wystąpienia wyjątku, blok `finally` nadal zostanie wykonany, a program będzie kontynuowany po całej instrukcji `try`. W tym przypadku instrukcja ta wydaje się dość głupia — równie dobrze moglibyśmy wpisać instrukcję `print` zaraz po wywołaniu funkcji i całkowicie pominąć instrukcję `try`.

```

fetcher(x, 3)
print('po pobraniu')

```

Takie rozwiązanie jest jednak problematyczne. Jeśli wywołanie funkcji zwróci wyjątek, nigdy nie wykonamy instrukcji `print`. Połączenie `try/finally` pozwala uniknąć tej pułapki — jeśli wyjątek wystąpił w bloku `try`, bloki `finally` wykonywane są, kiedy program jest rozwijany.

```

>>> def after():
...     try:
...         fetcher(x, 8)
...     finally:
...         print('po pobraniu')
...         print('po try?')
...
>>> after()
po pobraniu
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in after
    File "<stdin>", line 2, in fetcher
      IndexError: string index out of range
>>>

```

W kodzie tym nie otrzymujemy komunikatu „po try?”, ponieważ sterowanie normalnie nie jest wznawiane po bloku `try/finally`, kiedy wystąpi wyjątek. Zamiast tego Python przeskakuje

z powrotem do działania z `finally`, a następnie przekazuje wyjątek w góre do poprzedniego programu obsługi (w tym przypadku będzie to domyślny program obsługi wyjątków znajdujący się na najwyższym poziomie). Jeśli zmodyfikujemy to wywołanie wewnątrz funkcji, tak by nie wywoływało ono wyjątku, kod z `finally` nadal jest wykonywany, jednak po instrukcji `try` program jest kontynuowany.

```
>>> def after():
...     try:
...         fetcher(x, 3)
...     finally:
...         print('po pobraniu')
...         print('po try?')
...
>>> after()
po pobraniu
po try?
>>>
```

W praktyce kombinacje `try/except` przydają się do przechwytywania błędów i radzenia sobie z nimi, natomiast `try/finally` są użyteczne w celu zagwarantowania, że działania końcowe zostaną wykonane bez względu na to, czy w kodzie bloku `try` wystąpi jakiś wyjątek. Możemy na przykład użyć `try/except` do przechwycenia błędów zgłaszanych przez kod importowany z biblioteki zewnętrznej, natomiast `try/finally` wykorzystać w celu zapewnienia, że wywołania zamkające pliki lub połączenia z serwerem zawsze zostaną wykonane. Praktyczne przykłady takich sytuacji zobaczymy później w tej części książki.

Choć służą one do zupełnie innych celów, od Pythona 2.5 możemy teraz mieszać części `except` oraz `finally` w jednej instrukcji `try — finally` wykonywane jest przy wyjściu bez względu na to, czy wyjątek został zgłoszony, a także bez względu na to, czy wyjątek został przechwycony przez część `except`.

Jak dowiemy się z kolejnego rozdziału, Python 2.6 oraz 3.0 udostępnia alternatywę dla kombinacji `try/finally` przy użyciu pewnych typów obiektów. Instrukcja `with/as` wykonuje logikę menedżera kontekstu obiektu, gwarantując tym samym wystąpienie działania końcowego:

```
>>> with open('lumberjack.txt', 'w') as file:      # Zawsze zamkna plik przy wyjściu
...     file.write('Modrzew!\n')
```

Choć opcja ta wymaga mniejszej liczby wierszy kodu, ma ona zastosowanie jednie przy przetwarzaniu pewnych typów obiektów, dlatego kombinacja `try/finally` jest bardziej uniwersalną strukturą końcową. Z drugiej strony `with/as` może także wykonywać działania początkowe i obsługuje zdefiniowany przez użytkownika kod menedżera kontekstu.

Podsumowanie rozdziału

To już większość kwestii związanych z wyjątkami — jest to naprawdę proste narzędzie.

Podsumowując, wyjątki są w Pythonie narzędziem sterowania przebiegiem wysokiego poziomu. Mogą być zgłaszane przez Pythona lub nasze własne programy. W obu przypadkach mogą być ignorowane (w celu wywołania domyślnego komunikatu o błędzie) lub przechwytywane za pomocą instrukcji `try` (by można je było przetworzyć za pomocą kodu). Instrukcja `try` ma dwa formaty logiczne, które od Pythona 2.5 można łączyć — jeden obsługuje wyjątki, a drugi

Znaczenie sprawdzania błędów

Jednym ze sposobów przekonania się o przydatności wyjątków jest porównanie kodu napisanego w Pythonie oraz w językach niemających wyjątków. Jeśli na przykład chcemy pisać rozbudowane programy w języku C, musimy sprawdzać zwracane wartości oraz kody statusu po każdej operacji, która potencjalnie mogłaby pójść nie tak, i przekazywać wyniki testów w miarę działania programu.

```
doStuff()
{
    if (doFirstThing() == ERROR)
        return ERROR;
    if (doNextThing() == ERROR)
        return ERROR;
    ...
    return doLastThing();
}

main()
{
    if (doStuff() == ERROR)
        badEnding();
    else
        goodEnding();
}
```

Tak naprawdę prawdziwe programy w języku C często zawierają tyle samo kodu sprawdzającego błędy co kodu wykonującego prawdziwą pracę. Jednak w Pythonie nie musimy być aż tak metodyczni (i neurotyczni!). Zamiast tego możemy opakować dowolnie duże fragmenty programu w programy obsługi wyjątków i po prostu napisać te części, które wykonują prawdziwą pracę, zakładając, że wszystko będzie w porządku.

```
# Kod w Pythonie
# Nie przejmujemy się wyjątkami
# więc nie musimy ich wykrywać

def doStuff():
    doFirstThing()
    doNextThing()
    ...
    doLastThing()

if __name__ == '__main__':
    try:
        doStuff()
    except:
        badEnding()
    else:
        goodEnding()
```

Ponieważ sterowanie przeskakuje do programu obsługi natychmiast, kiedy tylko wystąpi wyjątek, nie ma konieczności angażowania całego kodu w stanie na straży i ostrzeganie przed błędami. Co więcej, ponieważ Python wykrywa błędy automatycznie, nasz kod zazwyczaj w ogóle nie musi ich sprawdzać. W rezultacie wyjątki pozwalają nam w dużej mierzeignorować niezwykłe przypadki oraz unikać kodu sprawdzającego błędy.

wykonuje kod końcowy bez względu na to, czy wyjątek wystąpił. Instrukcje `raise` oraz `assert` Pythona wywołują wyjątki na żądanie (dotyczy to zarówno wyjątków wbudowanych, jak i nowych, zdefiniowanych przez nas za pomocą klas). Instrukcja `with/as` jest alternatywnym sposobem upewnienia się, że działania końcowe zostaną wykonane dla obsługujących je obiektów.

W pozostałej części książki uzupełnimy pewne szczegóły dotyczące przedstawionych tu instrukcji, sprawdzimy, jakie elementy mogą pojawiać się pod `try`, i omówimy obiekty wyjątkowe.

ków oparte na łańcuchach znaków oraz klasach. Kolejny rozdział zaczniemy od bliższego przyjrzenia się przedstawionym tutaj instrukcjom. Zanim jednak przejdziemy dalej, zalecam wykonanie quizu podsumowującego omówione tutaj podstawy.

Sprawdź swoją wiedzę — quiz

1. Należy wymienić trzy cele, jakie pełni przetwarzanie wyjątków.
2. Co stanie się z wyjątkiem, kiedy nie zrobimy nic specjalnego w celu obsłużenia go?
3. W jaki sposób skrypt może sobie poradzić po wystąpieniu wyjątku?
4. Należy podać dwa sposoby wywoływania wyjątków w skrypcie.
5. Należy podać dwa sposoby określania działań, które zostaną wykonane w momencie zakończenia skryptu, bez względu na to, czy wystąpił wyjątek.

Sprawdź swoją wiedzę — odpowiedzi

1. Przetwarzanie wyjątków przydaje się w obsłudze błędów, działaniach końcowych oraz powiadomieniach o zdarzeniach. Może także uproszczyć obsługę przypadków specjalnych i można je wykorzystać do implementacji alternatywnego przebiegu sterowania. Ogólnie przetwarzanie wyjątków pozwala zminimalizować ilość kodu sprawdzającego błędy, jakiego może wymagać program. Ponieważ wszystkie błędy odfiltrowywane są do programów obsługi wyjątków, być może nie będziemy musieli sprawdzać wyniku każdego z wykonywanych działań.
2. Wszystkie nieprzechwycone wyjątki odfiltrowywane są do domyślnego programu obsługi wyjątków, jaki Python udostępnia na najwyższym poziomie skryptu. Ten program obsługuje wyjątków wyświetla znane komunikaty o błędach i zamyka skrypt.
3. Jeśli chcemy uniknąć domyślnego wyświetlania komunikatu o błędzie i zamknięcia, możemy zapisać w kodzie instrukcje `try/except`, przechwytyjące zgłoszone wyjątki i radzące sobie z nimi. Po przechwyceniu wyjątku ten zostaje zakończony, a program może być kontynuowany.
4. Instrukcje `raise` oraz `assert` można wykorzystać do wywołania wyjątku w dokładnie taki sam sposób, jak gdyby był on zgłoszony przez samego Pythona. Z reguły można także zgłosić wyjątek, popełniając błąd programistyczny, jednak nie jest to zazwyczaj cel twórcy programu.
5. Instrukcję `try/finally` można wykorzystać do upewnienia się, że pewne działania zostaną wykonane po wyjściu z bloku kodu, bez względu na to, czy kod ten zgłasza wyjątek, czy też nie. Do upewnienia się, że wykonane zostaną działania końcowe, może służyć także instrukcja `with/as`, jednak tylko przy przetwarzaniu typów obiektów, które to obsługują.

Szczegółowe informacje dotyczące wyjątków

W poprzednim rozdziale przyjrzaliśmy się nieco działaniu instrukcji powiązanych z wyjątkami. W tym zagłębimy się w to zagadnienie znacznie bardziej — w niniejszym rozdziale znaleźć można bardziej formalne wprowadzenie do składni przetwarzania wyjątków w Pythonie. Zapoznamy się tutaj zwłaszcza ze szczegółowymi informacjami dotyczącymi instrukcji `try`, `raise`, `assert` oraz `with`. Jak zobaczymy, choć instrukcje te są stosunkowo proste, oferują duże możliwości w zakresie radzenia sobie z wyjątkami w kodzie napisanym w Pythonie.



Jedna uwaga proceduralna: zagadnienie wyjątków w ostatnich latach uległo pewnym zmianom. Od Pythona 2.5 część `finally` może pojawiać się w tej samej instrukcji `try` co części `except` oraz `else` (wcześniej nie można ich było ze sobą łączyć). W Pythonie 3.0 oraz 2.6 nowa instrukcja menedżera kontekstu `with` stała się oficjalną częścią języka, natomiast wyjątki definiowane przez użytkownika muszą teraz być instancjami klas, dziedziczącymi po wbudowanej klasie nadzędnej wyjątków. W Pythonie 3.0 można także napotkać lekko zmodyfikowaną składnię instrukcji `raise` i części `except`. W tym wydaniu książki skupię się na stanie wyjątków z Pythona 2.6 i 3.0, jednak ponieważ przez jakiś czas będzie można w kodzie spotkać oryginalne techniki stosowane we wcześniejszych wersjach, przy okazji wskażę również, jak bardzo zmieniły się niektóre rzeczy w tej dziedzinie.

Instrukcja `try/except/else`

Skoro zapoznaliśmy się już z podstawami, czas na szczegóły. W poniższym omówieniu najpierw postaram się przedstawić `try/except/else` oraz `try/finally` jako osobne instrukcje, ponieważ w wersjach Pythona starszych od 2.5 pełnią one inne role i nie można ich łączyć. Jak wspomniałem w uwadze wyżej, w Pythonie 2.5 i nowszych wersjach `except` oraz `finally` mogą być łączone w jednej instrukcji `try`. Konsekwencje tej zmiany wyjaśnię po osobnym omówieniu dwóch oryginalnych form.

Instrukcja `try` jest instrukcją złożoną, a jej najpełniejsza forma przedstawiona jest poniżej. Rozpoczyna się ona od wiersza nagłówka `try`, po którym następuje blok (normalnie) wciętych instrukcji, a później jedna lub większa liczba części identyfikujących wyjątki, jakie mają

być przechwycone, oraz opcjonalna część `else` na końcu. Słowa `try`, `except` oraz `else` są ze sobą wiązane za pomocą indentacji na tym samym poziomie (czyli wyrównaniu w pionie). Podsumowując, ogólny format tej instrukcji w Pythonie 3.0 przedstawiony jest poniżej.

```
try:  
    <instrukcje>                      # Wykonanie najpierw tego głównego działania  
    except <nazwa1>:  
        <instrukcje>                  # Wykonane, jeśli nazwa1 zostanie zgłoszona w bloku try  
    except (nazwa2, nazwa3):  
        <instrukcje>                  # Wykonane, kiedy wystąpi jeden z tych wyjątków  
    except <nazwa4> as <dane>:  
        <instrukcje>                  # Wykonane, jeśli nazwa4 zostaje zgłoszona i zgłoszona zostaje instancja  
    except:  
        <instrukcje>                  # Wykonane dla wszystkich (pozostałych) zgłoszonych wyjątków  
    else:  
        <instrukcje>                  # Wykonane, jeśli żaden wyjątek nie został zgłoszony w bloku try
```

W instrukcji tej blok pod nagłówkiem `try` reprezentuje *podstawowe działanie* tej instrukcji — kod, który próbujemy wykonać. Części `except` definiują *programy obsługi* dla wyjątków zgłoszanych w bloku `try`, natomiast część `else` (o ile jest obecna) udostępnia program obsługi, który wykonamy, jeśli nie wystąpią żadne wyjątki. Wpis `<dane>` związany jest z możliwościami instrukcji `raise` i klasami wyjątków, które omówimy w dalszej części rozdziału.

A oto, jak działają instrukcje `try`. Po wejściu do instrukcji `try` Python oznacza kontekst bieżącego programu, tak by mógł do niego wrócić, jeśli wystąpi wyjątek. Instrukcje zagnieździone pod nagłówkiem `try` wykonywane są jako pierwsze. To, co się dzieje później, zależy od tego, czym w czasie wykonywania instrukcji z bloku `try` zgłoszane są wyjątki.

- Jeśli wyjątek *wystąpi*, kiedy wykonywane są instrukcje z bloku `try`, Python przeskakuje z powrotem do `try` i wykonuje instrukcje znajdujące się pod pierwszą częścią `except`, odpowiadającą zgłoszonemu wyjątkowi. Sterowanie podejmowane jest pod całą instrukcją `try` po wykonaniu bloku `except` (o ile blok ten nie powoduje zgłoszenia kolejnego wyjątku).
- Jeśli wyjątek występuje w bloku `try`, a nie pasuje do żadnej części `except`, jest on przesyłany do góry aż do ostatniej pasującej instrukcji `try`, do której wesliśmy wcześniej w programie, lub też — jeśli jest to pierwsza taka instrukcja — na najwyższy poziom procesu (co sprawia, że Python kończy działanie programu i wyświetla domyślny komunikat o błędzie).
- Jeśli żaden wyjątek nie pojawi się w czasie wykonywania instrukcji znajdujących się pod nagłówkiem `try`, Python wykonuje instrukcje znajdujące się pod wierszem `else` (o ile jest on obecny), a sterowanie wznowiane jest pod całą instrukcją `try`.

Innymi słowy, części `except` przechwytyują wszystkie wyjątki, jakie wystąpią w czasie wykonywania bloku `try`, natomiast część `else` wykonywana jest tylko wtedy, gdy w czasie wykonywania bloku `try` nie pojawi się żaden wyjątek.

Części `except` są *wyspecjalizowanymi* programami obsługi wyjątków — przechwytyują wyjątki pojawiające się jedynie wewnątrz instrukcji powiązanego z nimi bloku `try`. Ponieważ jednak instrukcje bloku `try` mogą wywoływać funkcje umieszczone w innych miejscach programu, źródło wyjątku może się znajdować poza samą instrukcją `try`. Więcej informacji na ten temat pojawi się przy okazji omawiania zagnieźdzania instrukcji `try` w rozdziale 35.

Części instrukcji try

Kiedy piszemy kod instrukcji `try`, pod nagłówkiem instrukcji `try` mogą się pojawić różne części. W tabeli 33.1 przedstawiono wszystkie możliwe formy; musimy użyć przynajmniej jednej z nich. Z niektórymi spotkaliśmy się już wcześniej — jak wiemy, `except` przechwytuje wyjątki, `finally` działa przy wychodzeniu, a `else` wykonywane jest, kiedy nie wystąpiły żadne wyjątki.

Tabela 33.1. Formy części instrukcji `try`

Forma części	Interpretacja
<code>except:</code>	Przechwytuje wszystkie (lub wszystkie pozostałe) typy wyjątków
<code>except nazwa:</code>	Przechwytuje jedynie wymieniony wyjątek
<code>except nazwa as wartość:</code>	Przechwytuje wymieniony wyjątek oraz jego instancję
<code>except (nazwa1, nazwa2):</code>	Przechwytuje dowolny z wymienionych wyjątków
<code>except (nazwa1, nazwa2) as wartość:</code>	Przechwytuje dowolny z wymienionych wyjątków i jego instancję
<code>else:</code>	Wykonywane, jeśli wyjątki nie zostały zgłoszone
<code>finally:</code>	Zawsze wykonuje ten blok

Z punktu widzenia składni w jednej instrukcji `try` może istnieć dowolna liczba części `except`, jednak `else` można użyć jedynie wtedy, gdy użyto przynajmniej jednego `except`, a ponadto może być tylko jedno `else` i jedno `finally`. Aż do wersji 2.4 część `finally` musiała się pojawiać samodzielnie (bez `else` lub `except`) i tak naprawdę `try/finally` było tam inną instrukcją. Od Pythona 2.5 `finally` może jednak występować w tej samej instrukcji `try` co `except` oraz `else` (więcej informacji na temat reguł stanowiących o kolejności elementów znajdzie się w dalszej części rozdziału, kiedy zapoznamy się z połączoną instrukcją `try`).

Części z dodatkowym wpisem w postaci `as wartość` omówimy, kiedy spotkamy instrukcję `raise`. Umożliwiają one dostęp do obiektów zgłaszanych jako wyjątki.

Pierwszy oraz czwarty wpis z tabeli 33.1 są jednak nowe.

- `except bez nazwy wyjątku (forma except:)` przechwytuje *wszystkie* wyjątki niewymienione wcześniej w instrukcji `try`,
- `except z listą wyjątków w nawiasach (forma except (e1, e2, e3):)` przechwytuje *dowolny* z wymienionych wyjątków.

Ponieważ Python szuka dopasowania wewnątrz instrukcji `try`, przeglądając części `except` od góry do dołu, wersja w nawiasach ma taki sam skutek jak wymienienie każdego wyjątku w osobnej części `except`, jednak ciało tej instrukcji wystarczy zapisać tylko raz. Poniżej znajduje się przykład działania instrukcji z wieloma częściami `except`, który pokazuje, jak bardzo specyficzne mogą być nasze programy obsługi wyjątków.

```
try:  
    action()  
except NameError:  
    ...  
except IndexError:  
    ...  
except KeyError:  
    ...
```

```
except (AttributeError, TypeError, SyntaxError):
    ...
else:
    ...
```

W powyższym przykładzie, jeśli wyjątek zostanie zgłoszony w czasie wykonywania wywołania funkcji `action`, Python powraca do `try` i szuka pierwszego `except` wymieniającego zgłoszony wyjątek. Części `except` sprawdzane są od góry do dołu i od lewej strony do prawej. Python wykonuje instrukcje znajdujące się pod pierwszym `except`, jakie pasuje. Jeśli żadna część `except` nie pasuje, wyjątek przekazywany jest poza tę instrukcję `try`. Warto zauważyc, że `else` wykonywane jest tylko wtedy, gdy w funkcji `action` nie wystąpi żaden wyjątek. Nie jest wykonywane natomiast, kiedy zgłoszony zostaje wyjątek niepasujący do żadnego `except`.

Jeśli naprawdę potrzebujemy uniwersalnej części `except` przechwytyjącej wszystko, przyda nam się puste `except`.

```
try:
    action()
except NameError:
    ...
except IndexError:
    ...
except:
    ...
else:
    ...
```

Obsługa NameError
Obsługa IndexError
Obsługa wszystkich pozostałych wyjątków
Obsługa przypadku bez wyjątku

Pusta część `except` przechwytuje wszystko i pozwala na to, by nasze programy obsługi wyjątków były tak ogólne lub tak specyficzne, jak tylko sobie życzymy. W niektórych sytuacjach ta forma może być wygodniejsza od podania wszystkich możliwych wyjątków w instrukcji `try`. Przykładowo poniższy kod przechwytuje wszystkie wyjątki bez wymieniania jakichkolwiek.

```
try:
    action()
except:
    ...
```

Przechwytuje wszystkie możliwe wyjątki

Puste `except` wiąże się z pewnymi problemami projektowymi. Choć jest wygodne, może przechwytywać nieoczekiwane wyjątki systemowe niezwiązane z naszym kodem, a także może niechcący przechwytywać wyjątki przeznaczone dla innych programów obsługi. Przykładowo w Pythonie nawet systemowe wywołania wyjścia z programu powodują zgłaszenie wyjątków i zazwyczaj chcemy, by się one powiodły. Struktura ta może jednak również przechwytywać prawdziwe błędy programistyczne, dla których naprawdopodobniej chcemy zobaczyć komunikaty o błędach. Powrócimy do tej pułapki pod koniec tej części książki. Na razie powiem tylko tyle: należy z tej opcji korzystać ostrożnie.

W Pythonie 3.0 wprowadzono rozwiązanie alternatywne rozwiązujące jeden z tych problemów. Przechwycenie wyjątku o nazwie `Exception` daje prawie taki sam efekt jak puste `except`, jednak ignoruje wyjątki powiązane z systemowymi wyjściami z programu.

```
try:
    action()
except Exception:
    ...
```

Przechwytuje wszystkie możliwe wyjątki z wyjątkiem wyjść z programu

Jest to tak samo wygodne jak puste `except`, jednak wiąże się z prawie takimi samymi niebezpieczeństwami. W kolejnym rozdziale, przy okazji omawiania klas wyjątków, przekonamy się, jak działa ta sztuczka.



Uwaga dotycząca wersji: W Pythonie 3.0 wymagane jest użycie części programu obsługi wyjątków `except Wyjątek as Wartość`: (wymienionej w tabeli 33.1 i wykorzystywanej w niniejszej książce) w miejsce starszej formy `except Wyjątek, Wartość..` Ta druga postać nadal jest dostępna w Pythonie 2.6 (jednak nie jest zalecana) — jeśli jednak zostanie użyta, zastosowana zostanie konwersja na pierwszą postać. Zmiana ta została dokonana w celu wyeliminowania błędów związanych z mylением tej starszej formy z zapisem z dwoma alternatywnymi wyjątkami, w Pythonie 2.6 zapisywany jako `except (Wyjątek1, Wyjątek2)..`. Ponieważ w Pythonie 3.0 obsługiwana jest jedynie postać `z as`, przecinki w części `except` zawsze oznaczają krotkę, bez względu na fakt użycia nawiasów lub ich braku, a wartości interpretowane są jako alternatywne wyjątki, które mają być przechwycone. Zmiana ta modyfikuje również reguły dotyczące zakresów — w przypadku zastosowania nowej składni `z as` zmenna `Wartość` na końcu bloku `except` jest usuwana.

Część try/else

Na pierwszy rzut oka cel części `else` nie zawsze jest oczywisty dla osób początkujących. Bez niej jednak nie da się powiedzieć (bez ustawiania i sprawdzania flag Boolean), czy przebieg sterowania przeszedł przez instrukcję `try`, ponieważ wyjątek nie został zgłoszony lub wystąpił i został obsłużony.

```
try:  
    ...wykonanie kodu...  
except IndexError:  
    ...obsłuszenie wyjątku...  
# Czy trafiliśmy tutaj dlatego, że instrukcja try się nie powiodła, czy wręcz przeciwnie?
```

Podobnie jak `else` w pętlach sprawia, że powód wyjścia z pętli jest bardziej oczywisty, część `else` w instrukcjach `try` udostępnia składnię sprawiającą, że to, co się wydarzyło, jest jasne i jednoznaczne.

```
try:  
    ...wykonanie kodu...  
except IndexError:  
    ...obsłuszenie wyjątku...  
else:  
    ...wyjątek nie wystąpił...
```

Możemy prawie emulować część `else`, przenosząc jej kod do bloku `try`.

```
try:  
    ...wykonanie kodu...  
    ...wyjątek nie wystąpił...  
except IndexError:  
    ...obsłuszenie wyjątku...
```

Może to jednak prowadzić do nieprawidłowych klasyfikacji wyjątków. Jeśli akcja „wyjątek nie wystąpił” wywołuje błąd `IndexError`, zostanie zarejestrowana jako niepowodzenie bloku `try` i tym samym błędnie wywoła program obsługi wyjątku znajdujący się pod `try` (subtelne, ale prawdziwe!). Wykorzystując zamiast tego jawnie zdefiniowaną część `else`, sprawiamy, że logika jest bardziej oczywista, i gwarantujemy, że program obsługi `except` zostanie wykonany jedynie dla prawdziwych błędów w kodzie opakowanym za pomocą `try`, a nie dla błędów w działaniu przypadku `else`.

Przykład — zachowanie domyślne

Ponieważ przebieg sterowania w programie jest łatwiejszy do ujęcia w Pythonie niż w języku polskim, wykonajmy kilka przykładów, które będą ilustrować podstawy wyjątków. Wspominałem już, że wyjątki nieprzechwycone przez instrukcje `try` przechodzą w górę do najwyższego poziomu procesu Pythona i wykonują logikę domyślnej obsługi wyjątków tego języka (co oznacza, że Python kończy działający program i wyświetla standardowy komunikat o błędzie). Przyjrzyjmy się teraz przykładowi. Wykonanie poniższego pliku modułu `bad.py` generuje wyjątek dzielenia przez zero.

```
def gobad(x, y):
    return x / y

def gosouth(x):
    print(gobad(x, 0))

gosouth(1)
```

Ponieważ program ignoruje wywoływany wyjątek, Python kończy jego działanie i wyświetla komunikat.

```
% python bad.py
Traceback (most recent call last):
  File "bad.py", line 7, in <module>
    gosouth(1)
  File "bad.py", line 5, in gosouth
    print(gobad(x, 0))
  File "bad.py", line 2, in gobad
    return x / y
ZeroDivisionError: int division or modulo by zero
```

Powyzszy kod wykonałem w oknie powłoki w Pythonie 3.0. Komunikat składa się ze śladu stosu („Traceback”) oraz nazwy (i dodatkowych danych) wyjątku, który został zgłoszony. Ślad stosu wymienia wszystkie wiersze, które były aktywne w czasie, gdy wystąpił wyjątek, od najstarszych do najnowszych. Warto zauważyc, że ponieważ nie pracujemy w sesji interaktywnej, w tym przypadku informacje dotyczące pliku i numeru wiersza mogą być bardziej przydatne. Na powyższym listingu widzimy na przykład, że niepoprawne dzielenie ma miejsce w ostatnim wpisie śladu — drugim wierszu pliku `bad.py`, w instrukcji `return`.¹

Ponieważ Python wykrywa i zgłasza wszystkie błędy w czasie wykonywania za pomocą zgłaszania wyjątków, wyjątki są nieodłącznie związane z zagadnieniami obsługi błędów oraz, szerzej, debugowania. Każdy, kto zapoznał się z przykładami zaprezentowanymi w książce, musiał po drodze widzieć co najmniej kilka wyjątków — nawet błędy literowe generują Syntax Error czy inny wyjątek, kiedy plik jest importowany lub wykonywany (czyli kiedy uruchamiany jest kompilator). Domyślnie otrzymujemy przydatny sposób wyświetlania błędów, taki, jak pokazano powyżej, co pozwala nam prześledzić problem.

Często ten standardowy komunikat o błędzie jest wszystkim, co jest potrzebne do rozwiązywania problemów w kodzie. W przypadku zaawansowanych zadań związanych z debugowaniem

¹ Jak wspomniano w poprzednim rozdziale, tekst komunikatów o błędach oraz śladów stosu z czasem i typem powłoki nieznacznie się zmienia. Nie należy się martwić, jeśli otrzymany komunikat nie jest identyczny z moim. Kiedy na przykład wykonałem to ćwiczenie w IDLE z Pythona 3.0, tekst komunikatu o błędzie pokazywał pełną ścieżkę bezwzględną do katalogu w nazwach plików.

możemy przechwytywać wyjątki za pomocą instrukcji `try` lub wykorzystać jedno z narzędzi do debugowania omówionych w rozdziale 3., do których powrócimy w rozdziale 35. (na przykład moduł `pdb` z biblioteki standardowej).

Przykład — przechwytywanie wbudowanych wyjątków

Domyślna obsługa wyjątków w Pythonie jest często dokładnie tym, czego potrzebujemy — w szczególności w przypadku kodu na najwyższym poziomie pliku skryptu błęd zazwyczaj powinien od razu zakończyć działanie programu. W wielu programach nie ma potrzeby być bardziej specyficzny w zakresie błędów w kodzie.

Czasami jednak chcemy przechwytywać błędy i radzić sobie z nimi. Jeśli nie chcemy, by nasz programkończył swoje działanie, kiedy Python zgłasza wyjątek, wystarczy przechwycić ten wyjątek, opakowując logikę programu w instrukcję `try`. Jest to ważna możliwość w programach takich, jak serwery sieciowe, które muszą działać bez przerwy. Poniższy kod przechwytuje błąd `TypeError`, który Python zgłasza natychmiast, kiedy próbujemy dokonać konkatenacji listy oraz łańcucha znaków (operator `+` oczekuje tego samego typu sekwencji po obu stronach), i radzi sobie z nim.

```
def kaboom(x, y):
    print(x + y)                                # Wywołanie TypeError

try:
    kaboom([0,1,2], "mielonka")
except TypeError:                               # Przechwycenie błędu i poradzenie sobie z nim
    print('Witaj, świecie!')
    print('wznowienie tutaj')                   # Kontynuacja bez względu na wystąpienie wyjątku
```

Kiedy w funkcji `kaboom` wystąpi wyjątek, sterowanie przeskakuje do części `except` instrukcji `try`, która wyświetla komunikat. Ponieważ wyjątek jest „martwy” po przechwyceniu w ten sposób, program kontynuuje wykonywanie pod instrukcją `try`, a nie zostaje zakończony przez Pythona. W rezultacie kod przetwarza i czyści błąd, a skrypt radzi sobie z nim:

```
% python kaboom.py
Witaj, świecie!
wznowienie tutaj
```

Warto zauważyć, że po przechwyceniu wyjątku sterowanie wznowiane jest w miejscu, w którym go przechwyciliśmy (czyli po `try`). Nie ma prostego sposobu na wrócenie do miejsca, w którym wyjątek wystąpił (w powyższym przykładzie — funkcja `kaboom`). W pewnym sensie sprawia to, że wyjątki bardziej przypominają proste przeskoki niż wywołania funkcji — nie istnieje możliwość powrócenia do kodu, który wywołał wyjątek.

Instrukcja `try/finally`

Inną odmianą instrukcji `try` jest jej specjalizacja związana z działaniami końcowymi. Jeśli w instrukcji `try` znajduje się część `finally`, Python zawsze wykoná jej blok instrukcji przy wychodzeniu z instrukcji `try`, bez względu na to, czy w czasie wykonywania bloku `try` wystąpił wyjątek. Jej ogólna forma jest następująca.

```
try:
    <instrukcje>                                # Wykonanie najpierw tego działania
finally:
    <instrukcje>                                # Zawsze wykonuje ten kod na wyjściu
```

W tym wariantie Python rozpoczyna od wykonania bloku instrukcji powiązanego z wierszem nagłówka `try`. To, co dzieje się następnie, uzależnione jest od tego, czy w czasie bloku `try` wystąpi wyjątek.

- Jeśli w czasie wykonywania bloku `try` nie wystąpi wyjątek, Python przeskakuje z powrotem w celu wykonania bloku `finally`, a następnie kontynuuje wykonywanie pod instrukcją `try`.
- Jeśli w czasie wykonywania bloku `try` wystąpi wyjątek, Python nadal powraca i wykonuje blok `finally`, ale następnie przekazuje wyjątek wyżej — albo do znajdującej się wyżej instrukcji `try`, albo do domyślnego programu obsługi na najwyższym poziomie programu. Program nie wznawia wykonywania poniżej instrukcji `try`. Blok `finally` jest zatem wykonywany, nawet jeśli zgłoszony zostanie wyjątek, jednak w przeciwieństwie do `except`, `finally` nie kończy wyjątku — jego zgłoszenie jest kontynuowane po wykonaniu bloku `finally`.

Forma `try/finally` jest przydatna, kiedy chcemy być całkowicie pewni, że po wykonaniu jakiegoś kodu na pewno wystąpi określone działanie, bez względu na zachowanie wyjątku w programie. W praktyce pozwala to na określenie działań oczyszczających, które muszą wystąpić zawsze — takich, jak zamknięcie plików czy połączenia z serwerem.

Warto zauważyć, że część `finally` nie może być wykorzystana w tej samej instrukcji `try` co `except` oraz `else` w Pythonie 2.4 oraz wcześniejszych wersjach, dlatego połączenie `try/finally` najlepiej jest sobie wyobrazić jako osobną formę instrukcji, jeśli korzystamy z tych starszych wydań Pythona. W Pythonie 2.5 i nowszych wersjach `finally` może się pojawiać w tej samej instrukcji co `except` oraz `else`, dlatego dzisiaj istnieje tak naprawdę jedna instrukcja `try` z wieloma opcjonalnymi częściami (więcej informacji na ten temat niedługo). Bez względu na wykorzystywaną wersję część `finally` nadal pełni tę samą rolę — pozwala na określenie działań „czyszczących”, które muszą być wykonywane zawsze, bez względu na wystąpienie wyjątków.



Jak zobaczymy również później w niniejszym rozdziale, w Pythonie 2.6 i 3.0 instrukcja `with` oraz menedżery kontekstu udostępniają oparty na obiektach sposób wykonania podobnego działania dla sytuacji wyjścia. W przeciwieństwie do `finally` ta nowa instrukcja obsługuje również działania wykonywane na wejściu, jednak jest ograniczona w swoim zakresie do obiektów implementujących protokół menedżera kontekstu.

Przykład — działania kończące kod z użyciem `try/finally`

Z kilkoma prostymi przykładami zastosowania instrukcji `try/finally` spotkaliśmy się już w poprzednim rozdziale. Poniżej znajduje się bardziej realistyczny przykład ilustrujący typowe zastosowanie tej instrukcji.

```
class MyError(Exception): pass

def stuff(file):
    raise MyError()

file = open('data', 'w')                                # Otwarcie pliku wyjścia
try:
    stuff(file)                                         # Zgłoszenie wyjątku
finally:
    file.close()                                         # Zawsze zamknięcie pliku w celu wyczyszczenia bufora wyjścia
    print('nie doszliśmy tutaj')                         # Kontynuacja tutaj tylko jeśli nie ma wyjątku
```

W powyższym kodzie opakowaliśmy wywołanie funkcji przetwarzającej plik w instrukcję `try` z częścią `finally` celu zapewnienia, że plik zawsze będzie zamkany, a tym samym finalizowany, bez względu na ewentualne wystąpienie wyjątku. W ten sposób późniejszy kod może mieć pewność, że zawartość bufora wyjścia została zrzucona z pamięci na dysk. Podobna struktura kodu może na przykład zagwarantować, że połączenia z serwerem zawsze będą zamkane.

Jak wiemy z rozdziału 9., obiekty plików są automatycznie zamkane w momencie czyszczenia pamięci — jest to szczególnie przydatne w przypadku plików tymczasowych, których nie przypisujemy do zmiennych. Nie zawsze jednak, zwłaszcza w większych programach, łatwo jest przewidzieć, kiedy nastąpi czyszczenie pamięci. Instrukcja `try` sprawia, że zamknięcie pliku odbywa się w sposób bardziej jawnego oraz przewidywalnego i odnosi się do określonego bloku kodu. Zapewnia zamknięcie pliku w momencie wyjścia z bloku bez względu na fakt wystąpienia wyjątku bądź jego brak.

Funkcja z tego akurat przykładu nie jest szczególnie przydatna (po prostu zgłasza ona wyjątek), jednak opakowanie wywołania w instrukcję `try/finally` jest dobrym sposobem na upewnienie się, że nasze działania kończące (czasu wyjścia) zawsze zostaną wykonane. Python zawsze wykonuje kod z bloków `finally`, bez względu na to, czy w bloku `try` pojawi się wyjątek.²

Kiedy funkcja z kodu powyżej zgłasza wyjątek, sterowanie przeskakuje do tyłu i wykonuje blok `finally` w celu zamknięcia pliku. Wyjątek jest następnie przekazywany albo do innej instrukcji `try`, albo do domyślnego programu obsługi na najwyższym poziomie pliku, wyświetlającego standardowy komunikat błędu i zamkajającego program. Instrukcje po `try` nigdy nie zostaną wykonane. Gdyby funkcja ta *nie* zgłosiła wyjątku, program nadal wykonałby blok `finally` w celu zamknięcia pliku, jednak byłby kontynuowany poniżej całej instrukcji `try`.

Warto również zauważyc, że wyjątek zdefiniowany przez użytkownika ponownie definiowany jest za pomocą klasy — jak zobaczymy w kolejnym rozdziale, wszystkie dzisiejsze wyjątki muszą być instancjami klas, zarówno w Pythonie 2.6, jak i 3.0.

Połączona instrukcja `try/except/finally`

We wszystkich wersjach Pythona przed wydaniem 2.5 (czyli przez mniej więcej pierwsze 15 lat jego istnienia) instrukcja `try` występowała w dwóch odmianach i tak naprawdę były to dwie odrębne instrukcje. Mogliśmy albo wykorzystać `finally` w celu zapewnienia, że zawsze wykonyany zostanie kod czyszczący, albo napisać bloki `except` w celu przechwytczenia określonych wyjątków i poradzenia sobie z nimi, a także opcjonalnego podania części `else`, która zostanie wykonana, jeśli nie wystąpią żadne wyjątki.

Część `finally` nie mogła zatem być łączona z `except` oraz `else`. Po części była to zasługa kwestii implementacyjnych, a po części było tak dla tego, że znaczenie połączonych instrukcji wydawało się niejasne — przechwytywanie wyjątków i radzenie sobie z nimi wydawało się koncepcją rozłączną z wykonywaniem działań czyszczących.

² O ile oczywiście sam Python całkowicie nie przestanie działać. Zazwyczaj dobrze mu jednak wychodzi unikanie tego — dzięki sprawdzaniu możliwych błędów w trakcie wykonywania programu. Kiedy jednak mamy do czynienia z naprawdę poważną awarią, zazwyczaj jest to zasługa błędu w kodzie dołączonego rozszerzenia w języku C, pozostającego poza zakresem Pythona.

W Pythonie 2.5 i nowszych wersjach (w tym wersjach 2.6 i 3.0, wykorzystywanych w niniejszej książce) te dwie instrukcje zostały połączone. Dzisiaj możemy łączyć części finally, except oraz else w jednej instrukcji try. Oznacza to, że możemy już pisać instrukcje o następującej formie.

```
try:                                # Forma połączona
    podstawowe_działanie
    except Wyjątek1:
        program_obsługi_1
    except Wyjątek2:
        program_obsługi_2
    ...
else:
    blok_else
finally:
    blok_finally
```

Kod z bloku *podstawowe_działanie* tej instrukcji wykonywany jest jak zwykle jako pierwszy. Jeśli kod ten zwraca wyjątek, wszystkie bloki except są po kolei sprawdzane, jeden po drugim, w poszukiwaniu dopasowania do zgłoszonego wyjątku. Jeśli zgłoszony wyjątek to *Wyjątek1*, wykonywany jest blok *program_obsługi_1*. Jeśli jest to *Wyjątek2*, wykonywany jest blok *program_obsługi_2* i tak dalej. Jeśli nie zostanie wykonany żaden wyjątek, wykonywany jest *blok_else*.

Bez względu na to, co stało się poprzednio, po zakończeniu głównego bloku działania i obsłużeniu zgłoszonych wyjątków wykonywany jest *blok_finally*. Kod w *blok_finally* zostanie wykonany nawet wtedy, gdy w programie obsługa wyjątku lub bloku części else pojawi się błąd i zgłoszony zostanie nowy wyjątek.

Jak zawsze część finally nie kończy wyjątku — jeśli wyjątek jest aktywny, kiedy wykonywany jest *blok_finally*, kontynuuje on bycie przekazywanym po wykonaniu kodu z finally, a sterowanie przeskakuje do innego miejsca w programie (innej instrukcji try lub domyślnego programu obsługi błędów z najwyższego poziomu). Jeśli w czasie wykonywania finally żaden wyjątek nie jest aktywny, sterowanie wznawiane jest po całej instrukcji try.

W rezultacie część finally wykonywana jest zawsze, bez względu na to, czy:

- wyjątek wystąpił w podstawowym działaniu i został obsłużony,
- wyjątek wystąpił w podstawowym działaniu i nie został obsłużony,
- w podstawowym działaniu nie wystąpiły żadne wyjątki,
- nowy wyjątek został wywołany w jednym z programów obsługi wyjątków.

Warto powtórzyć, że finally służy do określania działań czyszczących, które zawsze muszą się pojawić przy wyjściu z instrukcji try bez względu na to, jakie wyjątki zostały zgłoszone lub obsłużone.

Składnia połączonej instrukcji try

Przy takim połączeniu instrukcja try musi zawierać albo except, albo finally, a części te muszą mieć odpowiednią kolejność:

```
try -> except -> else -> finally
```

gdzie else i finally są opcjonalne i może występować zero lub większa liczba except; natomiast jeśli dostępne jest else, w kodzie musi się znaleźć przynajmniej jedno except. Instrukcja try składa się tak naprawdę z dwóch części — except z opcjonalnym else oraz finally.

Składnię połączonej instrukcji try najlepiej będzie opisać w następujący sposób (nawiasy kwadratowe oznaczają opcjonalność, a znak * oznacza „zero lub większą liczbę”).

```
try:                                # Format 1
    instrukcje
except [typ [as wartość]]:           # W Pythonie 2: [typ [, wartość]]
    instrukcje
[except [typ [as wartość]]:
    instrukcje]*
[else:
    instrukcje]
[finally:
    instrukcje]

try:                                # Format 2
    instrukcje
finally:
    instrukcje
```

Z uwagi na powyższe reguły część else może się pojawić jedynie wtedy, gdy występuje przy najmniej jedno except, i zawsze można mieszać ze sobą except oraz finally, bez względu na to, czy w instrukcji występuje else. Można także mieszać ze sobą finally oraz else, jednak tylko w sytuacji, w której w instrukcji pojawia się także except (choć except może pomijać nazwę wyjątku w celu przechwycenia wszystkiego i wykonania opisanej dalej instrukcji raise w celu ponownego zgłoszenia bieżącego wyjątku). Jeśli złamiemy którąś z reguł dotyczących kolejności, Python zwróci wyjątek błędu składni przed wykonaniem kodu.

Łączenie finally oraz except za pomocą zagnieżdżania

Przed wersją 2.5 możliwe było łączenie części finally oraz except w instrukcji try za pomocą składniowego zagnieżdżania instrukcji try/except w bloku try instrukcji try/finally (technikę tę omówimy dokładniej w rozdziale 35.). Tak naprawdę poniższy kod ma taki sam efekt jak nowa, połączona forma pokazana na początku tego podrozdziału.

```
try:                                # Zagnieżdżony odpowiednik formy połączonej
    try:
        podstawowe_działanie
    except Wyjątek1:
        program_obsługi_1
    except Wyjątek2:
        program_obsługi_2
    ...
    else:
        bez_błędu
finally:
    czyszczenie
```

Blok finally wykonywany jest zawsze przy wychodzeniu, bez względu na to, co stało się w podstawowym działaniu i bez względu na programy obsługi błędów wykonywane w zagnieżdżonej instrukcji try (można prześledzić wymienione wcześniej cztery przypadki w celu sprawdzenia, czy działa to tak samo). Ponieważ else zawsze wymaga istnienia except, postać zagnieżdżona ma te same ograniczenia w zakresie łączenia co forma połączona omówiona wyżej.

Zagnieżdżony odpowiednik jest jednak bardziej niejasny i wymaga więcej kodu niż nowa, połączona forma (co najmniej o jeden czteroznakowy wiersz). Włączenie finally do jednej instrukcji sprawia, że jest ona łatwiejsza do pisania oraz czytania, dlatego jest dziś preferowaną techniką.

Przykład połączonego try

Poniżej znajduje się demonstracja działania połączonej formy `try` w praktyce. Poniższy plik o nazwie `mergedexc.py` zawiera cztery często spotykane sytuacje wraz z instrukcjami `print` opisującymi znaczenie każdej z nich.

```
sep = '-' * 32 + '\n'
print(sep + 'WYJĄTEK ZGŁOSZONY I PRZECHWYCONY')
try:
    x = 'spam'[99]
except IndexError:
    print('wykonano except')
finally:
    print('wykonano finally')
print('po wykonaniu')

print(sep + 'WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY')
try:
    x = 'spam'[3]
except IndexError:
    print('wykonano except')
finally:
    print('wykonano finally')
print('po wykonaniu')

print(sep + 'WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY, WYKONANO ELSE')
try:
    x = 'spam'[3]
except IndexError:
    print('wykonano except')
else:
    print('wykonano else')
finally:
    print('wykonano finally')
print('po wykonaniu')

print(sep + 'WYJĄTEK ZGŁOSZONY, ALE NIEPRZECHWYCONY')
try:
    x = 1 / 0
except IndexError:
    print('wykonano except')
finally:
    print('wykonano finally')
print('po wykonaniu')
```

Po wykonaniu tego kodu w Pythonie 3.0 zwracane są następujące dane wyjściowe (tak naprawdę działanie i dane wyjściowe kodu są w wersji 2.6 takie same, ponieważ wywołania `print` za każdym razem wyświetlają jeden element). Należy prześledzić ten kod w celu zobaczenia, w jaki sposób obsługa wyjątków zwraca dane wyjściowe każdego z czterech testów.

```
c:\misc> C:\Python30\python mergedexc.py
-----
WYJĄTEK ZGŁOSZONY I PRZECHWYCONY
wykonano except
wykonano finally
po wykonaniu
-----
WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY
wykonano finally
po wykonaniu
-----
```

```
WYJĄTEK NIE ZOSTAŁ ZGŁOSZONY, WYKONANO ELSE
wykonano else
wykonano finally
po wykonaniu
-----
WYJĄTEK ZGŁOSZONY, ALE NIEPRZECHWYCONY
wykonano finally
Traceback (most recent call last):
  File "mergedexc.py", line 36, in <module>
    x = 1 / 0
ZeroDivisionError: int division or modulo by zero
```

Przykład ten wykorzystuje wbudowane operacje w podstawowym działaniu — do wywołania wyjątków lub niezrobienia tego. Opiera się na fakcie, iż Python zawsze sprawdza błędy w czasie wykonywania kodu. W kolejnym podrozdziale pokażemy, w jaki sposób można zamiast tego ręcznie wywoływać wyjątki.

Instrukcja raise

By w jawnym sposobie wywołać wyjątki, wykorzystuje się instrukcję `raise`. Ich ogólna forma jest prosta — instrukcja `raise` składa się ze słowa `raise`, po którym opcjonalnie następuje klasa zgłoszanego wyjątku lub jej instancja.

```
raise <instancja>                                # Zgłoszenie instancji klasy
raise <klasa>                                    # Utworzenie i zgłoszenie instancji klasy
raise                                         # Ponowne zgłoszenie ostatniego wyjątku
```

Jak wspomniano wcześniej, wyjątki w Pythonie 2.6 oraz 3.0 są zawsze instancjami klas. Tym samym pierwsza forma instrukcji `raise` jest także najbardziej popularna — przekazujemy bezpośrednio *instancję* utworzoną albo przed instrukcją `raise`, albo wewnątrz samej tej instrukcji. Jeśli zamiast tego przekażemy *klasę*, Python wywoła klasę bez argumentów konstruktora w celu utworzenia instancji do zgłoszenia. Forma ta jest odpowiednikiem dodania nawiasów po referencji do klasy. Ostatnia forma powoduje ponowne zgłoszenie ostatnio zgłoszonego wyjątku. Jest często wykorzystywana w programach obsługi wyjątków w celu przekazania przechwyconych wyjątków.

By to nieco lepiej wyjaśnić, przyjrzyjmy się kilku przykładom. W przypadku wbudowanych wyjątków dwie poniższe formy są równoważne — obie powodują zgłoszenie instancji podanej klasy wyjątku, jednak pierwsza z nich tworzy instancję w sposób niejawnym:

```
raise IndexError                                     # Klasa (utworzenie instancji)
raise IndexError()                                   # Instancja (utworzona w instrukcji)
```

Możemy także utworzyć instancję z wyprzedzeniem. Ponieważ instrukcja `raise` przyjmuje dowolny rodzaj referencji do obiektu, dwa poniższe przykłady powodują zgłoszenie wyjątku `IndexError` tak samo jak dwa poprzednie.

```
exc = IndexError()                                  # Utworzenie instancji z wyprzedzeniem
raise exc

excs = [IndexError, TypeError]
raise excs[0]
```

Kiedy wyjątek zostaje zgłoszony, Python przesyła zgłoszoną instancję wraz z wyjątkiem. Jeśli instrukcja `try` zawiera część `except nazwa as X:`, do zmiennej `X` zostanie przypisana instancja przekazana w instrukcji `raise`.

```
try:  
    ...  
except IndexError as X: # Do X przypisany zostaje zgłoszony obiekt instancji  
    ...
```

Część `as` jest w programie obsługi try opcjonalna (jeśli zostanie pominięta, instancja nie zostanie po prostu przypisana do zmiennej), jednak dołączenie jej pozwala programowi na dostęp zarówno do danych z instancji, jak i metod z klasy wyjątku.

Model ten działa tak samo w przypadku wyjątków zdefiniowanych przez użytkownika i utworzonych za pomocą klas. Poniższy kod przekazuje na przykład argumenty konstruktora klasy wyjątku, które zostają udostępnione w programie obsługi za pośrednictwem przypisanej instancji:

```
class MyExc(Exception): pass  
    ...  
raise MyExc('mielonka') # Klasa wyjątku z argumentami konstruktora  
    ...  
try:  
    ...  
except MyExc as X: # Atrybuty instancji dostępne w programie obsługi  
    print(X.args)
```

Ponieważ zagadnienie to zaczyna się jednak pokrywać z tematem kolejnego rozdziału, odłożymy na razie szczegółowe omówienie tych kwestii.

Bez względu na nazwy wyjątków zawsze są one identyfikowane przez obiekty instancji i w danym momencie aktywny może być najwyżej jeden. Po przechwyceniu przez część `except` w dowolnym miejscu programu wyjątek przestaje istnieć (to znaczy nie będzie przekazywany do innej instrukcji `try`), o ile nie zostanie zgłoszony ponownie przez kolejną instrukcję `raise` lub błąd.

Przekazywanie wyjątków za pomocą raise

Instrukcja `raise` niezawierająca nazwy wyjątku ani dodatkowych danych po prostu ponownie zgłasza bieżący wyjątek. Forma ta jest zazwyczaj wykorzystywana, kiedy musimy przechwycić oraz obsłużyć wyjątek, ale nie chcemy, by wyjątek ten zginął w kodzie.

```
>>> try:  
...     raise IndexError('mielonka') # Wyjątki pamiętają argumenty  
... except IndexError:  
...     print('przekazywanie')  
...     raise # Ponowne zgłoszenie ostatniego wyjątku  
...  
przekazywanie  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
IndexError: mielonka
```

Wykonanie `raise` w taki sposób ponownie zgłasza wyjątek i przekazuje go do wyższego programu obsługi (lub domyślnego programu obsługi znajdującego się na najwyższym poziomie, który zatrzymuje program ze standardowym komunikatem o błędzie). Warto zwrócić uwagę na to, jak argument przekazany do klasy wyjątku pokazuje się w komunikatach o błędach. O tym, dlaczego tak się dzieje, dowiemy się w kolejnym rozdziale.

Łańcuchy wyjątków w Pythonie 3.0 — raise from

Python 3.0 (ale nie wersja 2.6) zezwala na użycie w instrukcji `raise` opcjonalnej części `from`:

```
raise wyjątek from inny_wyjątek
```

Kiedy zastosowana zostanie forma z `from`, drugi wyjątek określa inną klasę lubinstancję wyjątku, dołączane do atrybutu `__cause__` zgłoszanego wyjątku. Jeśli zgłoszony wyjątek nie zostanie przechwycony, Python wyświetla oba wyjątki jako części standardowego komunikatu o błędzie:

```
>>> try:  
...     1 / 0  
... except Exception as E:  
...     raise TypeError('Źle!') from E  
...  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
ZeroDivisionError: int division or modulo by zero
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):  
  File "<stdin>", line 4, in <module>  
TypeError: Źle!
```

Kiedy wyjątek zostaje zgłoszony wewnętrz programu obsługi wyjątków, podobna procedura zostaje użyta w sposób niejawny. Poprzedni wyjątek zostaje dołączony do atrybutu `__context__` nowego wyjątku i jest ponownie wyświetlony w standardowym komunikacie o błędzie, jeśli wyjątek ten nie zostanie przechwycony. Jest to dość zaawansowane i nieco niejasne rozszerzenie działania wyjątków, dlatego po więcej informacji odsyłam do dokumentacji Pythona.



Uwaga na temat wersji: Python 3.0 nie obsługuje już formy `raise Wyjątek, Argumenty`, dostępnej jeszcze w Pythonie 2.6. W 3.0 należy zamiast tego użyć formy wywołania tworzącego instancję `raise Wyjątek(Argumenty)`, opisanej w książce. Równoważny zapis z przecinkiem z wersji 2.6 jest składnią udostępnioną w celu zachowania zgodności z niedziałającym już modelem wyjątków opartych na łaficach znaków, w Pythonie 2.6 uznawaną za przestarzałą. Jeśli zostanie ona użyta, nastąpi konwersja na postać z wywołaniem z wersji 3.0. W poprzednich wersjach Pythona dozwolona była także forma `raise Wyjątek`. W obu nowszych wersjach jest ona przekształcana na `raise Wyjątek()` i wywołuje konstruktor klasy bez argumentów.

Instrukcja assert

Python zawiera również instrukcję `assert` będącą w pewnym sensie przypadkiem specjalnym na cele debugowania. Jest to przede wszystkim po prostu składniowy skrót dla często wykorzystywanego wzorca z instrukcją `raise`, który można sobie wyobrazić jako *warunkową* instrukcję `raise`. Instrukcja w poniższej formie:

```
assert <test>, <dane> # Część <dane> jest opcjonalna
```

działa tak samo jak poniższy kod:

```
if __debug__:  
    if not <test>:  
        raise AssertionError(<dane>)
```

Innymi słowy, jeśli test okaże się fałszem, Python zgłasza wyjątek; element danych (jeśli jest podany) służy jako argument konstruktora wyjątku. Tak jak wszystkie wyjątki, zgłoszony `AssertionError` zakończy działanie programu, jeśli nie przechwyćmy go za pomocą instrukcji `try`; w tym drugim przypadku element danych zostanie wyświetlony jako część komunikatu o błędzie.

Dodatkowo instrukcje `assert` można usunąć z kodu bajtowego skompilowanego programu, jeśli w wierszu polecień Pythona użyjemy opcji `-O`, tym samym optymalizując program. `AssertionError` jest wbudowanym wyjątkiem, a opcja `_debug_` jest wbudowaną nazwą, która automatycznie ustawiana jest na 1 (`True`), o ile nie użyjemy opcji `-O`. W celu użycia trybu optymalizacji i wyłączenia instrukcji `assert` można zastosować kod wiersza polecień taki jak `python -O main.py`.

Przykład — wyłapywanie ograniczeń (ale nie błędów!)

Instrukcję `assert` najczęściej wykorzystuje się do weryfikowania warunków programu w czasie jego tworzenia. Po ich wyświetleniu tekst komunikatu o błędzie automatycznie zawiera informacje o wierszu w kodzie źródłowym, a także wartość podaną w instrukcji `assert`. Rozważmy poniższy plik `asserter.py`.

```
def f(x):
    assert x < 0, 'x musi być ujemne'
    return x ** 2

% python
>>> import asserter
>>> asserter.f(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "asserter.py", line 2, in f
    assert x < 0, 'x musi być ujemne'
AssertionError: x musi być ujemne
```

Należy pamiętać, że instrukcja `assert` przeznaczona jest przede wszystkim do wyłupywania ograniczeń zdefiniowanych przez użytkowników, a nie do przechwytywania prawdziwych błędów programistycznych. Ponieważ Python sam wyłapuje błędy programistyczne, zazwyczaj nie istnieje konieczność tworzenia instrukcji `assert` w celu wyłupywania elementów takich, jak indeksy poza granicami sekwencji, niedopasowanie typów czy dzielenie przez zero.

```
def reciprocal(x):
    assert x != 0                                     # Bezużyteczna instrukcja assert!
    return 1 / x                                       # Python automatycznie sprawdza dzielenie przez zero
```

Takie instrukcje `assert` są na ogół zbędne — ponieważ Python automatycznie zgłasza wyjątki dla błędów, możemy równie dobrze pozwolić mu wykonać tę pracę za nas³. Inny przykład często spotykanego zastosowania instrukcji `assert` można znaleźć w rozdziale 28. w przykładzie z klasą abstrakcyjną. Wykorzystaliśmy tam `assert` do zapewnienia, że nieudane wywołania niezdefiniowanych metod zwrócią komunikat.

³ Przynajmniej w większości przypadków. Jak sugerowaliśmy wcześniej, jeśli funkcja ma wykonać długo trwające lub niemożliwe do odwrócenia działania, zanim dotrze do miejsca, w którym wywołany zostanie wyjątek, nadal możemy chcieć sprawdzać ją pod kątem występowania błędów. Nawet w tym przypadku należy uważać, by nasze testy nie były nadmiernie specyficzne lub ograniczające, gdyż w przeciwnym razie możemy obniżyć użyteczność programu.

Menedżery kontekstu with/as

Wersje Pythona 2.6 oraz 3.0 wprowadziły nową instrukcję związaną z wyjątkami — with wraz z jej opcjonalną częścią as. Instrukcja ta została zaprojektowana do pracy z obiektami menedżerów kontekstu obsługującymi nowy protokół oparty na metodach. Możliwość ta dostępna jest także jako opcja w Pythonie 2.5, gdzie można ją włączyć za pomocą następującej instrukcji import:

```
from __future__ import with_statement
```

Mówiąc w skrócie, instrukcja with/as ma być alternatywą dla zwykłego zastosowania try/→finally. Podobnie jak ta instrukcja, ma ona służyć do określania działań czasu zakończenia lub działań „czyszczących”, które muszą być wykonywane zawsze, gdy wyjątek wystąpi na etapie przetwarzania. W przeciwieństwie do try/finally, instrukcja with obsługuje bogatszy protokół oparty na obiektach, pozwalając wyznaczać działania wejścia (początkowe) oraz wyjścia (końcowe) wokół bloku kodu.

Python ulepsza za pomocą menedżerów kontekstu niektóre wbudowane narzędzia, takie jak pliki zamykające się automatycznie czy blokady wątków, które automatycznie się blokują oraz odblokowują. Programiści mogą jednak sami tworzyć kod menedżerów kontekstu za pomocą własnych klas.

Podstawowe zastosowanie

Podstawowy format instrukcji with wygląda następująco.

```
with wyrażenie [as zmienna]:  
    blok_with
```

W powyższym kodzie *wyrażenie* ma zwracać obiekt obsługujący protokół zarządzania kontekstem (więcej informacji o tym protokole za moment). Obiekt ten może również zwracać wartość, która zostanie przypisana do elementu *zmienna*, jeśli obecna jest opcjonalna część as.

Warto zauważyć, że do elementu *zmienna* niekoniecznie przypisujemy *wynik* elementu *wyrażenie*. Wynikiem elementu *wyrażenie* jest obiekt obsługujący protokół kontekstu, a do elementu *zmienna* można przypisać coś innego, co zostanie wykorzystane wewnątrz instrukcji. Obiekt zwracany przez *wyrażenie* może następnie wykonać kod startowy, zanim rozpoczęty zostanie *blok_with*, a także kod końcowy po wykonaniu tego bloku, bez względu na to, czy zgłosił on jakiś wyjątek.

Niektóre obiekty wbudowane Pythona zostały rozszerzone w taki sposób, by obsługiwać protokół zarządzania kontekstem, dzięki czemu mogą być wykorzystywane w połączeniu z nową instrukcją with. Obiekty plików (omówione w rozdziale 9.) mają teraz na przykład menedżer kontekstu, który automatycznie zamyka plik po bloku with, bez względu na fakt wystąpienia jakiegoś wyjątku.

```
with open(r'C:\misc\data') as myfile:  
    for line in myfile:  
        print(line)  
        ...tutaj więcej kodu...
```

W powyższym kodzie wywołanie metody open zwraca prosty obiekt pliku przypisany do zmiennej *myfile*. Zmienną tą możemy wykorzystać w połączeniu z normalnymi narzędziami plików — w tym przypadku iterator pliku wczytuje go wiersz po wierszu w pętli *for*.

Obiekt ten obsługuje jednak również protokół zarządzania kontekstem wykorzystywany przez instrukcję `with`. Po wykonaniu instrukcji `with` mechanizm zarządzania kontekstem gwarantuje, że obiekt pliku, do którego odnosi się zmienna `myfile`, zostanie automatycznie zamknięty, nawet jeśli pętla `for` zgłosiła wyjątek w czasie przetwarzania pliku.

Choć obiekty plików są automatycznie zamknięte w momencie czyszczenia pamięci, nie zawsze takie oczywiste jest określenie, kiedy to nastąpi. Instrukcja `with` w tej roli jest alternatywą pozwalającą na zapewnienie zamknięcia pliku po wykonaniu określonego bloku kodu. Jak widzieliśmy wcześniej, podobny efekt możemy uzyskać za pomocą bardziej uniwersalnej i jawnej instrukcji `try/finally`, jednak wymaga ona czterech wierszy kodu administracyjnego w miejsce jednego, jak poniżej:

```
myfile = open(r'C:\misc\data')
try:
    for line in myfile:
        print(line)
        ... tutaj więcej kodu...
finally:
    myfile.close()
```

Nie będziemy omawiać w książce modułów wielowątkowych Pythona (więcej informacji na ten temat można znaleźć w książkach dotyczących poziomu aplikacji, takich jak *Programming Python*), jednak definiowane przez nie blokady oraz obiekty do warunkowej synchronizacji mogą także być wykorzystywane w połączeniu z instrukcją `with`, ponieważ obsługują również protokół zarządzania kontekstem.

```
lock = threading.Lock()
with lock:
    # kluczowy fragment kodu
    ... dostęp do współdzielonych zasobów...
```

W powyższym kodzie mechanizm zarządzania kontekstem gwarantuje, że blokada jest uzyskiwana automatycznie przed wykonaniem bloku i zdejmowana po jego zakończeniu, bez względu na wynik wyjątku.

Zgodnie z informacjami z rozdziału 5. moduł `decimal` wykorzystuje menedżery kontekstu do uproszczenia zapisywania i przywracania bieżącego kontekstu dziesiętnego określającego precyzję oraz zaokrąglenie na cele obliczeń.

```
with decimal.localcontext() as ctx:
    ctx.prec = 2
    x = decimal.Decimal('1.00') / decimal.Decimal('3.00')
```

Po wykonaniu tej instrukcji stan menedżera kontekstu bieżącego wątku jest automatycznie przywracany do stanu sprzed rozpoczęcia instrukcji. By uzyskać to samo za pomocą `try/finally`, musielibyśmy wcześniej zapisać kontekst i ręcznie go przywrócić.

Protokół zarządzania kontekstem

Choć menedżery kontekstu zostały dołączone do niektórych typów wbudowanych, możemy także pisać swoje własne. W celu zaimplementowania menedżerów kontekstów klasy wykorzystują metody specjalne mieszczące się w kategorii przeciążania operatorów. Interfejs oczekiwany od obiektów wykorzystywanych w połączeniu z instrukcją `with` jest nieco skomplikowany, a większości programistów wystarczy jedynie wiedza, jak wykorzystywać istniejące menedżery kontekstu. Na potrzeby osób budujących narzędzia, które będą musiały pisać menedżery kontekstu specyficzne dla aplikacji, przyjrzyjmy się krótko temu, jak to wygląda.

Poniżej znajduje się opis działania instrukcji with.

1. Wyrażenie jest analizowane, w wyniku czego otrzymujemy obiekt znany jako *menedżer kontekstu* (ang. *context manager*), który musi zawierać metody `__enter__` oraz `__exit__`.
2. Wywołana zostaje metoda `__enter__` menedżera kontekstu. Wartość przez nią zwracana jest przypisywana do zmiennej w części `as`, jeśli jest ona obecna. W przeciwnym razie jest ona usuwana.
3. Wykonywany jest kod w zagnieźdzonym bloku with.
4. Jeśli blok ten zwraca wyjątek, wywołana zostaje metoda `__exit__(typ, wartość, ślad)` zawierająca szczegóły wyjątku. Warto zauważyć, że są to te same wartości, które są zwracane przez `sys.exc_info`, opisane w dokumentacji Pythona oraz nieco później w tej części książki. Jeśli metoda ta zwraca wartość będącą fałszem, wyjątek jest zgłoszany ponownie. W przeciwnym razie wyjątek jest kończony. Wyjątek normalnie powinien być zgłoszony ponownie w celu przekazania go poza instrukcję with.
5. Jeśli blok nie zgłasza wyjątku, metoda `__exit__` nadal jest wywoływana, jednak do jej argumentów `typ`, `wartość` oraz `ślad` przekazywane są obiekty `None`.

Przyjrzyjmy się krócej demonstracji działania tego protokołu. Poniższy kod (plik *withas.py*) definiuje obiekt menedżera kontekstu, który śledzi początek i koniec bloku with w każdej instrukcji with, z jaką zostanie wykorzystany.

```
class TraceBlock:  
    def message(self, arg):  
        print('wykonywanie', arg)  
    def __enter__(self):  
        print('rozpoczęcie bloku with')  
        return self  
    def __exit__(self, exc_type, exc_value, exc_tb):  
        if exc_type is None:  
            print('normalne wyjście\n')  
        else:  
            print('zgłoszenie wyjątku!', exc_type)  
        return False # Przekazanie  
  
with TraceBlock() as action:  
    action.message('test 1')  
    print('osiągnięty')  
  
with TraceBlock() as action:  
    action.message('test 2')  
    raise TypeError  
    print('nie został osiągnięty')
```

Warto zauważyć, że metoda `__exit__` tej klasy zwraca `False` w celu przekazania wyjątku. Usunięcie instrukcji `return` dałoby tutaj ten sam efekt, ponieważ domyślona wartość `None` zwracana przez funkcję z definicji jest `False`. Warto również zwrócić uwagę na to, iż metoda `__exit__` zwraca `self` jako obiekt do przypisania do zmiennej `as`. W innych przypadkach użycia zamiast tego może zwrócić zupełnie inny obiekt.

Po wykonaniu tego kodu menedżer kontekstu śledzi początek i koniec bloku instrukcji `with` za pomocą metod `__enter__` oraz `__exit__`. Poniżej znajduje się przykład wykonania tego skryptu w Pythonie 3.0 (w wersji 2.6 także zostanie on wykonany, jednak wyświetli kilka dodatkowych nawiasów krotek).

```
% python withas.py
rozpoczęcie bloku with
wykonywanie test 1
osiągnięty
normalne wyjście

rozpoczęcie bloku with
wykonywanie test 2
zgłoszenie wyjątku! <class 'TypeError'>
Traceback (most recent call last):
  File "withas.py", line 20, in <module>
    raise TypeError
TypeError
```

Menedżery kontekstu są stosunkowo zaawansowanymi narzędziami, przeznaczonymi dla osób tworzących narzędzia, dlatego pominiemy tutaj szczegóły. Pełne informacje można znaleźć w dokumentacji standardowej Pythona; można się z niej na przykład dowiedzieć, że nowy moduł standardowy `contextlib` udostępnia dodatkowe narzędzia służące do tworzenia kodu menedżerów kontekstu. Dla prostszych zastosowań instrukcja `try/finally` udostępnia wystarczającą obsługę działań czasu zakończenia.



W nowej wersji Pythona 3.1 instrukcja `with` może również za pomocą nowej składni z przecinkami podawać kilka menedżerów kontekstu (czasami określanych mianem „zagnieżdżonych”). W poniższym przykładzie działania wyjścia obu plików wykonywane są automatycznie w momencie wyjścia z bloku instrukcji — bez względu na wyniki wyjątków.

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        if 'some key' in line:
            fout.write(line)
```

Można tutaj podać dowolną liczbę menedżerów kontekstu. Większa liczba elementów działa tak samo jak zagnieżdżone instrukcje `with`. W Pythonie 3.1 (oraz późniejszych wersjach) kod:

```
with A() as a, B() as b:
    ...instrukcje...
```

jest równoważny z następującym, działającym w wersjach 3.1, 3.0 oraz 2.6:

```
with A() as a:
    with B() as b:
        ...instrukcje...
```

Więcej informacji na ten temat można znaleźć w dokumentacji Pythona 3.1.

Podsumowanie rozdziału

W niniejszym rozdziale bliżej przyjrzaliśmy się przetwarzaniu wyjątków, zapoznając się z instrukcjami Pythona związanymi z tym zagadnieniem — `try` służy do przechwytywania, `raise` do wywoływania, `assert` do warunkowego wywoływania, a `with` do opakowania bloków kodu w menedżery kontekstu, które określają działania początkowe i końcowe.

Dotychczas wyjątki wydawały się raczej łatwym do zrozumienia i zastosowania narzędziem i tak rzeczywiście jest. Jedyną nieco skomplikowaną kwestią ich dotyczącą jest sposób ich identyfikowania. W kolejnym rozdziale będziemy kontynuować nasze omówienie, opisując,

w jaki sposób implementuje się własne obiekty wyjątków. Jak zobaczymy, klasy już dzisiaj pozwalają na tworzenie w kodzie nowych wyjątków specyficznych dla programu. Zanim jednak przejdziemy dalej, zalecam wykonanie quizu podsumowującego omówione tutaj podstawy.

Sprawdź swoją wiedzę — quiz

1. Do czego służy instrukcja `try`?
2. Jakie są dwie najczęściej wykorzystywane odmiany instrukcji `try`?
3. Do czego służy instrukcja `raise`?
4. Do czego służy instrukcja `assert` i jaką instrukcję przypomina?
5. Do czego służy instrukcja `with/as` i jaką instrukcję przypomina?

Sprawdź swoją wiedzę — odpowiedzi

1. Instrukcja `try` przechwytuje wyjątki i pozwala sobie z nimi radzić. Określa blok kodu, który ma być wykonany, a także jeden lub większą liczbę programów obsługujących wyjątki, które mogą być zgłoszone w czasie wykonywania bloku.
2. Dwie popularne odmiany instrukcji `try` to `try/except/else` (sługa do przechwytywania wyjątków) oraz `try/finally` (sługa do określania działań czyszczących, które muszą wystąpić bez względu na to, czy wyjątek się pojawi). Do Pythona 2.4 były to osobne instrukcje, które mogły być łączone jedynie w zagnieżdżeniu składniowym. Od wersji 2.5 bloki `except` oraz `finally` mogą być łączone w jednej instrukcji, dlatego w rezultacie te dwie formy instrukcji `try` zostały połączone. W połączonej formie `finally` jest wykonywane na wyjściu z instrukcji `try` bez względu na to, jakie wyjątki mogą zostać zgłoszone lub obsłużone.
3. Instrukcja `raise` zgłasza (wywołuje) wyjątek. Python wewnętrznie zgłasza wbudowane wyjątki dla błędów, natomiast nasze skrypty mogą za pomocą `raise` wywoływać wyjątki wbudowane lub zdefiniowane przez użytkownika.
4. Instrukcja `assert` zgłasza wyjątek `AssertionError`, jeśli warunek jest fałszem. Działa jak warunkowa instrukcja `raise` opakowana w instrukcję `if`.
5. Instrukcja `with/as` została zaprojektowana w celu zautomatyzowania działań początkowych oraz końcowych, które muszą wystąpić wokół bloku kodu. Przypomina w przybliżeniu instrukcję `try/finally`, ponieważ jej działania wyjściowe wykonywane są bez względu na ewentualne wystąpienie wyjątku. Pozwala ona jednak na użycie bogatszego protokołu opartego na obiektach, określającego działania wejścia oraz wyjścia.

Obiekty wyjątków

Dotychczas luźno deliberowaliśmy o tym, czym tak naprawdę *jest* wyjątek. Jak wspomniano w poprzednim rozdziale, w Pythonie 2.6 oraz 3.0 zarówno wyjątki wbudowane, jak i zdefiniowane przez użytkownika są identyfikowane przez obiekty *instancji klas*. Oznacza to co prawda, że w celu zdefiniowania nowych wyjątków w programach konieczne jest korzystanie z programowania zorientowanego obiektowo, jednak klasy i ten typ programowania mają kilka niezaprzeczalnych zalet.

Poniżej wymieniono kilka mocnych stron wyjątków opartych na klasach:

- **Można je organizować w kategorie.** Klasy wyjątków lepiej obsługują przyszłe zmiany dzięki udostępnianiu kategorii — dodawanie nowych wyjątków w przeszłości nie będzie wymagało modyfikacji instrukcji `try`.
- **Dołączają informacje o stanie.** Klasy wyjątków udostępniają naturalne miejsce do przechowywania informacji kontekstowych, które mogą być później wykorzystane w programach obsługi `try` — mogą zawierać zarówno dołączane informacje o stanie, jak i wywoływalne metody, dostępne za pośrednictwem instancji.
- **Obsługują dziedziczenie.** Wyjątki oparte na klasach mogą uczestniczyć w hierarchiach dziedziczenia w celu uzyskania wspólnego zachowania i dostosowywania wspólnego zachowania do własnych potrzeb. Przykładowo odziedziczone metody wyświetlania mogą udostępniać wspólny wygląd i zachowanie komunikatów o błędach.

Ze względu na te zalety wyjątki oparte na klasach dobrze obsługują ewolucję programów i większych systemów. Tak naprawdę wszystkie wbudowane wyjątki identyfikowane są przez klasy i są zorganizowane w drzewo dziedziczenia — z powodów wymienionych wyżej. To samo możemy zrobić z własnymi wyjątkami zdefiniowanymi przez użytkownika.

W Pythonie 3.0 wyjątki zdefiniowane przez użytkownika dziedziczą po wbudowanych klasach nadrzędnych wyjątków. Ponieważ te klasy nadrzędne udostępniają przydatne wartości domyślne na cele wyświetlania oraz zachowywania stanu — jak zobaczymy — tworzenie wyjątków zdefiniowanych przez użytkownika wymaga zrozumienia roli wyjątków wbudowanych.



Uwaga dotycząca wersji: Python 2.6 oraz 3.0 wymagają definiowania wyjątków za pomocą klas. Dodatkowo wersja 3.0 wymaga, by klasy wyjątków pochodziły od wbudowanej klasy nadzędnej wyjątków `BaseException` — w sposób pośredni lub bezpośredni. Jak zobaczymy, w większości programów wyjątki dziedziczą po klasie podzędnej tej klasy o nazwie `Exception`, tak by obsługiwane były przechwytyjące wszystko programy obsługi przeznaczone dla normalnych typów wyjątków — wymienienie tej klasy w programie obsługi powoduje przechwycenie wszystkich błędów, jakie powinna przechwytywać większość skryptów. W Pythonie 2.6 klasyczne niezależne klasy mogą także pełnić rolę wyjątków, jednak wymagane jest, by klasy w nowym stylu pochodziły od wbudowanych klas wyjątków, tak samo jak w wersji 3.0.

Wyjątki — powrót do przyszłości

Dawno, dawno temu (no dobrze, w czasach przed Pythonem 2.6 i 3.0) wyjątki można było definiować na dwa różne sposoby. Komplikowało to instrukcje `try`, instrukcje `raise` i w ogóle Pythona. Dzisiaj można to robić tylko w jeden sposób. To dobre rozwiązanie — pozwala na usunięcie z języka różnych pozostałości, które nagromadziły się w nim z uwagi na konieczność zachowania zgodności z wcześniejszymi wersjami. Ponieważ jednak starszy model ułatwia zrozumienie, dlaczego wyjątki są dzisiaj tym, czym są, a także z uwagi na to, że nie da się całkowicie usunąć historii czegoś, co było wykorzystywane przez miliony osób w ciągu ostatniego dwudziestolecia, nasze omówienie teraźniejszości rozpoczęliśmy od krótkiego przyjrzenia się przeszłości.

Wyjątki oparte na łańcuchach znaków znikają

Przed Pythonem 2.6 i 3.0 wyjątki można było definiować zarówno za pomocą instancji klas, jak i obiektów łańcuchów znaków. Wyjątki oparte na łańcuchach znaków w Pythonie 2.5 zaczęły generować ostrzeżenia, a w wersjach 2.6 oraz 3.0 całkowicie zniknęły, dlatego obecnie należy korzystać z zaprezentowanych w książce wyjątków opartych na klasach. Jeśli jednak ktoś pracuje z kodem napisanym w przeszłości, nadal może natrafić na wyjątki oparte na łańcuchach znaków. Mogą się one także pojawić w artykułach i źródłach internetowych napisanych kilka lat temu (co, licząc w pythonowych latach, można uznać za całą wieczność!).

Wyjątki oparte na łańcuchach znaków były łatwe w użyciu — wystarczył dowolny łańcuch znaków. Wyjątki te dopasowywane były po tożsamości obiektu, a nie jego wartości (czyli z użyciem `is`, a nie `==`).

```
C:\misc> C:\Python25\python
>>> myexc = "Mój łańcuch znaków wyjątku"                                # Czy kiedykolwiek byliśmy tacy młodzi?
>>> try:
...     raise myexc
... except myexc:
...     print('przechwycony')
...
przechwycony
```

Ta postać wyjątków została usunięta, ponieważ z punktu widzenia większych programów oraz utrzymania kodu nie dorównywała ona klasom. Choć dzisiaj nie można już używać wyjątków opartych na łańcuchach znaków, są one naturalną podstawą omówienia modelu wyjątków opartych na klasach.

Wyjątki oparte na klasach

Łańcuchy znaków były prostym sposobem definiowania wyjątków. Jak jednak wspomniano wcześniej, klasy mają pewne zalety, które zasługują na zwrócenie na nie uwagi. Co najważniejsze, pozwalają nam identyfikować *kategorie* wyjątków, które są bardziej elastyczne do wykorzystywania oraz utrzymywania od prostych łańcuchów znaków. Co więcej, klasy w naturalny sposób pozwalają na dołączanie szczegółów dotyczących wyjątków oraz obsługują dziedziczenie. Ponieważ są lepszym rozwiązakiem, teraz są one wymagane.

Odkładając na bok szczegóły dotyczące kodu, podstawowa różnica pomiędzy wyjątkami opartymi na łańcuchach znaków oraz tymi opartymi na klasach związana jest ze sposobem dopasowywania zgłoszonych wyjątków do części `except` w instrukcjach `try`.

- Wyjątki oparte na łańcuchach znaków dopasowywane były przez prostą *tożsamość obiektu* — zgłoszony wyjątek dopasowywany był do części `except` za pomocą testu `is` Pythona.
- Wyjątki oparte na klasach dopasowywane są przez *związki z klasami nadzędnymi* — zgłoszony wyjątek dopasowywany jest do części `except`, kiedy ta część `except` wymienia klasę wyjątku lub jego dowolną klasę nadziedzną.

Oznacza to, że kiedy część `except` z instrukcji `try` wymienia klasę nadziedzną, przechwytyuje instancje tej klasy, a także instancje wszystkich jej klas podrzędnych znajdujących się niżej w drzewie klas. W rezultacie wyjątki oparte na klasach obsługują tworzenie *hierarchii* wyjątków — klasy nadziedzne stają się nazwami kategorii, a klasy podrzędne stają się specyficznymi rodzajami wyjątków wewnętrz poszczególnych kategorii. Podając ogólną klasę nadziedzną wyjątków, część `except` może przechwycić całą kategorię wyjątków — dopasowana zostanie dowolna bardziej specyficzna klasa podrzędna.

W przypadku wyjątków opartych na łańcuchach znaków taka koncepcja nie istniała. Ponieważ były one dopasowywane po prostej tożsamości obiektu, nie istniał żaden bezpośredni sposób organizowania wyjątków w bardziej elastyczne kategorie czy grupy. W rezultacie programy obsługi wyjątków były łączone ze zbiorami wyjątków w sposób utrudniający wszelkie modyfikacje.

Oprócz kwestii kategorii wyjątki oparte na klasach lepiej obsługują *informacje o stanie wyjątku* (dołączane do instancji) i pozwalają wyjątkom na uczestniczenie w *hierarchiach dziedziczenia* (w celu uzyskania wspólnego zachowania). Ponieważ oferują one wszystkie zalety klas i ogólnie programowania zorientowanego obiektywnego, stanowią poważną alternatywę dla niedziałającego już modelu wyjątków opartych na łańcuchach znaków kosztem niewielkiej ilości dodatkowego kodu.

Tworzenie klas wyjątków

Przyjrzyjmy się teraz przykładowi, który pokaże nam, jak tak naprawdę w praktyce działają wyjątki. W poniższym pliku `classexc.py` definiujemy klasę nadziedzną o nazwie `General` oraz dwie klasy podrzędne `Specific1` i `Specific2`. Przykład ten ilustruje pojęcie kategorii wyjątków — `General` to nazwa kategorii, a jej dwie klasy podrzędne są specyficznymi typami wyjątków wewnętrz tej kategorii. Programy obsługi przechwytyujące `General` przechwycą również wszystkie klasy podrzędne tej kategorii, w tym `Specific1` oraz `Specific2`.

```

class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0():
    X = General()
    raise X                                # Zgłoszenie instancji klasy nadzędnej

def raiser1():
    X = Specific1()
    raise X                                # Zgłoszenie instancji klasy podrzędnej

def raiser2():
    X = Specific2()
    raise X                                # Zgłoszenie instancji innej klasy podrzędnej

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General:                         # Dopasowanie General lub którejś z jej podklas
        import sys
        print('przechwycono:', sys.exc_info()[0])

```

C:\python30> **python classexc.py**

```

przechwycono: <class '__main__.General'>
przechwycono: <class '__main__.Specific1'>
przechwycono: <class '__main__.Specific2'>

```

Kod ten jest w większości zrozumiały, jednak warto zanotować kilka uwag dotyczących jego implementacji.

Klasa nadzędna Exception

Klasy wykorzystywane do budowania drzew kategorii wyjątków mają niewiele wymagań — tak naprawdę w powyższym przykładzie są w większości puste, a ich ciała zawierają jedynie instrukcje `pass`. Warto jednak zwrócić uwagę na to, że klasa najwyższego poziomu dziedziczy tutaj po wbudowanej klasie `Exception`. Jest to wymagane w Pythonie 3.0; Python 2.6 pozwala, by rolę wyjątków spełniały także klasyczne niezależne klasy, jednak wszystkie klasy w nowym stylu muszą pochodzić od wbudowanych klas wyjątków, tak jak w wersji 3.0. Ponieważ `Exception` udostępnia nam pewne przydatne zachowania, z którymi spotkamy się później (choć tutaj z tego nie korzystamy), w obu nowszych wersjach Pythona dobrze jest, by wyjątki dziedziczyły po tej klasie.

Zgłaszanie instancji

W powyższym kodzie wywołujemy klasy w celu utworzenia *instancji* na potrzeby instrukcji `raise`. W modelu wyjątków opartych na klasach zawsze zgłaszamy i przechwytyujemy obiekt instancji klasy. Jeśli w instrukcji `raise` wymienimy nazwę klasy bez nawiasów, Python wywoła klasę bez argumentu konstruktora w celu utworzenia instancji za nas. Instancje wyjątków można tworzyć przed wykonaniem instrukcji `raise`, tak jak w kodzie powyżej, lub wewnątrz tej instrukcji.

Przechwytywanie kategorii wyjątków

Powyższy kod zawiera funkcje zgłaszające instancje wszystkich trzech klas jako wyjątki, a także instrukcję `try` najwyższego poziomu, która wywołuje funkcje i przechwytuje wyjątki `General`. Ta sama instrukcja `try` przechwytuje również oba wyjątki szczególne, ponieważ są one klasami podrzędnymi `General`.

Szczegóły wyjątku

Program obsługi wyjątku w kodzie powyżej wykorzystuje wywołanie `sys.exc_info` — jak zobaczymy w następnym rozdziale, w ten sposób możemy w uniwersalny sposób uzyskać dostęp do ostatnio zgłoszonego wyjątku. Mówiąc w skrócie, pierwszy element w wyniku jest klasą zgłoszonego wyjątku, natomiast drugi to zgłoszona instancja. Ogólnie w części `except` takiej jak powyższa, przechwytyująca wszystkie klasy w kategorii, `sys.exc_info` jest jednym ze sposobów ustalenia, co dokładnie się stało. W tym akurat przypadku odpowiada to pobraniu atrybutu `__class__` instancji. Jak zobaczymy w kolejnym rozdziale, rozwiązanie z `sys.exc_info` jest często wykorzystywane w połączeniu z pustą częścią `except` przechwytyującą wszystkie błędy.

Ostatnia uwaga wymaga wyjaśnienia. Kiedy wyjątek zostaje przechwycony, możemy być pewni, że zgłoszona instancja jest instancją klasy wymienionej w części `except` lub jednej z jej bardziej specyficznych klas podrzędnych. Z tego powodu atrybut `__class__` instancji podaje także typ instancji. Poniższy wariant kodu działa na przykład tak samo jak poprzedni przykład:

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass

def raiser0(): raise General()
def raiser1(): raise Specific1()
def raiser2(): raise Specific2()

for func in (raiser0, raiser1, raiser2):
    try:
        func()
    except General as X: # X to zgłoszona instancja
        print('przechwycono:', X.__class__)
    # To samo co sys.exc_info()[0]
```

Ponieważ atrybut `__class__` można wykorzystać w ten sposób w celu ustalenia określonego typu zgłoszonego wyjątku, `sys.exc_info` bardziej przydaje się w przypadku pustych części `except`, które nie miałyby w inny sposób dostępu do instancji lub jej klas. Co więcej, prawdziwe programy zazwyczaj *nie powinny musieć się martwić* o to, jaki wyjątek został zgłoszony — wywołując metody instancji w sposób ogólny, automatycznie wyzwalamy działanie przeznaczone dla zgłoszonego wyjątku. Więcej informacji na ten temat oraz na temat `sys.exc_info` znajdziesz w kolejnym rozdziale. Osoby, które nie pamiętają już, co w instancji oznacza `__class__`, odyssey do rozdziału 28. oraz szóstej części książki.

Do czego służą hierarchie wyjątków?

Ponieważ w ostatnim przykładzie istnieją tylko trzy możliwe wyjątki, niezbyt dobrze oddaje on użyteczność wyjątków opartych na klasach. Tak naprawdę ten sam efekt możemy uzyskać, tworząc listę nazw wyjątków w nawiasach wewnętrznych części `except`.

```
try:
    func()
except (General, Specific1, Specific2): # Przechwycenie dowolnego z tych wyjątków
    ...
```

Takie rozwiązanie działało także w przypadku niedziałającego już modelu wyjątków opartych na łańcuchach znaków. W przypadku dużych lub wysokich hierarchii wyjątków łatwiej

może być jednak przechwytywać kategorie za pomocą klas, niż wymieniać każdy element kategorii w jednej części `except`. Co jednak ważniejsze, kategorie wyjątków możemy rozszerzać, dodając nowe klasy podrzędne bez zakłócania działania istniejącego kodu.

Załóżmy na przykład, że tworzymy w Pythonie bibliotekę numeryczną, z której będzie korzystała większa liczba osób. Kiedy piszemy bibliotekę, identyfikujemy dwie kwestie, które mogą pójść nie tak z liczbami w naszym kodzie — dzielenie przez zero oraz przepelenienie liczbowe. Dokumentujemy je jako dwa wyjątki, które może zgłosić biblioteka.

Plik `mathlib.py`

```
class Divzero(Exception): pass
class Oflow(Exception): pass

def func():
    ...
    raise Divzero()
```

Kiedy teraz ktoś będzie używał biblioteki, zazwyczaj opakuje wywołania do naszych funkcji lub klas w instrukcje `try` przechwytyjące nasze dwa wyjątki (jeśli nie przechwycą one naszych wyjątków, wyjątki z biblioteki kończą działanie ich kodu).

Plik `client.py`

```
import mathlib

try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow):
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

Takie rozwiązanie działa dobrze i wiele osób zaczyna korzystać z naszej biblioteki. Po sześciu miesiącach decydujemy się ją zmodyfikować (jak to często mają w zwyczaju programiści). Przy okazji identyfikujemy nowy element, który może pójść nie tak — niedomiar — i dodajemy go jako nowy wyjątek.

Plik `mathlib.py`

```
class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

Niestety, kiedy publikujemy ponownie kod, tworzymy problem w utrzymywaniu dla użytkowników. Jeśli w jawnym sposób wymienili oni nasze wyjątki, teraz muszą wrócić do kodu i zmodyfikować go w każdym miejscu, w którym wywołuje on naszą bibliotekę, tak by dodać również nazwę nowego wyjątku.

Plik `client.py`

```
try:
    mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

Może to nie być koniec świata. Jeśli nasza biblioteka jest wykorzystywana tylko wewnętrznie, sami możemy wprowadzić te zmiany. Możemy również opublikować skrypt w Pythonie, stojący się naprawić taki kod automatycznie (zapewne miałby on tylko kilka wierszy i przy najmniej czasami by się nie mylił). Gdyby jednak większa liczba osób musiała modyfikować wszystkie swoje instrukcje `try` za każdym razem, gdy zmienimy zbiór wyjątków, nie byłaby to najlepsza polityka udostępniania aktualnień.

Użytkownicy mogą próbować unikać tej pułapki, tworząc puste części `except` przechwytyujące *wszystkie* możliwe wyjątki.

```
# Plik client.py

try:
    mathlib.func(...)
except: # Przechwytywanie tutaj wszystkiego
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

Takie obejście może jednak przechwycić więcej, niż należy — elementy takie, jak błędy braku pamięci, sekwencje przerwania wpisane z klawiatury (*Ctrl+C*), wyjścia z systemu, a nawet błędy literowe w kodzie własnych instrukcji `try` użytkowników wywołują wyjątki, a wolelibyśmy, by nie były one przechwytywane i błędnie klasyfikowane jako błędy biblioteki.

Tak naprawdę w tej sytuacji użytkownicy chcą przechwytywać i radzić sobie jedynie z określonymi wyjątkami, które definiuje i powinna zgłaszać biblioteka. Jeśli w trakcie wywołania biblioteki pojawi się jakiś inny wyjątek, naprawdopodobniej będzie to błąd w samej bibliotece (i czas skontaktować się z jej sprzedawcą!). Z reguły w programach obsługi wyjątków lepiej jest być szczególnym niż ogólnym (do kwestii tej powrócimy w kolejnym rozdziale w podrzędziale poświęconym pułapkom związanym z wyjątkami).¹

Co zatem należy zrobić? Hierarchie wyjątków opartych na klasach pozwalają całkowicie zażegnać ten problem. Zamiast definiować wyjątki biblioteki jako zbiór autonomicznych klas, należy ułożyć je w drzewo klas ze wspólną klasą nadrzędną obejmującą całą kategorię.

```
# Plik mathlib.py

class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
...
def func():
    ...
    raise DivZero()
```

W ten sposób użytkownicy biblioteki muszą podać jedynie wspólną klasę nadrzędną (czyli kategorię) w celu przechwycenia wszystkich wyjątków biblioteki, zarówno teraz, jak i w przyszłości.

```
# Plik client.py

import mathlib
...
try:
    mathlib.func(...)
except mathlib.NumErr:
    ...zgłoszenie i poradzenie sobie z wyjątkiem...
```

¹ Jak zasugerował jeden mój mądry student, moduł biblioteki mógłby również udostępniać obiekt krotki zawierający wszystkie wyjątki zgłoszane przez bibliotekę. Klient importowałby wtedy krotkę i podawał ją w części `except` w celu przechwycenia wszystkich wyjątków biblioteki (przypomnijmy, że krotka w `except` oznacza *dowolny* z jej wyjątków). Kiedy później dodamy nowy wyjątek, biblioteka musi jedynie rozszerzyć eksportowaną krotkę. Takie rozwiązanie działa, jednak nadal musielibyśmy uaktualniać krotkę zgodnie z wyjątkami zgłoszonymi w module biblioteki. Wyjątki oparte na klasach oferują również więcej zalet niż tylko kategorie — obsługują też dziedziczenie stanu oraz metod, a także model dostosowywania do własnych potrzeb — coś, czego nie robią proste, pojedyncze wyjątki.

Kiedy powrócimy do kodu i zaczniemy go znowu modyfikować, nowe wyjątki możemy dodać jako nowe podklasy wspólnej klasy nadrzędnej.

```
# Plik mathlib.py
```

```
...
class Uflow(NumErr): pass
```

W rezultacie kod użytkownika przechwytyujący wyjątki biblioteki będzie działał *bez zmian*. Tak naprawdę w przyszłości możemy swobodnie dodawać, usuwać i modyfikować wyjątki w dowolny sposób — dopóki klient wymienia klasę nadrzędną, jest izolowany od zmian w naszym zbiorze wyjątków. Innymi słowy, wyjątki oparte na klasach są o wiele lepszym rozwiązaniem pod względem utrzymywania niż łańcuchy znaków.

Hierarchie wyjątków opartych na klasach mogą także obsługiwać zachowywanie stanu oraz dziedziczenie na sposoby, które idealnie sprawdzają się w większych programach. By jednak zrozumieć te role, musimy najpierw sprawdzić, w jaki sposób klasy wyjątków zdefiniowanych przez użytkownika odnoszą się do wbudowanych wyjątków, po których dziedziczą.

Wbudowane klasy wyjątków

Tak naprawdę przykładów z poprzedniego podrozdziału nie wziąłem z powietrza. Wszystkie wbudowane wyjątki zgłoszane przez Pythona są zdefiniowane jako obiekty klas. Co więcej, są one zorganizowane w płytce hierarchię z ogólnymi klasami nadrzędnymi kategorii i specyficznymi typami klas podrzędnych, podobnie jak wyjątki z drzewa klas, które utworzyliśmy przed chwilą.

W Pythonie 3.0 wszystkie znane wyjątki, z jakimi się spotkaliśmy (na przykład `SyntaxError`), są tak naprawdę po prostu zdefiniowanymi klasami dostępnymi jako nazwy wbudowane w module `builtins` (w Pythonie 2.6 znajdują się zamiast tego w module `__builtin__` i są one także atrybutami modułu biblioteki standardowej o nazwie `exceptions`). Dodatkowo Python organizuje wbudowane wyjątki w hierarchię w celu obsługiwanego różnych trybów przechwytywania. Na przykład:

`BaseException`

Klasa nadrzędna wyjątków najwyższego poziomu. Klasa ta nie jest przeznaczona do bezpośredniego dziedziczenia przez klasy zdefiniowane przez użytkownika (zamiast tego należy użyć `Exception`). Udostępnia domyślny sposób wyświetlania oraz zachowywanie stanu dziedziczone przez klasy podrzędne. Jeśli na instancji tej klasy wywołana zostanie wbudowana funkcja `str` (na przykład przez `print`), klasa ta zwraca łańcuch znaków wyświetlania dla argumentów konstruktora przekazanych przy tworzeniu instancji (lub pusty łańcuch znaków, jeśli argumenty były nieobecne). Dodatkowo, o ile klasa podrzędna nie zastąpi konstruktora tej klasy, wszystkie argumenty do niej przekazane w momencie tworzenia instancji przechowywane są w atrybutie `args` w postaci krotki.

`Exception`

Klasa nadrzędna najwyższego poziomu dla wszystkich wyjątków powiązanych z aplikacjami. Jest bezpośrednią klasą podrzędną `BaseException`, a także klasą nadrzędną wszystkich pozostałych wbudowanych wyjątków, z wyjątkiem klas zdarzeń wyjścia systemu (`SystemExit`, `KeyboardInterrupt` oraz `GeneratorExit`). Prawie wszystkie klasy zdefiniowane przez użytkownika powinny dziedziczyć po tej klasie, a nie po `BaseException`. Jeśli

postępujemy zgodnie z tą konwencją, wymienienie `Exception` w programie obsługi instrukcji `try` sprawi, że nasz skrypt będzie przechwytywał wszystko z wyjątkiem zdarzeń wyjątka systemu, które w normalnych warunkach powinny móc zostać wykonane. W rezultacie `Exception` staje się klasą przechwytyującą wszystko w instrukcjach `try` i jest bardziej dokładna od pustej części `except`.

`ArithmetError`

Klasa nadrzędną wszystkich błędów liczbowych (i jednocześnie klasa podzielona `Exception`).

`OverflowError`

Klasa podzielona `ArithmetError` identyfikująca specyficzny błąd liczbowy.

I tak dalej — więcej informacji na temat tej struktury można znaleźć albo w dodatkowych materiałach, takich jak książka *Python. Leksykon kieszonkowy*, albo w dokumentacji biblioteki Pythona. Warto również zauważyc, że drzewo klas wyjątków różni się nieco w wersjach 2.6 oraz 3.0. Drzewo klas można zobaczyć w tekście pomocy modułu `exceptions` jedynie w Pythonie 2.6 (moduł ten został usunięty z wersji 3.0). Więcej informacji na temat funkcji `help` można znaleźć w rozdziałach 4. oraz 15.).

```
>>> import exceptions  
>>> help(exceptions)  
...pominięto wiele tekstu...
```

Kategorie wbudowanych wyjątków

Drzewo wbudowanych klas pozwala nam wybrać, jak bardzo specyficzny lub ogólny będzie nasz program obsługi wyjątków. Wbudowany wyjątek `ArithmetError` jest klasą nadrzędną dla wyjątków bardziej specyficznych, takich jak `OverflowError` czy `ZeroDivisionError`. Wymieniając `ArithmetError` w instrukcji `try`, przechwycimy dowolny rodzaj zgłoszonego błędu liczbowego. Wymieniając tylko `OverflowError`, przechwycimy jedynie ten specyficzny rodzaj błędu i żaden inny.

Podobnie, ponieważ w Pythonie 3.0 `Exception` jest klasą nadrzędną wszystkich wyjątków poziomu aplikacji, możemy jej użyć w roli *klasy przechwytyjącej wszystko* — rezultat będzie przypominał puste `except`, jednak pozwala to na potraktowanie wyjątków wyjątka z systemu w sposób, jaki zazwyczaj jest pożądany.

```
try:  
    action()  
except Exception:  
    ...obsługa wszystkich wyjątków aplikacji...  
else:  
    ...obsługa przypadków bez wyjątków...
```

W Pythonie 2.6 nie do końca to jednak działa, ponieważ samodzielne wyjątki zdefiniowane przez użytkownika w postaci klasycznych klas nie muszą być klasami podzielnymi klasie podstawowej `Exception`. Technika ta jest o wiele bardziej niezawodna w Pythonie 3.0, ponieważ w tej wersji wszystkie klasy muszą pochodzić od wyjątków wbudowanych. Nawet jednak w Pythonie 3.0 z rozwiązaniem tym wiążą się te same, opisane w poprzednim rozdziale, potencjalne pułapki co z pustym `except` — może ono przechwytywać wyjątki przeznaczone dla innych fragmentów kodu, a także ukrywać prawdziwe błędy programistyczne. Ponieważ jest to tak często spotykany problem, powrócimy do niego w opisie pułapek w kolejnym rozdziale.

Bez względu na to, czy będziemy korzystać z kategorii we wbudowanym drzewie klas, jest to dobry przykład. Używając podobnych technik dla własnych klas wyjątków w kodzie, możemy udostępniać elastyczne oraz łatwe w modyfikacji zbiory wyjątków.

Domyślne wyświetlanie oraz stan

Wbudowane wyjątki udostępniają również domyślne sposoby wyświetlania, a także zachowanie stanu. Często jest to wystarczający poziom logiki wymagany przez klasy zdefiniowane przez użytkownika. O ile nie definiujemy ponownie konstruktorów dziedziczonych przez nasze klasy po wyjątkach wbudowanych, wszystkie argumenty konstruktora przekazane do tych klas zapisywane są w atrybutie krótki args instancji i są automatycznie pokazywane, kiedy instancja jest wyświetlana. Jeśli nie przekazano żadnych argumentów konstruktora, używane są pusta krotka i łańcuch znaków wyświetlania.

Wyjaśnia to, dlaczego argumenty przekazane do wbudowanych klas wyjątków pokazywane są w komunikatach o błędach — wszystkie argumenty konstruktora są dołączane do instancji i pokazywane przy jej wyświetleniu:

```
>>> raise IndexError                                     # To samo co IndexError() — brak argumentów
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError

>>> raise IndexError('mielonka')                      # Argument konstruktora dołączony i wyświetlony
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: mielonka

>>> I = IndexError('mielonka')                         # Dostępny w atrybucie obiektu
>>> I.args
('mielonka',)
```

Tak samo jest w przypadku wyjątków zdefiniowanych przez użytkownika, ponieważ dziedziczą one metody konstruktora oraz wyświetlania obecne we wbudowanych klasach nadrzędnych:

```
>>> class E(Exception): pass
...
>>> try:
...     raise E('mielonka')
... except E as X:
...     print(X, X.args)                                  # Wyświetla i zapisuje argumenty konstruktora
...
mielonka ('mielonka',)
>>> try:
...     raise E('mielonka', 'jajka', 'szynka')
... except E as X:
...     print(X, X.args)
...
('mielonka', 'jajka', 'szynka') ('mielonka', 'jajka', 'szynka')
```

Warto zwrócić uwagę na to, że obiekty instancji wyjątków nie są łańcuchami znaków, a jedynie wykorzystują omówiony w rozdziale 29. protokół przeciążania operatorów `__str__` w celu udostępnienia łańcuchów znaków wyświetlania. W celu dokonania konkatenacji ze zwykłymi łańcuchami znaków należy wykonać ręczną konwersję: `str(X) + "łańcuch_znaków"`.

Choć automatyczna obsługa zachowywania stanu oraz sposobu wyświetlania jest przydatna sama w sobie, w przypadku specyficznych potrzeb w tym zakresie zawsze można ponownie zdefiniować odziedziczone metody, takie jak `__str__` i `__init__` w klasach podrzędnych `Exception`. W kolejnym podrozdziale pokażemy, jak to zrobić.

Własne sposoby wyświetlania

Jak widzieliśmy w poprzednim podrozdziale, domyślnie po przechwyceniu i wyświetleniu instancje wyjątków opartych na klasach wyświetlają wszystko, co przekazaliśmy do konstruktora klasy.

```
>>> class MyBad(Exception): pass
...
>>> try:
...     raise MyBad('Przepraszam -- mój błąd!')
... except MyBad as X:
...     print(X)
...
Przepraszam -- mój błąd!
```

Ten odziedziczony domyślny model wyświetlania wykorzystywany jest także wtedy, gdy wyjątek wyświetlany jest jako część komunikatu o błędzie, kiedy nie zostanie on przechwycony.

```
>>> raise MyBad('Przepraszam -- mój błąd!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    __main__.MyBad: Przepraszam -- mój błąd!
```

W wielu sytuacjach będzie to wystarczające. By jednak uzyskać sposób wyświetlania lepiej dostosowany do naszych potrzeb, możemy zdefiniować w klasie jedną z dwóch metod przeciążania reprezentacji łańcuchów znaków (`__repr__` lub `__str__`) w celu zwrócenia pożdanego łańcucha znaków i wyświetlenia wyjątku. łańcuch znaków zwracany przez metodę zostanie pokazany, kiedy wyjątek zostanie albo przechwycony i wyświetlony, albo dotrze do domyślnego programu obsługi wyjątków.

```
>>> class MyBad(Exception):
...     def __str__(self):
...         return 'Zawsze patrz na życie z humorem...'
...
>>> try:
...     raise MyBad()
... except MyBad as X:
...     print(X)
...
Zawsze patrz na życie z humorem...

>>> raise MyBad()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    __main__.MyBad: Zawsze patrz na życie z humorem...
```

Drobna uwaga, którą warto uwzględnić: zazwyczaj w tym celu należy zdefiniować ponownie metodę `__str__`, ponieważ wbudowane klasy nadrzędne wyjątków ją zawierają, a metoda ta jest bardziej pożądana niż `__repr__` w większości kontekstów (w tym w trakcie wyświetlania). Jeśli zdefiniujemy `__repr__`, w trakcie wyświetlania wywołana zostanie zamiast tego metoda `__str__` klasy nadrzędnej! Więcej informacji na temat tych metod specjalnych można znaleźć w rozdziale 29.

To, co zwraca metoda, zawarte zostaje w komunikatach o błędach dla wyjątków nieprzechwyconych, a także wykorzystane zostaje przy wyświetlaniu wyjątków w jawnym sposobie. W powyższym kodzie w celach ilustracyjnych metoda zwraca zapisany na stałe łańcuch znaków, jednak może ona także wykonywać dowolne przetwarzanie tekstu, na przykład z wykorzystaniem informacji o stanie dołączanych do obiektu instancji. Opcjonalnym związkiem z informacjami o stanie poświęcony jest następny podrozdział.

Własne dane oraz zachowania

Poza obsługą elastycznych hierarchii klasy wyjątków udostępniają również miejsce na dodatkowe informacje o stanie w postaci atrybutów instancji. Jak widzieliśmy wcześniej, wbudowane klasy nadzorujące wyjątków udostępniają domyślny konstruktor automatycznie zapisujący argumenty konstruktora w atrybutie krotki instancji o nazwie args. Choć konstruktor domyślny w wielu przypadkach będzie wystarczający, w niektórych sytuacjach możemy udostępnić własny. Dodatkowo klasy mogą definiować metody wykorzystywane w programach obsługi, udostępniające gotową logikę przetwarzania wyjątków.

Udostępnianie szczegółów wyjątku

Kiedy wyjątek jest zgłoszany, może on przechodzić granice plików — wywołującą wyjątek instrukcja raise i przechwytyjącą go instrukcja try mogą się znajdować w zupełnie innych plikach modułów. Zazwyczaj nie jest najlepszym pomysłem przechowywanie dodatkowych szczegółów w zmiennych globalnych, ponieważ instrukcja try może nie wiedzieć, w którym pliku znajdują się te zmienne. Przekazywanie dodatkowych informacji o stanie w samym wyjątku daje instrukcji try bardziej niezawodny dostęp do nich.

W przypadku klas odbywa się to niemalże automatycznie. Jak widzieliśmy, kiedy zgłoszany jest wyjątek, Python przekazuje obiekt instancji klasy wraz z wyjątkiem. Kod z instrukcją try może uzyskać dostęp do zgłoszonej instancji, wymieniając dodatkowe zmienne po słowie kluczowym as w programie obsługi except. Daje nam to naturalny punkt zaczepienia dla dostarczania danych oraz zachowania do programu obsługi.

Przykładowo program analizujący pliki danych może sygnalizować błąd składniowy, zgłaszając instancję wyjątku wypełnioną dodatkowymi szczegółami dotyczącymi błędu.

```
>>> class FormatError(Exception):
...     def __init__(self, line, file):
...         self.line = line
...         self.file = file
...
>>> def parser():
...     raise FormatError(42, file='spam.txt')      # Kiedy znaleziony zostanie błąd
...
>>> try:
...     parser()
... except FormatError as X:
...     print('Błąd w', X.file, X.line)
...
Błąd w spam.txt 42
```

W części `except` zmienna `X` przypisywana jest do referencji do instancji wygenerowanej, kiedy zgłoszony zostanie wyjątek.² Daje nam to dostęp do atrybutów dołączonych do instancji przez własny konstruktor. Choć moglibyśmy polegać na obsłudze zachowanego stanu we wbudowanych klasach nadrzędnych, dla naszej aplikacji ma to mniejsze znaczenie.

```
>>> class FormatError(Exception): pass           # Odziedziczony konstruktor
...
>>> def parser():
...     raise FormatError(42, 'spam.txt')          # Słowa kluczowe nie są dozwolone!
...
>>> try:
...     parser()
... except FormatError as X:
...     print('Błąd w:', X.args[0], X.args[1])      # Nie jest specyficzne dla tej aplikacji
...
Błąd w: 42 spam.txt
```

Udostępnianie metod wyjątków

Poza umożliwieniem obsługi specyficznych dla aplikacji informacji o stanie własne konstruktory lepiej obsługują dodatkowe zachowania obiektów wyjątków. Klasa wyjątku może również definiować *metody* wywoływanie w programie obsługi. Poniższy kod dodaje na przykład metodę wykorzystującą informacje o stanie wyjątku do zapisania błędów w pliku.

```
class FormatError(Exception):
    logfile = 'formaterror.txt'
    def __init__(self, line, file):
        self.line = line
        self.file = file
    def logerror(self):
        log = open(self.logfile, 'a')
        print('Błąd w:', self.file, self.line, file=log)

    def parser():
        raise FormatError(40, 'spam.txt')

try:
    parser()
except FormatError as exc:
    exc.logerror()
```

Po wykonaniu powyższy skrypt zapisuje komunikat o błędzie do pliku w odpowiedzi na wywołanie metody w programie obsługi wyjątku:

```
C:\misc> C:\Python30\python parse.py
C:\misc> type formaterror.txt
Błąd w spam.txt 40
```

W takiej klasie metody (jak `logerror`) mogą również być odziedziczone po klasach nadrzędnych, a atrybuty instancji (jak `line` oraz `file`) udostępniają miejsce do zapisywania informacji o stanie udostępniających dodatkowy kontekst do wykorzystania w późniejszych wywołaniach

² Jak sugerowaliśmy wcześniej, zgłoszony obiekt instancji jest również dostępny ogólnie, jako drugi element krotki wyników wywołania `sys.exc_info()` — narzędzia zwracającego informacje o ostatnio zgłoszonym wyjątku. Interfejs ten musi być wykorzystywany, jeśli nie podamy nazwy wyjątku w części `except`, ale nadal potrzebny jest nam dostęp do wyjątku, który wystąpił, lub do dołączonych do niego informacji o stanie bądź metod. Więcej informacji na temat `sys.exc_info` znajduje się w kolejnym rozdziale.

metod. Co więcej, klasy wyjątków mogą w dowolny sposób rozszerzać odziedziczone zachowania i dostosowywać je do własnych potrzeb. Innymi słowy, ponieważ wyjątki są w Pythonie definiowane za pomocą klas, wszystkie omówione w szóstej części książki zalety programowania zorientowanego obiektowo dostępne są do użycia.

Podsumowanie rozdziału

W niniejszym rozdziale zajęliśmy się tworzeniem wyjątków zdefiniowanych przez użytkownika. Jak się dowiedzieliśmy, wyjątki implementowane są jako obiekty instancji klas w Pythonie 2.6 oraz 3.0 (wcześniej alternatywa w postaci modelu wyjątków opartych na łańcuchach znaków dostępna była w starszych wersjach, jednak obecnie jest ona przestarzała). Klasy wyjątków obsługują koncepcję hierarchii wyjątków (ułatwiającą późniejsze utrzymanie kodu), pozwalając na dodawanie danych oraz zachowania do wyjątków w postaci atrybutów i metod instancji, a także pozwalają na dziedziczenie w instancjach danych oraz zachowania po klasach nadrzędnych.

Widzieliśmy, że w instrukcji `try` przechwycenie klasy nadrzędnej przechwytuje tę klasę, a także wszystkie klasy podrzędne znajdujące się pod nią w drzewie klas. Klasy nadrzędne stają się nazwami kategorii wyjątków, a klasy podrzędne stają się bardziej specyficznymi typami wyjątków wewnątrz tych kategorii. Widzieliśmy również, że wbudowane klasy nadrzędne wyjątków, po których musimy dziedziczyć, udostępniają przydatne opcje domyślne wyświetlanego oraz zachowywania stanu. W miarę potrzeby możemy je jednak nadpisać.

W kolejnym rozdziale zakończymy tę część książki, omawiając pewne często spotykane przypadki użycia wyjątków oraz badając narzędzia wykorzystywane często przez programistów Pythona. Zanim jednak tam przejdziemy, czas na quiz podsumowujący niniejszy rozdział.

Sprawdź swoją wiedzę — quiz

1. Jakie są dwa nowe ograniczenia, którym w Pythonie 3.0 podlegają wyjątki zdefiniowane przez użytkownika?
2. W jaki sposób zgłasiane wyjątki oparte na klasach dopasowywane są do programów obsługi?
3. Należy wymienić dwa sposoby pozwalające dołączyć informacje kontekstowe do obiektów wyjątków.
4. Należy wymienić dwa sposoby pozwalające na podanie tekstu komunikatu o błędzie dla obiektów wyjątków.
5. Dlaczego nie powinniśmy już dzisiaj używać wyjątków opartych na łańcuchach znaków?

Sprawdź swoją wiedzę — odpowiedzi

1. W Pythonie 3.0 wyjątki muszą być definiowane za pomocą klas (oznacza to, że zgłaszany i przechwytywany jest obiekt instancji klasy). Dodatkowo klasy wyjątków muszą pochodzić od wbudowanej klasy `BaseException` (większość programów dziedziczy po jej klasie).

podzielonej `Exception`, by móc obsługiwać programy obsługi wszystkich błędów dla normalnych typów wyjątków).

2. Wyjątki oparte na klasach dopasowywane są po związkach z klasami nadzędnymi. Podanie klasy nadzędnej w programie obsługi przechwyci instancje tej klasy, a także instancje wszystkich jej klas podzielonych znajdujących się niżej w drzewie klas. Z tego powodu możemy sobie wyobrazić klasy nadzędne jako ogólne kategorie wyjątków, a klasy podzielne jako bardziej szczegółowe typy wyjątków wewnętrz tych kategorii.
3. Informacje kontekstowe możemy dodawać do wyjątków opartych na klasach, wypełniając atrybuty instancji w zgłoszonym obiekcie instancji, zazwyczaj we własnym konstruktorze klasy. W przypadku mniej skomplikowanych potrzeb wbudowane klasy nadzędne wyjątków udostępniają konstruktor automatycznie przechowujący argumenty w instancji (w atrybucie `args`). W programach obsługi wyjątków wymienia się zmienną, która ma być przypisana do zgłoszonej instancji, a następnie przechodzi jej nazwę w celu uzyskania dostępu do dodatkowych informacji o stanie i wywołania metod zdefiniowanych w klasie.
4. Tekst komunikatu o błędzie w wyjątkach opartych na klasach można określić za pomocą własnej metody przeciążania operatorów `__str__`. W przypadku mniej skomplikowanych potrzeb wbudowane klasy nadzędne wyjątków automatycznie wyświetlały wszystkie informacje przekazane do konstruktora klasy. Operacje takie, jak `print` czy `str` automatycznie pobierająła znaków wyświetlania obiektu wyjątku, kiedy jest on pokazywany, albo w sposób bezpośredni, albo jako część komunikatu o błędzie.
5. Ponieważ tak powiedział Guido — zostały one usunięte w wersjach 2.6 oraz 3.0 Pythona. A tak naprawdę istnieją ku temu dobre powody. Wyjątki oparte na łańcuchach znaków nie obsługiwały kategorii, informacji o stanie ani dziedziczenia zachowania w sposób, w jaki robią to wyjątki oparte na klasach. W praktyce sprawiało to, że z wyjątków opartych na klasach łatwiej korzystało się na początku, kiedy programy były małe, ale coraz bardziej było to robić w miarę ich rozrastania się.

Projektowanie z wykorzystaniem wyjątków

Niniejszy rozdział kończy tę część książki kolekcją zagadnień związanych z projektowaniem wyjątków, a także przykładami często spotykanych przypadków użycia, po których zamieszczono omówienie pułapek oraz ćwiczenia przeznaczone dla tej części książki. Ponieważ rozdział ten kończy również główną część książki, zawiera krótkie omówienie narzędzi programistycznych, które mogą nam pomóc przejść z poziomu początkującego użytkownika Pythona do poziomu programisty aplikacji w tym języku.

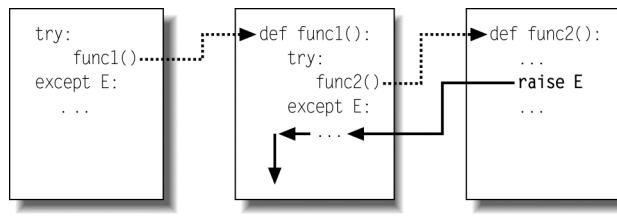
Zagnieżdżanie programów obsługi wyjątków

Nasze dotychczasowe przykłady wykorzystywały pojedynczą instrukcję `try` do przechwytywania wyjątków, jednak co się stanie, jeśli jedna instrukcja `try` zostanie zagnieżdżona wewnętrznej? I co oznacza, jeśli `try` wywołuje funkcję, która wykonuje kolejną instrukcję `try`? Z technicznego punktu widzenia instrukcje `try` mogą być zagnieżdżane w kategoriach składni oraz przebiegu sterowania w programie w czasie wykonywania.

Oba te przypadki możemy zrozumieć, jeśli zdamy sobie sprawę z tego, że Python w czasie wykonywania układu instrukcje `try` na stosie. Kiedy zgłoszany jest wyjątek, Python wraca do ostatniej instrukcji `try` z pasującą częścią `except`. Ponieważ każda instrukcja `try` pozostawia znacznik, Python może przeskoczyć do wcześniejszych `try`, badając ułożone na stosie znaczniki. To właśnie zagnieżdżanie aktywnych programów obsługi mamy na myśli, kiedy mówimy o przekazywaniu (propagacji) wyjątków do „wyższych” programów obsługi — takie programy obsługi to po prostu instrukcje `try`, do których wesliśmy wcześniej w czasie przebiegu wykonywania programu.

Na rysunku 35.1 przedstawiono, co się dzieje, kiedy instrukcje `try` z częściami `except` zagnieżdżane są w czasie wykonywania. Ilość kodu, jaka trafia do bloku `try`, może być dość znaczna i może zawierać wywołania funkcji wywołujące inny kod, który może śledzić te same wyjątki. Kiedy wyjątek zostanie w końcu zgłoszony, Python przeskakuje z powrotem do ostatniej instrukcji `try` wymieniającej ten wyjątek, wykonuje część `except` tej instrukcji, a następnie wznowia wykonywanie po niej.

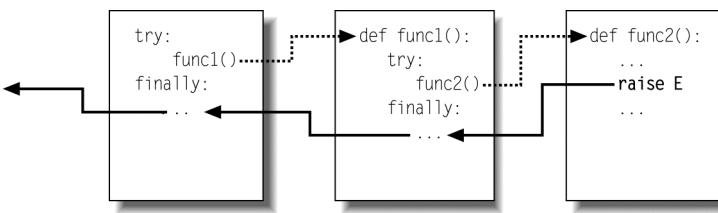
Po przechwyceniu wyjątku jego życie dobiera końca — sterowanie nie przeskakuje z powrotem do *wszystkich* pasujących instrukcji `try` wymieniających ten wyjątek, ponieważ jedynie



Rysunek 35.1. Zagnieżdżone instrukcje `try/except`. Kiedy wyjątek zostanie zgłoszony (przez nas bądź przez Pythona), sterowanie przeskakuje z powrotem do ostatniej odwiedzonej instrukcji `try` z pasującą częścią `except`, a program jest wznowiany po tej instrukcji `try`. Części `except` przechwytyują i zatrzymują wyjątek — to w nich przetwarza się wyjątki oraz radzi sobie z nimi

pierwsza z nich otrzymuje szansę na obsłużenie wyjątku. Na rysunku 35.1 widać na przykład, że instrukcja `raise` z funkcji `func2` przesyła sterowanie z powrotem do programu obsługi z funkcji `func1`, a następnie program jest kontynuowany w `func1`.

W przeciwnieństwie do powyższej sytuacji, kiedy zagnieżdżane są instrukcje `try` zawierające jedynie części `finally`, w momencie wystąpienia wyjątku wykonywany jest każdy blok `finally` po kolei. Python kontynuuje przekazywanie wyjątku do góry do innych instrukcji `try`, a na końcu być może nawet do domyślnego programu obsługi najwyższego poziomu (wyświetlającego standardowe komunikaty o błędach). Jak widać na rysunku 35.2, części `finally` nie kończą wyjątku — po prostu określają one kod, jaki ma być wykonany przy okazji wychodzenia z każdej instrukcji `try` w czasie procesu przekazywania wyjątku. Jeśli wiele części `finally` z instrukcją `try` jest aktywnych w czasie wystąpienia wyjątku, wykonane zostaną *wszystkie*, o ile część `except` nie przechwyci wyjątku gdzieś po drodze.



Rysunek 35.2. Zagnieżdżone instrukcje `try/finally`. Kiedy wyjątek zostanie zgłoszony w tej sytuacji, sterowanie powraca do ostatniej instrukcji `try` w celu wykonania jej części `finally`, a następnie wyjątek przekazywany jest do wszystkich `finally` we wszystkich aktywnych instrukcjach `try`, aż wreszcie dociera do domyślnego programu obsługi na najwyższym poziomie, gdzie wyświetlany jest komunikat o błędzie. Części `finally` przechwytyują wyjątek (jednak go nie zatrzymują) — służą do zamieszczania działań, które mają być wykonywane przy wyjściu

Innymi słowy, to, dokąd podąża program po zgłoszeniu wyjątku, jest całkowicie uzależnione od tego, *gdzie był* — jest to funkcją przebiegu sterowania w skrypcie w czasie wykonywania, a nie tylko kwestią jego składni. Przekazywanie wyjątku w gruncie rzeczy przechodzi z powrotem do instrukcji `try`, do których weszliśmy, ale których jeszcze nie opuściliśmy. Przekazywanie zatrzymuje się, gdy tylko sterowanie odnajduje pasującą część `except`, jednak nie kiedy przechodzi przez części `finally` znajdujące się po drodze.

Przykład — zagnieżdżanie przebiegu sterowania

Przejdzmy teraz do przykładu, by skonkretyzować nieco koncepcje związane z takim zagnieżdżaniem. Poniższy plik modułu `nestexc.py` definiuje dwie funkcje. Funkcja `action2` została zapisana w taki sposób, by wywoływać wyjątek (nie da się dodawać liczb i sekwencji), natomiast funkcja `action1` opakowuje wywołanie `action2` w program obsługi try w celu przechwycenia wyjątku.

```
def action2():
    print(1 + [])
                                # Wygenerowanie wyjątku TypeError

def action1():
    try:
        action2()
    except TypeError:
        print('wewnętrzne try')          # Najbardziej aktualna pasująca instrukcja try

try:
    action1()
except TypeError:
    print('zewnętrzne try')          # Tutaj tylko jeśli action1 ponownie zgłasza wyjątek

% python nestexc.py
wewnętrzne try
```

Warto jednak zauważyć, że kod najwyższego poziomu znajdujący się na dole pliku opakowuje wywołanie `action1` w kolejny program obsługi try. Kiedy funkcja `action2` wywołuje wyjątek `TypeError`, dwie instrukcje try będą aktywne — ta z funkcji `action1` oraz ta znajdująca się na najwyższym poziomie pliku. Python wybiera i wykonuje ostatnią instrukcję try z pasującą częścią `except`, którą w tym przypadku okazuje się instrukcja try znajdująca się wewnątrz funkcji `action1`.

Jak wspomniałem, miejsce, do którego przeskakuje wyjątek, uzależnione jest od przebiegu sterowania w programie w czasie wykonywania. Z tego powodu, by wiedzieć, gdzie trafiemy, musimy wiedzieć, gdzie jesteśmy. W tym przypadku to, gdzie wyjątki są obsługiwane, jest raczej funkcją przebiegu sterowania, a nie składni instrukcji. Możemy jednak również zagnieździć programy obsługi wyjątków za pomocą składni, co zobaczymy poniżej.

Przykład — zagnieżdżanie składniowe

Jak wspomniałem przy okazji omawiania połączonej instrukcji `try/except/finally` w rozdziale 33., można również zagnieździć instrukcje try składniowo, przez ich pozycję w kodzie źródłowym.

```
try:
    try:
        action2()
    except TypeError:
        print('wewnętrzne try')          # Najbardziej aktualna pasująca instrukcja try
    except TypeError:
        print('zewnętrzne try')          # Tutaj tylko jeśli action1 ponownie zgłasza wyjątek
```

Tak naprawdę kod ten ustawia tylko tę samą strukturę zagnieżdżonych programów obsługi wyjątków co poprzedni przykład (zachowuje się również tak samo). Zagnieżdżanie składniowe działa dokładnie tak samo jak przypadki z rysunków 35.1 oraz 35.2. Jedyna różnica polega na tym, że zagnieżdżone programy obsługi są fizycznie osadzone wewnątrz bloku try, a nie

umieszczone w funkcjach znajdujących się gdzie indziej. Przykładowo zagnieźdzone części finally wszystkie wykonywane są w momencie wystąpienia wyjątku, bez względu na to, czy są one zagnieździone składniowo, czy za pomocą przebiegu wykonywania w fizycznie oddzielonych od siebie częściach kodu.

```
>>> try:  
...     try:  
...         raise IndexError  
...     finally:  
...         print('mielonka')  
... finally:  
...     print('MIELONKA')  
...  
mielonka  
MIELONKA  
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
IndexError
```

Graficzna ilustracja działania tego kodu znajduje się na rysunku 35.2. Rezultat jest taki sam, jednak logika funkcji została wstawiona tutaj jako instrukcje zagnieźdzone. Bardziej użyteczny przykład działania zagnieżdżania składniowego można znaleźć w poniższym pliku *except-finally.py*.

```
def raise1(): raise IndexError  
def noraise(): return  
def raise2(): raise SyntaxError  
  
for func in (raise1, noraise, raise2):  
    print('\n', func, sep='')  
    try:  
        try:  
            func()  
        except IndexError:  
            print('przechwycono IndexError')  
    finally:  
        print('wykonano finally')
```

Kod ten przechytuje wyjątek (jeśli zostanie on zgłoszony) i wykonuje działania końcowe części finally bez względu na ewentualne wystąpienie wyjątku. Zrozumienie tego kodu może zająć chwilę, jednak jego rezultat jest w dużej mierze taki sam jak dzisiejsze połączenie *except* oraz *finally* w jednej instrukcji *try* w Pythonie 2.5 i późniejszych wersjach.

```
% python except-finally.py  
  
<function raise1 at 0x026ECA98>  
przechwycono IndexError  
wykonano finally  
  
<function noraise at 0x026ECA50>  
wykonano finally  
  
<function raise2 at 0x026ECBB8>  
wykonano finally  
  
Traceback (most recent call last):  
  File "except-finally.py", line 9, in <module>  
    func()  
  File "except-finally.py", line 3, in raise2  
    def raise2(): raise SyntaxError  
SyntaxError: None
```

Jak widzieliśmy w rozdziale 33., od Pythona 2.5 części except oraz finally można łączyć w tej samej instrukcji try. Niektóre struktury zagnieźdzania składowego widoczne w tym podrozdziale stają się zatem zbędne, choć nadal działają; mogą się pojawić w kodzie utworzonym przed wersją 2.5, z którym nadal można się spotkać, i mogą zostać wykorzystane jako technika implementująca alternatywne zachowania obsługi wyjątków.

Zastosowanie wyjątków

Zapoznaliśmy się już z mechanizmami będącymi podstawą wyjątków. Przyjrzyjmy się teraz niektórym sposobom typowego wykorzystania wyjątków.

Wyjątki nie zawsze są błędami

W Pythonie wszystkie błędy są wyjątkami, ale nie wszystkie wyjątki są błędami. W rozdziale 9. widzieliśmy na przykład, że metody odczytu plików zwracają pusty łańcuch znaków na końcu pliku. Wbudowana funkcja `input` (przedstawiona najpierw w rozdziale 3., a później zastosowana w pętli interaktywnej w rozdziale 10.) wczytuje wiersz tekstu ze standardowego strumienia wejścia (`sys.stdin`) w każdym wywołaniu i zgłasza wbudowany `EOFError` na końcu pliku. Funkcja ta w Pythonie 2.6 nosi nazwę `raw_input`.

W przeciwnieństwie do metod plików funkcja ta nie zwraca pustego łańcucha znaków — pusty łańcuch znaków z `input` oznacza pusty wiersz. Pomimo nazwy wyjątek `EOFError` (czyli ang. *end-of-file error*, błęd konca pliku) jest w tym kontekście tylko sygnałem, a nie błędem. Ze względu na to zachowanie, o ile koniec pliku nie powinien kończyć skryptu, funkcja `input` często pojawia się opakowana w program obsługi wyjątków `try` i zagnieżdzona w pętli, jak w poniższym kodzie.

```
while True:
    try:
        line = input()                      # Wczytanie wiersza ze stdin
    except EOFError:
        break                                # Wyjście z pętli na końcu pliku
    else:
        ...przetwarzanie kolejnego wiersza...
```

Kilka innych wbudowanych wyjątków w podobny sposób jest sygnałami, a nie błędami. Przykładowo wywołanie `sys.exit()` i naciśnięcie na klawiaturze przycisków `Ctrl+C` powodują, odpowiednio, zgłoszenie wyjątków `SystemExit` oraz `KeyboardInterrupt`. Python zawiera również zbiór wbudowanych wyjątków reprezentujących raczej ostrzeżenia niż błędy. Niektóre z nich służą do sygnalizowania przestarzałych opcji języka. Informacje o wbudowanych wyjątkach można znaleźć w dokumentacji biblioteki standardowej, natomiast dokumentacja modułu `warnings` zawiera więcej szczegółów dotyczących ostrzeżeń.

Funkcje mogą sygnalizować warunki za pomocą raise

Wyjątki zdefiniowane przez użytkownika mogą również sygnalizować warunki niebędące błędami. Procedura wyszukiwania może na przykład zostać utworzona tak, by zgłaszać wyjątek, kiedy dopasowanie zostanie odnalezione, zamiast zwracać opcję statusu, którą wywołujący musi zinterpretować. W poniższym kodzie program obsługi wyjątków `try/except/else` wykonuje pracę testu sprawdzającego zwracaną wartość `if/else`.

```

class Found(Exception): pass

def searcher():
    if ...sukces...:
        raise Found()
    else:
        return

try:
    searcher()
except Found:
    ...sukces...          # Wyjątek, jeśli element został odnaleziony
else:
    ...porażka...        # else zwracane, kiedy element nie został odnaleziony

```

Mówiąc bardziej ogólnie, taka struktura kodu może przydać się każdej funkcji, która nie może zwrócić wartości w celu oznaczenia sukcesu bądź porażki. Jeśli na przykład wszystkie obiekty są potencjalnie poprawnymi zwracanymi wartościami, żadna z tych wartości nie będzie w stanie zasygnalizować niezwykłych warunków. Wyjątki umożliwiają sygnalizowanie wyników bez zwracania wartości.

```

class Failure(Exception): pass

def searcher():
    if ...sukces...:
        return ...znaleziony element...
    else:
        raise Failure()

try:
    item = searcher()
except Failure:
    ...zgłoszenie...
else:
    ...tutaj wykorzystanie elementu...

```

Ponieważ Python jest językiem z typami dynamicznymi oraz obsługą polimorfizmu, wyjątki (a nie zwracane wartości) są na ogół preferowanym sposobem sygnalizowania takich warunków.

Zamykanie plików oraz połączeń z serwerem

Przykłady z tej kategorii spotkaliśmy w rozdziale 33. W skrócie, narzędzia do przetwarzania wyjątków wykorzystywane są często do zakończenia korzystania z zasobów systemowych, bez względu na to, czy w trakcie przetwarzania wystąpił błąd.

Przykładowo niektóre serwery wymagają zamykania połączeń w celu zakończenia sesji. W podobny sposób pliki wyjścia mogą wymagać zamknięcia wszystkich wywołań w celu zrzucenia buforów na dysk, natomiast pliki wejścia bez zamknięcia mogą zużywać deskryptory plików. Choć obiekty plików są zamykane automatycznie, kiedy pamięć jest czyszczona — w trakcie gdy są one jeszcze otwarte — czasami trudno jest przewidzieć, kiedy to nastąpi.

Najbardziej ogólnym i bezpośrednim sposobem gwarantującym działania końcowe określonego bloku kodu jest instrukcja `try/finally`.

```

myfile = open(r'C:\misc\script', 'w')
try:
    ...przetworzenie myfile...
finally:
    myfile.close()

```

Jak widzieliśmy w rozdziale 33., niektóre obiekty ułatwiają to w Pythonie 2.6 oraz 3.0, udostępniając *menedżery kontekstu* wykonywane przez instrukcję `with/as` i zamykające dla nas obiekt automatycznie.

```
with open('C:\misc\script', 'w') as myfile:  
    ...przetworzenie myfile...
```

Która opcja będzie zatem lepszym rozwiązaniem? Jak zwykle zależy to od programu. W porównaniu z `try/finally` menedżery kontekstu są *bardziej niejawne*, co jest sprzeczne z ogólną filozofią projektową Pythona. Menedżery kontekstu są także bez wątpienia *mniej ogólne* — dostępne są jedynie dla wybranych obiektów, a pisanie zdefiniowanych przez użytkownika menedżerów kontekstu obsługujących ogólne wymagania w zakresie zamykania jest bardziej skomplikowane od użycia instrukcji `try/finally`.

Z drugiej strony, użycie istniejących menedżerów kontekstu wymaga *mniejszej ilości kodu* niż w przypadku skorzystania z `try/finally`, co widać w powyższych przykładach. Co więcej, protokół menedżerów kontekstu obok działań końcowych (wyjścia) obsługuje także działania *początkowe* (wejścia). Choć `try/finally` jest chyba techniką o szerszym zastosowaniu, menedżery kontekstu mogą być bardziej odpowiednim wyborem, kiedy są już dostępne albo gdy ich dodatkowy poziom skomplikowania ma uzasadnienie.

Debugowanie z wykorzystaniem zewnętrznych instrukcji `try`

Programy obsługi można również wykorzystać do zastąpienia domyślnego zachowania programów obsługi błędów Pythona znajdujących się na najwyższym poziomie. Opanowując cały program (lub jego wywołanie) w zewnętrznej instrukcję `try` w kodzie najwyższego poziomu, możemy przechwytywać wszystkie wyjątki, jakie mogą wystąpić w czasie wykonywania tego programu, tym samym odwracając domyślne zakończenie programu.

W poniższym kodzie pusta część `except` przechwytuje wszystkie nieprzechwycone wyjątki zgłoszone w trakcie wykonywania programu. By uzyskać prawdziwy wyjątek, jaki wystąpił, należy pobrać wynik wywołania funkcji `sys.exc_info` z wbudowanego modułu `sys`. Funkcja ta zwraca krotkę, której dwa pierwsze elementy zawierają klasę bieżącego wyjątku, a także zgłoszony obiekt instancji (więcej informacji na temat funkcji `sys.exc_info` za moment).

```
try:  
    ...wykonanie programu...  
except:                                     # Wszystkie nieprzechwycone wyjątki trafiają tutaj  
    import sys  
    print('nie przechwycono!', sys.exc_info()[0], sys.exc_info()[1])
```

Struktura ta jest często wykorzystywana w czasie programowania w celu zachowania działania programów nawet po wystąpieniu błędów — pozwala ona wykonać dodatkowe testy bez konieczności ponownego uruchamiania. Jest ona również stosowana przy testowaniu kodu innego programu, co opisane jest w kolejnym podrozdziale.

Testowanie kodu wewnętrz tego samego procesu

Niektóre z omówionych wyżej wzorców kodu można połączyć w aplikacji testującej inny kod wewnętrz tego samego procesu.

```
import sys  
log = open('testlog', 'a')  
from testapi import moreTests, runNextTest, testName
```

```

def testdriver():
    while moreTests():
        try:
            runNextTest()
        except:
            print('PORAŻKA', testName(), sys.exc_info()[:2], file=log)
        else:
            print('SUKCES', testName(), file=log)
testdriver()

```

W powyższym kodzie funkcja `testdriver` przechodzi serię wywołań sprawdzających (szczegóły modułu `testapi` pominiemy). Ponieważ nieprzechwycony wyjątek w testowanym kodzie spowodowałby zakończenie całego procesu funkcji przeprowadzającej test, musimy opakować wywołania testowanego kodu w instrukcję `try`, jeśli chcemy móc kontynuować proces testowania po jego niepowodzeniu. Jak zawsze pusta część `except` przechwytyuje wszystkie nieprzechwycone wyjątki wygenerowane przez testowany kod i wykorzystuje funkcję `sys.exc_info` do zapisania wyjątków do pliku. Część `else` wykonywana jest wtedy, gdy nie wystąpią żadne wyjątki, czyli kiedy test się powiedzie.

Taki ustandaryzowany kod jest typowy dla systemów sprawdzających funkcje, moduły oraz klasy za pomocą wykonania ich w tym samym procesie co kod wykonujący testy. W praktyce jednak testowanie może być o wiele bardziej wyszukane od powyższego przykładu. By na przykład testować programy zewnętrzne, moglibyśmy zamiast tego sprawdzać kody statusu lub dane wyjściowe generowane przez narzędzia uruchamiające programy, takie jak `os.system` oraz `os.popen`, omówione w dokumentacji biblioteki standardowej (takie narzędzia na ogół nie zgłaszają wyjątków dla błędów w programach zewnętrznych — tak naprawdę testowany kod może działać równolegle do kodu go testującego).

Na końcu rozdziału spotkamy również kilka późniejszych platform testowych udostępnianych przez Pythona, takich jak `doctest` oraz `PyUnit`, które oferują narzędzia służące do porównywania oczekiwanych danych wyjściowych z prawdziwymi wynikami.

Więcej informacji na temat funkcji `sys.exc_info`

Wynik funkcji `sys.exc_info` wykorzystany w dwóch poprzednich przykładach pozwala programowi obsługi wyjątków na uzyskanie dostępu do ostatnio zgłoszonego wyjątku. Jest to szczególnie przydatne, kiedy wykorzystujemy pustą część `except` w celu ślepego przechwyceania wszystkich wyjątków i ustalenia, co zostało zgłoszone.

```

try:
    ...
except:
    # sys.exc_info()[0:2] to klasa i instancja wyjątku

```

Jeśli żaden wyjątek nie jest obsługiwany, wywołanie to zwraca krotkę zawierającą trzy wartości `None`. W przeciwnym razie zwracane wartości to krotka (`typ, wartość, ślad`), gdzie:

- `typ` to klasa obsługiwanej wyjątku,
- `wartość` to instancja klasy wyjątku, która została zgłoszona,
- `ślad` to obiekt śladu reprezentujący stos wywołań w momencie, w którym początkowo wystąpił wyjątek (w dokumentacji modułu `traceback` można znaleźć informacje na temat narzędzi, które można wykorzystać w połączeniu z tym obiektem w celu ręcznego wygenerowania komunikatów o błędach).

Jak widzieliśmy w poprzednim rozdziale, `sys.exc_info` może się czasami przydać do ustalenia określonego typu wyjątku przy przechwytywaniu klas nadzędnych kategorii wyjątków. Ponieważ jednak, jak widzieliśmy, w tym przypadku możemy także uzyskać typ wyjątku, pobierając atrybut `__class__` instancji uzyskany za pomocą części `as`, `sys.exc_info` jest obecnie wykorzystywany przede wszystkim przez pustą część `except`.

```
try:  
    ...  
except General as instance:  
    # instance.__class__ to klasa wyjątku
```

Skorzystanie z interfejsów obiektu instancji oraz polimorfizmu jest zatem często lepszym rozwiązaniem od sprawdzania typów wyjątków — metody wyjątków można definiować dla poszczególnych klas i wykonywać w sposób ogólny:

```
try:  
    ...  
except General as instance:  
    # instance.method() robi, co należy, dla tej instancji
```

Jak zwykle bycie zbyt specyficzny w Pythonie może ograniczyć elastyczność kodu. Rozwiązanie polimorficzne, takie jak w ostatnim przykładzie, lepiej obsługuje przyszłą ewolucję kodu.



Uwaga na temat wersji: W Pythonie 2.6 starsze narzędzia `sys.exc_type` oraz `sys.exc_value` nadal działają i służą do pobierania typu oraz wartości wyjątku, jednak mogą zarządzać tylko jednym, globalnym wyjątkiem na cały proces. Te dwie nazwy zostały usunięte w Pythonie 3.0. Nowsze i preferowane wywołanie `sys.exc_info()` dostępne w wersji 2.6 oraz 3.0 śledzi natomiast informacje o wyjątkach dla każdego wątku, dlatego jest powiązane z wątkiem. Oczywiście rozróżnienie to ma znaczenie tylko wtedy, gdy w programach napisanych w Pythonie wykorzystuje się wątki (zagadnienie to wykracza poza zakres niniejszej książki, jednak Python 3.0 wymusza tę kwestię). Więcej szczegółów znajduje się w bardziej zaawansowanych tekstach.

Wskazówki i pułapki dotyczące projektowania wyjątków

W tym rozdziale łączę ze sobą wskazówki projektowe oraz pułapki, ponieważ okazuje się, że najczęściej spotykane pułapki w dużej mierze wynikają z kwestii związanych z projektowaniem. Wyjątki są generalnie w Pythonie łatwe w użyciu. Prawdziwą sztuką jest jednak zdecydowanie, jak szczegółowe lub ogólne mają być nasze części `except` oraz jak dużo kodu należy opakować w instrukcje `try`. Spróbujmy najpierw odnieść się do tego drugiego problemu.

Co powinniśmy opakować w `try`

Moglibyśmy opakować każdą instrukcję skryptu w osobną instrukcję `try`, jednak byłoby to po prostu głupie (instrukcje `try` musiałyby następnie być opakowywane w następne instrukcje `try!`). Tak naprawdę jest to zagadnienie dotyczące projektowania, wykraczające poza sam język, które stanie się bardziej oczywiste z czasem i nabieranym doświadczeniem. Poniżej znajduje się jednak kilka ogólnych reguł, które mogą się przydać już teraz.

- Operacje, które często kończą się niepowodzeniem, powinny być opakowywane w instrukcje `try`. Pierwszorzędnymi kandydatami do instrukcji `try` są na przykład operacje wchodzące w interakcje ze stanem systemu (jak otwieranie plików czy wywołania gniazd).
- Istnieją jednak wyjątki od powyższej reguły — w prostym skrypcie możemy *chcieć*, by niepowodzenie takiej operacji kończyło nasz program zamiast przechwytywania i ignorowania błędów. Jest tak szczególnie wtedy, gdy to niepowodzenie jest prawdziwą przeszkołą dla działania programu. Niepowodzenie w Pythonie zazwyczaj kończy się użytecznym komunikatem o błędzie (a nie całkowitym wyłączeniem systemu), dlatego często jest to najlepszy wynik, na jaki mogliśmy liczyć.
- Powinniśmy implementować działania końcowe w instrukcjach `try/finally`, by zagwarantować ich wykonanie, o ile niedostępny jest menedżer kontekstu w postaci opcji `with/as`. Ta forma instrukcji pozwala na wykonywanie kodu w dowolnych scenariuszach bez względu na to, czy wystąpią wyjątki.
- Czasami wygodniej jest opakować wywołanie większej funkcji w pojedynczą instrukcję `try`, niż zaśmiecać samą funkcję licznymi instrukcjami `try`. W ten sposób wszystkie wyjątki z funkcji są przekazywane do instrukcji `try` znajdującej się wokół wywołania i redukujemy ilość kodu wewnętrz funkcji.

Typy programów, jakie będziemy pisać, najprawdopodobniej wpłyną na ilość obsługi wyjątków stosowaną przez nas w kodzie. Serwery muszą działać stale, dlatego będą na ogół wymagały instrukcji `try` przechwytyujących wyjątki i pomagających sobie z nimi radzić. Programy testujące inny kod w jednym procesie, jakie widzieliśmy w tym rozdziale, najprawdopodobniej również będą obsługiwać wyjątki. Prostsze, często jednorazowe skrypty często będą całkowicie ignorowały obsługę wyjątków, ponieważ niepowodzenie na dowolnym etapie skryptu będzie powodowało jego zakończenie.

Jak nie przechwytywać zbyt wiele — unikanie pustych `except` i wyjątków

Przejdźmy do kwestii ogólności wyjątków. Python pozwala nam wybrać, które wyjątki chcemy przechwytywać, jednak czasami musimy uważać, by nie obejmować zbyt wiele. Widzieliśmy już na przykład, że pusta część `except` przechwytuje *każdy* wyjątek, jaki może zostać zgłoszony, kiedy wykonywany jest kod z bloku `try`.

Rozwiążanie to łatwo jest utworzyć w kodzie i czasami jest ono pożądane, jednak może się okazać, że przechwycimy w ten sposób błąd, który jest oczekiwany przez program obsługi `try` gdzieś wyżej w strukturze zagnieżdżenia wyjątków. Przykładowo program obsługi wyjątków taki, jak poniższy przechwytuje i zatrzymuje każdy wyjątek, jaki do niego dociera, bez względu na to, czy nie oczekuje go inny program obsługi.

```
def func():
    try:
        ...
    except:
        ...

    try:
        func()
    except IndexError:
        ...
        # Tu zgłaszany jest wyjątek IndexError
        # Jednak wszystko trafia tu i się zatrzymuje

        # Wyjątek powinien być przetworzony tutaj
```

Co gorsza, taki kod może również przechwycić niezwiążane wyjątki systemowe. W Pythonie wyjątki zgłaszą nawet takie zdarzenia, jak błędy pamięci, prawdziwe błędy programistyczne, zatrzymania iteracji, przerwania z klawiatury czy wyjścia z systemu. Takie wyjątki zazwyczaj nie powinny być przechwytywane.

Przykładowo skrypty normalnie kończą działanie, kiedy sterowanie wyjdzie poza koniec pliku najwyższego poziomu. Python udostępnia jednak również wywołanie `sys.exit(statuscode)` pozwalające na wczesne zakończenie. Takie rozwiązanie działa, zgłaszając wbudowany wyjątek `SystemExit` w celu zakończenia programu. Jeśli zatem jakieś programy `try/finally` działają przy wychodzeniu, specjalne typy programów mogą przechwycić to zdarzenie.¹ Z tego powodu instrukcja `try` z pustą częścią `except` może w niezamierzony sposób zapobiec kluczowemu zakończeniu działania, jak w poniższym pliku `exiter.py`.

```
import sys
def bye():
    sys.exit(40)                                # Kluczowy błąd — zakończenie!
try:
    bye()
except:
    print('mam go')                          # Oj — zignorowaliśmy wyjście
print('kontynuuuję...')

% python exiter.py
mam go
kontynuuuję...
```

Tak naprawdę nie jesteśmy w stanie przewidzieć wszystkich rodzajów wyjątków, jakie mogą wystąpić w czasie operacji. Użycie wbudowanych klas wyjątków z poprzedniego rozdziału może nam pomóc w tym akurat przypadku, ponieważ klasa nadziedzona `Exception` nie jest klasą nadziedzoną `SystemExit`:

```
try:
    bye()
except Exception:                         # Nie przechwyci wyjścia, ale *przechwyci* wiele innych wyjątków
    ...
```

W innych przypadkach takie rozwiązanie nie jest lepsze od pustej części `except`. Ponieważ `Exception` jest klasą nadziedzoną wszystkich wbudowanych wyjątków poza zdarzeniami wyjścia z systemu, nadal może potencjalnie przechwycić wyjątki przeznaczone dla innego miejsca programu.

Co chyba najgorsze, zarówno pusta część `except`, jak i przechwytywanie klasy `Exception` przechwytyują również prawdziwe błędy programistyczne, które powinny przez większość czasu być dozwolone. Tak naprawdę te dwie techniki mogą w rezultacie *wyłączyć* mechanizm zgłoszenia błędów Pythona, co sprawi, że zauważenie błędów w kodzie stanie się naprawdę trudne. Rozważmy na przykład poniższy kod.

```
mydictionary = {...}
...
try:
```

¹ Powiązane wywołanie `os._exit` również kończy program, jednak za pomocą natychmiastowego zakończenia — pomija działania czyszczące i nie może być przechwytywane za pomocą bloków `try/except` lub `try/finally`. Zazwyczaj jest wykorzystywane jedynie w procesach potomnych, zagadnieniu wykraczającym poza zakres niniejszej książki. Więcej szczegółów na ten temat można znaleźć w dokumentacji biblioteki lub bardziej zaawansowanych teksthach.

```

x = myditctionary['mielonka']           # Oj — błąd w pisowni
except:
    x = None                           # Zakłada, że mamy KeyError
...kontynuujemy z x...

```

Powyższy kod zakłada, że jedynym błędem, jaki może nam się przytrafić w trakcie indeksowania słownika, jest błąd brakującego klucza. Jednak ponieważ nazwa myditctionary jest napisana z błędem (powinno to być mydictionary), Python zgłasza wyjątek NameError zamiast referencji do niezdefiniowanej zmiennej. Błąd ten zostanie po cichu przechwycony przez program obsługi wyjątków i zignorowany. Program obsługi niepoprawnie wypełni wartość domyślną w dostępie do słownika, maskując błąd programu. Co więcej, przechwycenie Exception miałoby tutaj taki sam rezultat jak pusta część except. Jeśli takie coś zdarzy się w kodzie znacznie oddalonym od miejsca, w którym wykorzystane są pobrane wartości, usunięcie takiego błędu może być dużym wyzwaniem!

Z reguły należy w programach obsługi wyjątków być tak specyfcznym, jak to możliwe — puste części except i przechwytywanie Exception są poręczne, ale potencjalnie bardzo podatne na błędy. W ostatnim przykładzie lepiej byłoby na przykład napisać except KeyError:, by nasze intencje były jasne i by uniknąć przechwytywania niezwiązań zdarzeń. W prostszych skryptach potencjalna możliwość wystąpienia problemów może nie być na tyle znacząca, by przeważyć nad wygodą stosowania pustych części except przechwytyjących wszystko, jednak najczęściej ogólne programy obsługi oznaczają kłopoty.

Jak nie przechwytywać zbyt mało — korzystanie z kategorii opartych na klasach

Z drugiej strony, nasze programy obsługi wyjątków nie powinny być nadmiernie specyficzne. Kiedy w instrukcji try wymienimy określone wyjątki, przechwycimy jedynie to, co jest na tej liście. Nie zawsze jest to złe rozwiązanie, ale jeśli system ewoluje i może w przyszłości zwracać inne wyjątki, być może trzeba będzie powrócić do kodu i dodać te wyjątki do list znajdujących się w każdym miejscu.

Z tym zjawiskiem spotkaliśmy się w poprzednim rozdziale. Poniższy program obsługi został napisany w taki sposób, by traktować MyExcept1 oraz MyExcept2 jako normalne przypadki, a wszystko inne jako błąd. Tym samym jeśli w przyszłości dodamy również przypadek MyExcept3, zostanie on przetworzony jako błąd, o ile nie uaktualnimy listy wyjątków.

```

try:
    ...
except (MyExcept1, MyExcept2):          # Przestaje działać po dodaniu MyExcept3
    ...
else:                                    # Nie-błędy
    ...

```

Zakładamy, że jest błędem

Na szczęście rozważne użycie omówionych w rozdziale 33. wyjątków opartych na klasach może sprawić, że pułapka ta zupełnie zniknie. Jak widzieliśmy, jeśli przechwycimy ogólną klasę nadziedną, możemy w przyszłości dodawać bardziej specyficzne klasy podrzędne bez konieczności ręcznego rozszerzania listy z części except — klasa nadziedna staje się rozszerzalną kategorią wyjątków.

```

try:
    ...
except SuccessCategoryName:            # Będzie OK, jeśli dodamy podkласę MyExcept3

```

```
...  
else:  
...  
# Nie-błędy  
# Zakładamy, że jest błędem
```

Innymi słowy, niewielka ilość projektowania ma spore konsekwencje. Morał z tej historii jest taki, że należy uważać, by w programach obsługi wyjątków nie być ani zbyt ogólnym, ani zbyt specyficzny, i mądrze wybierać poziom szczegółowości naszych instrukcji try opakowujących kod. W szczególności w przypadku większych systemów polityka dotycząca wyjątków powinna być częścią całościowego projektu.

Podsumowanie jądra języka Python

Gratulacje! Niniejszy tekst kończy nasze omówienie jądra języka Python. Każdy, kto dotarł aż do końca, może się od teraz uznawać za Oficjalnego Programistę Pythona (i powinien dodać Pythona do swojego spisu umiejętności w życiorysie przy okazji następnego jego aktualnienia). Widzieliśmy już w zasadzie wszystko, co jest do zobaczenia w samym języku, a do tego nie raz na poziomie bardziej zaawansowanym, niż wielu aktywnych programistów Pythona robi to na początku. Omówiliśmy typy wbudowane, instrukcje oraz wyjątki, a także narzędzia wykorzystywane do budowania większych jednostek programów (takich, jak funkcje, moduły oraz klasy). Zapoznaliśmy się nawet z najważniejszymi zagadnieniami dotyczącymi między innymi projektowania, z programowaniem zorientowanym obiektywnie, a także architekturą programów.

Zbiór narzędzi Pythona

Od teraz nasza przyszła kariera programisty Pythona będzie w dużej mierze zależała od doskonalenia się w wykorzystywaniu *zbioru narzędzi* dostępnych dla programowania na poziomie aplikacji. Jak się okaże, jest to zajęcie ciągłe. Biblioteka standardowa zawiera na przykład setki modułów, a w Internecie można znaleźć jeszcze więcej narzędzi. Można spokojnie spędzić całą dekadę, próbując podnieść swoje umiejętności w zakresie wykorzystywania tych narzędzi, w szczególności dlatego, że ciągle pojawiają się nowe (wiem, co mówię!).

Mówiąc ogólnie, Python udostępnia następującą hierarchię zbiorów narzędzi:

Narzędzia wbudowane

Typy wbudowane, takie jak łańcuchy znaków, listy czy słowniki, sprawiają, że proste programy pisze się o wiele szybciej.

Rozszerzenia Pythona

Dla bardziej wymagających zadań możemy rozszerzyć Pythona, pisząc własne funkcje, moduły oraz klasy.

Skompilowane rozszerzenia

Choć nie omawialiśmy tego zagadnienia w książce, Pythona można również rozszerzać za pomocą modułów napisanych w innych zewnętrznych językach programowania, takich jak C czy C++.

Ponieważ Python układą swoje zbiory narzędzi w warstwy, możemy zadecydować, jak bardzo nasze programy mają się zagłębiać w tę hierarchię dla określonego zadania — narzędzia wbudowane możemy wykorzystywać w prostych skryptach, na potrzeby większych systemów

dodawać rozszerzenia napisane w Pythonie, natomiast do zadań zaawansowanych zastosować skompilowane rozszerzenia. W książce omówiliśmy jedynie pierwsze dwie kategorie, jednak powinno to w zupełności wystarczyć do rozpoczęcia całkiem rozsądnego programowania w Pythonie.

W tabeli 35.1 przedstawiono niektóre źródła funkcjonalności wbudowanej lub istniejącej i dostępnej dla programistów Pythona. Zapoznawanie się z niektórymi z pozostałych zagadnień zapewne zajmą nam resztę naszej kariery programisty Pythona. Aż do teraz większość naszych przykładów była niewielka i samodzielnna. Zostały one celowo napisane w taki sposób, byśmy mogli mistrzowsko opanować podstawy. Skoro już wiemy wszystko o jądrze samego języka, czas rozpocząć naukę tego, jak należy wykorzystywać wbudowane interfejsy Pythona w celu wykonania prawdziwej pracy. Wkrótce okaże się, że w prostym języku programowania, jakim jest Python, często wykonywane zadania są o wiele prostsze, niż można się tego spodziewać.

Tabela 35.1. Kategorie zbiorów narzędzi Pythona

Kategoria	Przykłady
Typy obiektów	Listy, słowniki, pliki, łańcuchy znaków
Funkcje	len, range, open
Wyjątki	IndexError, KeyError
Moduły	os, tkinter, pickle, re
Atrybuty	<code>__dict__</code> , <code>__name__</code> , <code>__class__</code>
Narzędzia peryferyjne	NumPy, SWIG, Jython, IronPython, Django

Narzędzia programistyczne przeznaczone do większych projektów

Po opanowaniu podstaw zobaczymy wkrótce, że nasze przykłady stanań się znacznie większe od tych, z którymi dotychczas eksperymentowaliśmy. Dla celów rozwijania większych systemów w samym Pythonie oraz w Internecie istnieje zbiór narzędzi programistycznych. Działanie niektórych z nich już widzieliśmy, o innych tylko wspomniałem. Poniżej znajduje się krótkie omówienie narzędzi najczęściej wykorzystywanych w tej dziedzinie.

PyDoc oraz łańcuchy znaków dokumentacji

Funkcja `help` z PyDoc oraz interfejsy HTML zostały omówione w rozdziale 15. PyDoc udostępnia system dokumentacji przeznaczony dla modułów oraz obiektów i integruje się z łańcuchami znaków dokumentacji Pythona. Jest standardową częścią systemu Pythona — więcej informacji na ten temat można znaleźć w dokumentacji. Należy również powrócić do źródeł dokumentacji wymienionych w rozdziale 4. w celu uzyskania informacji dotyczących innych zasobów informacyjnych Pythona.

PyChecker oraz PyLint

Ponieważ Python jest językiem tak dynamicznym, niektóre błędy programistyczne nie są zgłaszane, dopóki program nie będzie wykonywany (błędy składni są na przykład przechwytywane, kiedy plik jest wykonywany lub importowany). Nie jest to wielka wada — tak jak w większości języków programowania, oznacza to tylko tyle, że musimy przetestować kod przed udostępnieniem go. W najgorszym razie w przypadku Pythona zastępujemy fazę komplikacji fazą początkowych testów. Co więcej, dynamiczna natura Pythona,

automatyczne komunikaty o błędach oraz model wyjątków sprawiają, że łatwiej i szybciej jest znaleźć i naprawić błędy w Pythonie niż w innych językach programowania (w przeciwieństwie do języka C, Python nie przestaje działać w momencie wystąpienia błędu).

Systemy PyChecker oraz PyLint udostępniają obsługę przechwytywania dużego zbioru często spotykanych błędów z wyprzedzeniem, zanim skrypt zostanie wykonany. Pełnią podobną rolę jak program „lint” w programowaniu w języku C. Niektóre grupy programistów Pythona przepuszczają swój kod przez system PyChecker przed testowaniem lub publikowaniem w celu przechwycenia wszystkich potencjalnych problemów. Tak naprawdę biblioteka standardowa Pythona jest regularnie sprawdzana przez PyChecker przed opublikowaniem. PyChecker i PyLint to pakiety zewnętrzne na licencji open source. Można je znaleźć na stronie <http://www.python.org>, w witrynie PyPI lub za pomocą wyszukiwarki.

PyUnit (inaczej unittest)

W rozdziale 24. widzieliśmy, w jaki sposób do pliku Pythona dodaje się kod samosprawdzający, wykorzystując sztuczkę `__name__ == '__main__'` umieszczoną na dole pliku. Dla bardziej zaawansowanych celów testowania Python zawiera dwa narzędzia wspomagające sprawdzanie kodu. Pierwsze z nich, PyUnit (w dokumentacji biblioteki nazywane unittest), udostępnia zorientowaną obiektywnie platformę klas służącą do określania oraz dostosowywania do własnych potrzeb przypadków testowych oraz oczekiwanych rezultatów. Naśladuje ona platformę JUnit przeznaczoną dla języka Java. Jest to wyszukany system testów jednostkowych oparty na klasach; więcej informacji na jego temat można znaleźć w dokumentacji biblioteki Pythona.

doctest

Moduł biblioteki standardowej `doctest` udostępnia drugie i prostsze podejście do testów regresyjnych. Oparty jest na łańcuchach znaków dokumentacji Pythona. By użyć `doctest`, należy w przybliżeniu wyciąć i wkleić log testowej sesji interaktywnej do łańcuchów znaków w plikach źródłowych. Moduł `doctest` dokonuje ekstrakcji łańcuchów znaków dokumentacji, pobiera z nich przypadki testowe oraz wyniki i wykonuje testy kolejny raz w celu zweryfikowania oczekiwanych wyników. Operacje `doctest` można na różne sposoby dostosować do własnych potrzeb; więcej informacji na ten temat można znaleźć w dokumentacji biblioteki.

Zintegrowane środowiska programistyczne

Zintegrowane środowiska programistyczne (IDE) przeznaczone dla Pythona omawialiśmy w rozdziale 3. IDE takie, jak IDLE udostępniają środowisko graficzne służące do edycji, wykonywania, debugowania oraz przeglądania naszych programów napisanych w Pythonie. Niektóre zaawansowane środowiska programistyczne (takie jak Eclipse, Komodo, NetBeans czy Wing IDE) mogą również obsługiwać dodatkowe zadania programistyczne, w tym integrację z systemami kontroli wersji, refaktoryzację kodu czy narzędzia do zarządzania projektami. Więcej informacji na temat dostępnych zintegrowanych środowisk programistycznych oraz narzędzi do budowania graficznych interfejsów użytkownika można znaleźć w rozdziale 3., na stronie poświęconej edytorom tekstu w witrynie <http://www.python.org> oraz za pomocą wyszukiwarki internetowej.

Programy profilujące

Ponieważ Python jest językiem wysokopoziomowym oraz dynamicznym, kwestie związane z wydajnością poznane dzięki doświadczeniu z innymi językami programowania zazwyczaj nie mają zastosowania do kodu napisanego w tym języku. By naprawdę wyizolować „wąskie gardła” dla wydajności kodu, musimy za pomocą narzędzi zegarowych dodać

logikę pomiarową dostępną w modułach `time` oraz `timeit` lub wykonać swój kod pod modułem `profile`. Przykład działania modułów pomiarowych widzieliśmy przy okazji porównywania szybkości narzędzi iteracyjnych w rozdziale 20. Profilowanie jest zazwyczaj pierwszym krokiem ku optymalizacji kodu — kod jest profilowany w celu wyizolowania „wąskich gardel”, a następnie dokonuje się pomiarów dla alternatywnych rozwiązań.

Moduł `profile` pochodzi z biblioteki standardowej i implementuje program profilujący dla kodu źródłowego napisanego w Pythonie. Wykonuje on dostarczony łańcuch kodu (na przykład operację importowania pliku skryptu lub wywołanie funkcji), a następnie, domyślnie, wyświetla w standardowym strumieniu wyjścia raport podający statystyki wydajności — liczbę wywołań każdej funkcji, czas spędzony w każdej funkcji i dużo więcej.

Moduł `profile` można wykonać w postaci skryptu lub zimportować. Można go także na różne sposoby dostosowywać do własnych potrzeb. Może on na przykład zapisywać statystyki wykonywania do pliku, który będzie później analizowany za pomocą modułu `pstats`. W celu dokonania interaktywnego profilowania należy zimportować moduł `profile` i wywołać `profile.run('kod')`, przekazując kod, który chcemy sprofilować, w postaci łańcucha znaków (na przykład wywołanie funkcji, zimportowanie całego pliku). W celu wykonania profilowania z systemowego wiersza poleceń powłoki należy skorzystać z polecenia `python -m profile main.py args...` (więcej informacji na temat tego formatu znajduje się w dodatku A). Inne opcje profilowania omówione są w dokumentacji biblioteki standardowej Pythona. Moduł `cProfile` ma na przykład identyczne interfejsy jak `profile`, jednak jego wykonanie jest mniej kosztowne, przez co może się lepiej nadawać do zastosowania w przypadku długo działających programów.

Debugery

W rozdziale 3. omówiliśmy także opcje debugowania kodu (ramka „Debugowanie kodu w Pythonie”). Słownem przypomnienia, większość środowisk programistycznych przeznaczonych dla Pythona obsługuje debugowanie oparte na graficznym interfejsie użytkownika, a biblioteka standardowa Pythona udostępnia również moduł debugowania kodu źródłowego o nazwie `pdb`. Moduł ten udostępnia interfejs wiersza poleceń i działa w przybliżeniu jak popularne debugery języka C (na przykład `dbx, gdb`).

Podobnie jak narzędzie do profilowania, debugger `pdb` możemy wykorzystywać albo interaktywnie, albo z wiersza poleceń; można go także zimportować i wywołać z programu napisanego w Pythonie. By użyć go w sposób interaktywny, importujemy moduł, zaczynamy wykonywać kod, wywołując funkcję `pdb` (na przykład `pdb.run("main()")`), a następnie wpisujemy polecenia debugowania z interaktywnej zachęty `pdb`. W celu uruchomienia `pdb` z systemowego wiersza poleceń powłoki należy użyć polecenia `python -m pdb main.py args...` (więcej informacji na temat tego formatu znajduje się w dodatku A). Moduł `pdb` zawiera również przydatne narzędzie do analizy postmortem, wywołanie `pdm.pm()`, które włącza debugger po napotkaniu wyjątku.

Ponieważ zintegrowane środowiska programistyczne takie jak IDLE zawierają interfejsy do debugowania za pomocą klikania, moduł `pdb` nie jest dzisiaj narzędziem krytycznym, z wyjątkiem sytuacji, gdy GUI nie jest dostępne albo pożądany jest wyższy stopień kontroli. Więcej wskazówek na temat wykorzystywania graficznego interfejsu debugera w IDLE można znaleźć w rozdziale 3. Tak naprawdę w praktyce ani `pdb`, ani debugery ze zintegrowanych środowisk programistycznych nie są zbyt często wykorzystywane. Jak pisaliśmy

w rozdziale 3., większość programistów albo wstawia po prostu do kodu instrukcję `print`, albo czyta komunikaty o błędach Pythona (nie jest to może najbardziej nowoczesne rozwiązanie, ale jego praktyczność decyduje o jego przewadze w świecie Pythona).

Opcje udostępniania kodu

W rozdziale 2. wprowadziliśmy narzędzia służące do pakowania programów Pythona. `PyInstaller`, `py2exe` oraz `freeze` mogą pakować kod bajtowy oraz maszynę wirtualną Pythona w zamrożone pliki binarne będące samodzielnymi plikami wykonywalnymi niewymagającymi zainstalowania Pythona na komputerze docelowym i w pełni ukrywającymi kod systemu. Dodatkowo w rozdziale 2. książki widzieliśmy, że programy napisane w Pythonie mogą być udostępniane w postaci źródeł (`.py`) lub kodu bajtowego (`.pyc`), a punkty zaczepienia operacji importowania obsługują specjalne techniki pakowania, takie jak automatyczna ekstrakcja plików `.zip` oraz szyfrowanie kodu bajtowego.

Przelotnie zapoznaliśmy się z modułami biblioteki standardowej `distutils`, które udostępniają opcje pakowania modułów oraz pakietów Pythona, a także rozszerzeń w języku C.Więcej informacji na ten temat można znaleźć w dokumentacji Pythona. Nowszy zewnętrzny system pakowania o nazwie „eggs” udostępnia kolejną alternatywę powiązaną z zależnościami; więcej informacji na jego temat można znaleźć w Internecie.

Opcje optymalizacji

Istnieje kilka możliwości optymalizacji programów. Opisany w rozdziale 2. system Psyco udostępnia kompilator JIT służący do przekładania kodu bajtowego Pythona na binarny kod maszynowy, natomiast Shedskin oferuje translator z Pythona na język C++. Czasami można się również spotkać ze zoptymalizowanymi plikami z kodem bajtowym o rozszerzeniu `.pyo`, które są generowane oraz wykonywane z opcją wiersza poleceń `-O` (omówioną w rozdziałach 21. oraz 33.). Ponieważ daje to jednak naprawdę skromną przewagę z zakresie wydajności, nie jest wykorzystywane zbyt często.

Jako ostatnią deskę ratunku można potraktować przeniesienie części programu do języka skompilowanego, takiego jak C, w celu podniesienia ich wydajności. Więcej informacji na temat rozszerzeń języka C można znaleźć w książce *Programming Python*, a także dokumentacji Pythona. Mówiąc ogólnie, szybkość Pythona poprawia się z czasem, dlatego kiedy tylko jest to możliwe, należy uaktualniać Pythona do szyszej wersji.

Inne wskazówki przeznaczone dla większych projektów

W tekście spotkaliśmy wreszcie wiele różnych opcji języka, które stają się bardziej przydatne w miarę tworzenia większych projektów. Pośród nich są między innymi pakiety modułów (rozdział 23.), wyjątki oparte na klasach (rozdział 33.), atrybuty pseudoprzywatne klas (rozdział 30.), łańcuchy znaków dokumentacji (rozdział 15.), pliki konfiguracyjne ścieżki modułów (rozdział 21.), ukrywanie zmiennych przed instrukcjami `from *` za pomocą list `_all_` oraz nazw w stylu `_X` (rozdział 24.), dodawanie kodu samo-sprawdzającego za pomocą sztuczki `_name_ == '__main__'` (rozdział 24.), wykorzystywanie często stosowanych reguł projektowania dla funkcji oraz modułów (rozdziały 17., 19. oraz 24.) czy wykorzystanie zorientowanych obiektywo wzorców programowania (rozdział 30. i inne).

By dowiedzieć się więcej na temat dostępnych w Internecie narzędzi programowania w Pythonie na dużą skalę, należy koniecznie przejrzeć strony PyPI w witrynie <http://www.python.org> czy po prostu Internet.

Podsumowanie rozdziału

Niniejszy rozdział zakończył część książki poświęconą wyjątkom omówieniem instrukcji związanych z tym zagadnieniem. Przyjrzaliśmy się często stosowanym przypadkom użycia wyjątków, a także zapoznaliśmy się z krótkim omówieniem często wykorzystywanych narzędzi programistycznych.

Rozdział ten kończy również główną część książki. W tym punkcie Czytelnikom przedstawiono pełny zbiór narzędzi Pythona wykorzystywany przez większość programistów. Tak naprawdę każda osoba, która dotarła aż tutaj, może się swobodnie uznawać za *oficjalnego programistę Pythona*. Przy najbliższej okazji należy rozważyć zakupienie koszulki z takim napisem.

Kolejna i zarazem ostatnia część książki to zbiór rozdziałów omawiających zagadnienia zaawansowane, które jednak mieszą się w kategoriach jądra języka. Rozdziały te są *lekturą opcjonalną*, ponieważ nie każdy programista Pythona musi zagłębiać się w przedstawione tam zagadnienia. Tak naprawdę większość osób może zatrzymać się już tutaj i zacząć poznawać role Pythona w rozmaitych dziedzinach zastosowania. Szczerze mówiąc, biblioteki aplikacji wydają się w praktyce bardziej istotnymi zagadnieniami niż zaawansowane (i dla niektórych osób egzotyczne) opcje języka.

Z drugiej strony, dla osób, które muszą zajmować się takimi kwestiami, jak dane binarne czy Unicode, mają do czynienia z narzędziami służącymi do budowania API, takimi jak deskryptory, dekoratory oraz metaklasy, albo po prostu chcą pogłębić swoją wiedzę, kolejna część książki będzie dobrym punktem wyjścia. Większe przykłady z tej części umożliwią także zobaczenie zastosowania omówionych już koncepcji w nieco bardziej realistyczny sposób.

Ponieważ jest to koniec głównej części książki, nie będę przesadzał z trudnością quizu — tym razem zawiera on tylko jedno pytanie. Jak zawsze należy jednak pamiętać o wykonaniu ćwiczeń końcowych w celu utrwalenia tego, czego nauczyliśmy się w ostatnich kilku rozdziałach. Ponieważ kolejna część jest lekturą opcjonalną, jest to ostatnia seria ćwiczeń kończących część książki. Osoby, które chcą zobaczyć przykłady połączenia tego, czego się nauczyliśmy, w prawdziwych skryptach pochodzących z często stosowanych aplikacji, odsyłam do rozwiązania ćwiczenia 4. w dodatku B.

Sprawdź swoją wiedzę — quiz

1. (To pytanie jest powtórką z pierwszego quizu z rozdziału 1. — mówiłem, że będzie łatwo! :-)). Dlaczego mielonka pojawia się w tak wielu przykładach kodu Pythona w książce oraz Internecie?

Sprawdź swoją wiedzę — odpowiedzi

1. Ponieważ Python został nazwany na cześć brytyjskiej grupy komediowej Monty Python (na podstawie przeprowadzonych w czasie moich szkoleń ankiet mogę powiedzieć, że w świecie Pythona jest to aż zbyt dobrze utrzymywana tajemnica!). Odniesienie do mielonki (ang. *spam*) pochodzi ze skeczu grupy Monty Python, w którym para próbującą

zamówić coś do jedzenia w kafeterii zostaje zagłuszona przez chór Wikingów śpiewających piosenkę o mielonce. I gdybym mógł wstawić tutaj plik dźwiękowy z tą piosenką, z pewnością bym to zrobił...

Sprawdź swoją wiedzę — ćwiczenia do części siódmej

Ponieważ dotarliśmy do końca tej części książki, czas na kilka ćwiczeń związanych z wyjątkami, które pozwolą nam sprawdzić naszą znajomość podstaw. Wyjątki są naprawdę prostym narzędziem. Jeśli rozwiążemy poniższe ćwiczenia, oznacza to, że doskonale je opanowaliśmy.

Rozwiązania można znaleźć w podrozdziale „Część VII Wyjątki oraz narzędzia” w dodatku B.

1. *Instrukcja try/except.* Należy napisać funkcję o nazwie `oops` w jawnym sposobie zgłaszającą po wywołaniu wyjątek `IndexError`. Później należy napisać kolejną funkcję wywołującą `oops` wewnętrz instrukcji `try/except` w celu przechwycenia błędu. Co się stanie, jeśli zmienimy funkcję `oops` w taki sposób, by zgłaszała ona wyjątek `KeyError` zamiast `IndexError`? Skąd pochodzą nazwy `KeyError` oraz `IndexError`? Wskazówka: warto przypomnieć, że wszystkie nazwy bez składni kwalifikującej pochodzą z jednego z czterech zakresów.
2. *Obiekty wyjątków oraz listy.* Należy zmodyfikować napisaną wyżej funkcję `oops` w taki sposób, by zgłaszała zdefiniowany przez nas wyjątek o nazwie `MyError`. Wyjątek należy zidentyfikować za pomocą klasy. Następnie należy rozszerzyć instrukcję `try` w funkcji przechwytyjącej w taki sposób, by oprócz `IndexError` przechwytywała ona ten wyjątek oraz jego instancje, a także wyświetlała przechwytywaną instancję.
3. *Obsługa błędów.* Należy napisać funkcję o nazwie `safe(func, *args)`, która wykonuje dowolną funkcję z dowolną liczbą argumentów za pomocą składni wywołania z dowolną liczbą argumentów `*nazwa`, przechwytuje każdy wyjątek zgłoszony w czasie wykonywania funkcji i wyświetla ten wyjątek za pomocą wywołania `exc_info` z modułu `sys`. Następnie należy wykorzystać funkcję `safe` do wykonania funkcji `oops` z pierwszych dwóch ćwiczeń. Funkcję `safe` należy umieścić w pliku modułu o nazwie `tools.py` i przekazać do niej interaktywnie funkcję `oops`. Jaki rodzaj komunikatu o błędzie otrzymamy? Wreszcie należy rozbudować funkcję `safe` w taki sposób, by kiedy wystąpi błąd, wyświetlała ona również ślad stosu Pythona, wywołując wbudowaną funkcję `print_exc` ze standardowego modułu `traceback` (szczegółowe informacje na jej temat można znaleźć w dokumentacji biblioteki Pythona).
4. *Przykłady do samodzielnego przestudiowania.* Pod koniec dodatku B zamieściłem kilka przykładowych skryptów utworzonych jako ćwiczenia grupowe na kursach z Pythona, by każdy przestudiował je samodzielnie i wykonał w połączeniu ze zbiorem dokumentacji Pythona. Nie są one opisane i wykorzystują narzędzia z biblioteki standardowej Pythona, z którymi trzeba będzie zapoznać się samodzielnie. Wielu osobom powinno to pomóc zobaczyć, w jaki sposób zagadnienia omówione w książce łączą się ze sobą w prawdziwych programach. Jeśli zaostrzy to nasz apetyt, ogromną liczbę większych i bardziej zaawansowanych programów w Pythonie na poziomie aplikacji można znaleźć w książkach będących kontynuacją tej, takich jak *Programming Python*, a także w Internecie.

Zagadnienia zaawansowane

Łańcuchy znaków Unicode oraz łańcuchy bajtowe

W rozdziale poświęconym *łańcuchom znaków* w części o typach podstawowych Pythona (rozdział 7.) celowo ograniczyłem omawiane zagadnienia do tych, które muszą znać wszyscy programiści Pythona. Ponieważ większość z nich pracuje z prostymi formami tekstu, takimi jak ASCII, mogą oni z radością wykorzystywać podstawowy typ `str` Pythona i powiązane z nim operacje, nie musząc zajmować się bardziej zaawansowanymi koncepcjami związanymi z łańcuchami znaków. Tak naprawdę ci programiści mogą w dużej mierze zignorować zmiany łańcuchów znaków w Pythonie 3.0 i kontynuować wykorzystanie tego typu w taki sposób jak w przeszłości.

Z drugiej istnieją strony programiści zajmujący się bardziej wyspecjalizowanymi typami danych, jak znaki spoza zbioru ASCII czy zawartość plików graficznych. Na ich potrzeby (oraz potrzeby innych osób, które mogą kiedyś do nich dołączyć) w niniejszym rozdziale uzupełnimy informacje o łańcuchach znaków Pythona i przyjrzymy się pewnym bardziej zaawansowanym koncepcjom z nimi związanym.

W szczególności omówimy podstawy obsługi *tekstu Unicode* w Pythonie — łańcuchów znaków wykorzystywanych w aplikacjach zinternacjonalizowanych — a także *danych binarnych* — łańcuchów reprezentujących bezwzględne wartości bajtowe. Jak zobaczymy, zaawansowane zagadnienia związane z łańcuchami znaków w nowszych wersjach Pythona zostały zróżnicowane:

- *Python 3.0* udostępnia alternatywny typ łańcucha znaków dla danych binarnych i obsługuje tekst Unicode w normalnym typie łańcucha znaków (ASCII traktowane jest jako prosty rodzaj Unicode).
- *Python 2.6* udostępnia alternatywny typ łańcucha znaków dla tekstu Unicode poza ASCII, a w normalnym typie łańcucha znaków obsługuje zarówno prosty tekst, jak i dane binarne.

Dodatkowo, ponieważ model łańcuchów znaków Pythona ma bezpośredni wpływ na to, w jaki sposób przetwarzamy *pliki* spoza ASCII, omówimy tutaj podstawy również tego zagadnienia. Wreszcie przyjrzymy się krótko niektórym bardziej zaawansowanym *narzędziom* do obsługi łańcuchów znaków oraz danych binarnych, takim jak dopasowywanie do wzorców, serializacja obiektów, pakowanie danych binarnych, analiza składniowa XML, a także sposobom, w jakie na narzędzia te wpłynęły zmiany łańcuchów znaków z wersji 3.0.

Oficjalnie jest to rozdział poświęcony zagadnieniom zaawansowanym, ponieważ nie wszyscy programiści będą musieli zagłębić się w świat kodowania Unicode czy danych binarnych. Jeśli jednak ktoś będzie chciał to kiedyś zrobić, szybko zobaczy, że model łańcuchów znaków Pythona obsługuje to, co trzeba.

Zmiany w łańcuchach znaków w Pythonie 3.0

Jedną z najbardziej zauważalnych zmian w wersji 3.0 jest zmiana w typach obiektów łańcuchów znaków. W skrócie typy `str` i `unicode` z wersji 2.X złączyły się w typy `str` oraz `bytes` z wersji 3.0; dodany został nowy zmienny typ `bytearray`. Typ `bytearray` jest także dostępny w Pythonie 2.6 (choć nie we wcześniejszych wersjach), jednak został tam przeniesiony z wersji 3.0 i nie rozróżnia w sposób tak wyraźny tekstu i zawartości binarnej.

Zmiany te mogą w znacznym stopniu wpływać na nasz kod, zwłaszcza jeśli przetwarzamy dane Unicode lub binarne. Mówiąc ogólnie, to, czy zagadnienie to ma dla nas znaczenie, w dużej mierze uzależnione jest od tego, do której z poniższych kategorii się zaliczamy:

- Jeśli zajmujemy się *tekstem Unicode* spoza zbioru ASCII — na przykład w kontekście międzynarodizowanych aplikacji czy wyniku niektórych analizatorów składniowych XML — obsługa kodowania tekstu w 3.0 zmieni się dla nas, jednak jednocześnie będzie też bardziej bezpośrednią, nieroóżnialną i dostępna w porównaniu z sytuacją z Pythona 2.6.
- Jeśli zajmujemy się *danymi binarnymi* — na przykład w postaci plików graficznych bądź audio czy spakowanych danych przetwarzanych za pomocą modułu `struct` — będziemy musieli opanować nowy obiekt `bytes` z Pythona 3.0, a także inne i ostrzejsze rozróżnienie pomiędzy danymi i plikami tekstowymi a binarnymi.
- Jeśli nie mieścimy się w *żadnej* z dwóch poprzednich kategorii, łańcuchów znaków w Pythonie 3.0 będziemy w dużej mierze używać w taki sam sposób jak w wersji 2.6, z ogólnym typem łańcucha znaków `str`, plikami tekstowymi i wszystkimi znymi operacjami na tego typu obiektach, jakie już omówiliśmy. Wykorzystywane przez nas łańcuchy znaków będą kodowane i rozkodowywane za pomocą domyślnego rodzaju kodowania wykorzystywanej platformy (na przykład UTF-8 w systemie Windows — jeśli ktoś ma ochotę to sprawdzić, funkcja `sys.getdefaultencoding()` zwraca wartość domyślną); najprawdopodobniej nawet tego jednak nie zauważymy.

Innymi słowy, jeśli pisany przez nas tekst jest zawsze ASCII, wystarczą nam normalne obiekty łańcuchów znaków i plików tekstowych, a większość omawianych tu zagadnień możemy pominać. Jak zobaczymy za chwilę, ASCII jest po prostu typem Unicode i podziobiem innych rodzajów kodowania, dlatego jeśli nasze programy przetwarzają tekst ASCII, operacje na łańcuchach znaków i plikach po prostu będą działać.

Mimo to jednak, nawet jeśli mieścimy się w ostatniej z trzech przedstawionych wyżej kategorii, podstawowe zrozumienie modelu łańcuchów znaków Pythona z wersji 3.0 może pomóc zarówno wyjaśniać pewne zachowania tego typu, jak i ułatwić późniejsze opanowanie zagadnień związanych z Unicode czy danymi binarnymi w przyszłości.

Obsługa Unicode oraz danych binarnych z Pythona 3.0 jest dostępna w wersji 2.6, jednak w innej postaci. Choć w niniejszym rozdziale skupimy się przede wszystkim na typach łańcuchów znaków z wersji 3.0, przy okazji przedstawimy także niektóre różnice z Pythonem 2.6. Bez względu na wykorzystawaną wersję tego języka omawiane tutaj narzędzia mogą mieć duże znaczenie w wielu typach programów.

Podstawy łańcuchów znaków

Zanim przyjrzymy się jakiemukolwiek kodowi, zacznijmy od ogólnego omówienia modelu łańcuchów znaków Pythona. By zrozumieć, dlaczego wersja 3.0 została tak zmodyfikowana w tym zakresie, musimy zacząć od omówienia tego, w jaki sposób znaki są naprawdę reprezentowane w komputerze.

Kodowanie znaków

Większość programistów myśli o łańcuchach znaków jako o seriach znaków wykorzystywanych do reprezentowania danych tekstowych. Sposób przechowywania znaków w pamięci komputera może się jednak różnić w zależności od zbioru znaków, jaki musi zostać zapisany.

Standard *ASCII* powstał w Stanach Zjednoczonych Ameryki i definiuje tekstowe łańcuchy znaków w rozumieniu większości amerykańskich programistów. ASCII definiuje znaki od 0 do 127 i pozwala, by każdy znak mógł być przechowywany w jednym 8-bitowym bajcie (z którego tak naprawdę wykorzystywanych jest jedynie siedem bitów). Przykładowo standard ASCII odwzorowuje znak 'a' na wartość liczbową 97 (0x61 w notacji szesnastkowej), która zostaje przechowana w pojedynczym bajcie w pamięci i plikach. Jeśli chcemy zobaczyć, jak to działa, funkcja wbudowana Pythona `ord` zwraca wartość binarną znaku, natomiast funkcja `chr` zwraca znak dla podanej wartości kodu liczbowego:

```
>>> ord('a')                                # 'a' jest w ASCII bajtem z wartością binarną 97
97
>>> hex(97)
'0x61'
>>> chr(97)                                 # Wartość binarna 97 oznacza znak 'a'
'a'
```

Czasami jednak jeden bajt na znak nie wystarczy. Różne symbole i litery ze znakami diakrytycznymi nie mieścią się w zakresie znaków zdefiniowanych przez ASCII. By uwzględnić znaki specjalne, niektóre standardy dopuszczają, by w 8-bitowym bajcie wszystkie możliwe wartości od 0 do 255 reprezentowały znaki, natomiast wartości od 128 do 255 (poza zakresem ASCII) były znakami specjalnymi. Jeden z takich standardów, znany pod nazwą *Latin-1*, wykorzystywany jest powszechnie w Europie Zachodniej. W Latin-1 kody znaków powyżej 127 przypisane są do liter ze znakami diakrytycznymi i innych znaków specjalnych. Na przykład znak przypisany do wartości bajtowej 196 jest specjalną literą spoza zbioru ASCII:

```
>>> 0xC4
196
>>> chr(196)
'Ã'
```

Standard ten pozwala na reprezentowanie szerokiej gamy dodatkowych znaków specjalnych. Mimo to niektóre alfabety definiują liczbę dodatkowych znaków uniemożliwiającą reprezentowanie ich wszystkich jako pojedynczych bajtów. Standard *Unicode* pozwala na większą elastyczność. W przypadku tekstu Unicode każdy znak może być reprezentowany za pomocą kilku bajtów. Unicode zazwyczaj wykorzystywany jest w programach *internacjonalizowanych* i reprezentuje zbiory znaków europejskich oraz azjatyckich mające więcej znaków, niż da się reprezentować za pomocą bajtów 8-bitowych.

By przechować taki tekst w pamięci komputera, mówimy, że znaki tłumaczone są na „surowe” bajty i z powrotem za pomocą kodowania — reguł przekładania łańcucha znaków Unicode na

sekwencję bajtów, a także ekstrakcji łańcucha znaków z sekwencji bajtów. Z punktu widzenia procedur takie tłumaczenie pomiędzy bajtami a łańcuchami znaków definiuje się za pomocą dwóch pojęć:

- *Kodowanie* to proces przekładania łańcucha znaków na jego odpowiednik w postaci surowych bajtów, zgodnie z pożądaną nazwą kodowania.
- *Dekodowanie (odkodowanie)* to proces przekładania łańcucha surowych bajtów na postać łańcucha znaków, zgodnie z pożądaną nazwą kodowania.

Oznacza to zatem, że *kodujemy* łańcuch znaków na surowe bajty i *dekodujemy* surowe bajty na łańcuchy znaków. W przypadku niektórych typów kodowania sam proces tłumaczenia jest trywialny — przykładowo ASCII i Latin-1 odwzorowują każdy znak na pojedynczy bajt, dzięki czemu żaden przekład nie jest konieczny. W przypadku innych typów kodowania odwzorowania mogą być bardziej skomplikowane i wymagać kilku bajtów na znak.

Szeroko wykorzystywane kodowanie *UTF-8* pozwala na przykład na reprezentowanie szerokiej gamy znaków, wykorzystując rozwiążanie ze zmienną liczbą bajtów. Kody znaków mniejsze od 128 reprezentowane są przez pojedynczy bajt; kody pomiędzy 128 a 0x7ff (2047) zamieniane są na dwa bajty, gdzie każdy bajt ma wartość pomiędzy 128 a 255. Kody większe od 0x7ff zamieniane są w sekwencje trzy- lub czterobajtowe o wartościach pomiędzy 128 a 255. Dzięki temu łańcuchy znaków ASCII są zwięzłe, omijane są problemy dotyczące kolejności bajtów, a także unikane bajty zerowe, które mogą spowodować problemy dla bibliotek języka C oraz pracy w sieci.

Ponieważ tabele odwzorowań znaków różnych typów kodowania z uwagi na zgodność przypisują znaki do tych samych kodów, ASCII jest *podzbiorem* zarówno Latin-1, jak i UTF-8. Oznacza to, że poprawny łańcuch znaków ASCII jest także poprawnym łańcuchem znaków w kodowaniu Latin-1 oraz UTF-8. Tak samo jest również, gdy dane przechowywane są w plikach — każdy plik ASCII jest poprawnym plikiem UTF-8, ponieważ ASCII jest 7-bitowym podzbiorem UTF-8.

I odwrotnie, kodowanie UTF-8 jest zgodne binarnie z ASCII dla wszystkich kodów znaków mniejszych od 128. Latin-1 i UTF-8 pozwalały po prostu na dodatkowe znaki — Latin-1 na znaki odwzorowane na wartości od 128 do 255 w ramach bajta, a UTF-8 na znaki, które mogą być reprezentowane za pomocą kilku bajtów. Również inne typy kodowania pozwalały na podobne stosowanie szerszych zbiorów znaków, jednak wszystkie wymienione — ASCII, Latin-1 i UTF-8 — a także wiele innych uznawane są za Unicode.

W przypadku programistów Pythona kodowanie określane jest jako łańcuch znaków zawierający jego nazwę. Python zawiera około stu różnych rodzajów kodowania — pełną listę można znaleźć w dokumentacji biblioteki tego języka. Zimportowanie modułu `encodings` i wykonanie `help(encodings)` pokazuje wiele z nazw typów kodowania. Przykładowo *latin-1*, *iso_8859_1* i *8859* są synonimami tego samego kodowania — Latin-1. Do kodowania powrócimy w dalszej części niniejszego rozdziału, przy omawianiu technik zapisywania łańcuchów znaków Unicode w skrypcie.

Więcej informacji na temat Unicode można znaleźć w dokumentacji biblioteki standardowej Pythona. W dziale „Python HOWTOs” znajduje się artykuł „Unicode HOWTO” zawierający dodatkowe informacje, które tutaj, z uwagi na brak miejsca, pominiemy.

Typy łańcuchów znaków Pythona

Przechodząc do konkretów, Python udostępnia typy danych łańcuchów znaków reprezentujące w naszych skryptach tekst. Typy łańcuchów znaków wykorzystywane w skryptach uzałożone są od wykorzystywanej wersji Pythona.

Python 2.X ma ogólny typ łańcucha znaków reprezentujący dane binarne oraz prosty tekst 8-bitowy, taki jak ASCII, wraz z odrębnym typem reprezentującym wielobajtowy tekst Unicode:

- str reprezentuje tekst 8-bitowy oraz dane binarne,
- unicode reprezentuje tekst Unicode.

Dwa typy łańcuchów znaków w Pythonie 2.X różnią się od siebie (unicode zezwala na dodatkową wielkość znaków i zawiera obsługę kodowania i dekodowania), jednak zbiory ich działań w dużej mierze pokrywają się. Typ łańcucha znaków str w wersji 2.0 wykorzystywany jest w przypadku tekstu, który można reprezentować za pomocą bajtów 8-bitowych, a także danych binarnych reprezentujących bezwzględne wartości bajtów.

W Pythonie 3.X można natomiast znaleźć trzy typy obiektów łańcuchów znaków — jeden na potrzeby danych tekstowych i dwa przeznaczone dla danych binarnych:

- str reprezentuje tekst Unicode (zarówno 8-bitowy, jak i większy),
- bytes reprezentuje dane binarne,
- bytearray jest zmienną odmianą typu bytes.

Jak wspomniano wcześniej, typ bytearray jest również dostępny w Pythonie 2.6, jednak został tam po prostu przeniesiony z wersji 3.0, ma mniej działań charakterystycznych dla swojej zawartości i generalnie uznawany jest za typ z Pythona 3.0.

Wszystkie typy łańcuchów znaków z wersji 3.0 obsługują podobny zbiór operacji, jednak pełnią one odmienne role. Najważniejszą zmianą leżącą u podstaw zmian z wersji 3.0 było połączenie typów normalnego łańcucha znaków i łańcucha znaków Unicode z Pythona 2.X w jeden typ obsługujący tekst zwykły i Unicode. Programiści chcieli odejść od dychotomii łańcuchów znaków z wersji 2.X i sprawić, by przetwarzanie Unicode stało się bardziej naturalne. Biorąc pod uwagę to, że ASCII i inne typy tekstu 8-bitowego tak naprawdę są prostym rodzajem Unicode, takie połączenie wydaje się logiczne i rozsądne.

By to uzyskać, typ str z Pythona 3.0 zdefiniowany został jako *niezmieniona sekwencja znaków* (niekoniecznie bajtów), która może być albo normalnym tekstem jak ASCII, z jednym bajtem na znak, albo bardziej rozbudowanym zbiorem tekstu takim jak UTF-8 w Unicode, który może zawierać znaki z większą liczbą bajtów. Łańcuchy znaków przetwarzane przez nasz skrypt za pomocą tego typu są kodowane zgodnie z wartością domyślną wykorzystywanej platformy. Nazwy kodowania można także podać w sposób jawny, tak by przekładać obiekty str za pomocą różnych schematów — zarówno w pamięci, jak i przy przenoszeniu ich do plików i z nich.

Choć nowy typ str z Pythona 3.0 spowodował zamierzone połączenie zwykłych łańcuchów znaków i łańcuchów znaków Unicode, wiele programów nadal musi przetwarzać surowe dane binarne, które nie są kodowane zgodnie z jakimkolwiek formatem tekstowym. Pliki graficzne czy audio, a także spakowane dane wykorzystywane w interfejsach z urządzeniami

czy programami języka C, które możemy przetwarzać za pomocą modułu `struct` Pythona, mieścią się w tej właśnie kategorii. By móc obsłużyć przetwarzanie prawdziwych danych binarnych, wprowadzono zatem nowy typ danych o nazwie `bytes`.

W wersji 2.X ogólny typ `str` pełnił rolę danych binarnych, ponieważ łańcuchy znaków były po prostu sekwencjami bajtów (odróżnny typ `unicode` zajmował się łańcuchami znaków wielobajtowych). W Pythonie 3.0 typ `bytes` zdefiniowany został jako *niezmienna sekwencja 8-bitowych liczb całkowitych* reprezentująca bezwzględne wartości bajtów. Co więcej, typ `bytes` z 3.0 obsługuje prawie te same operacje co typ `str` — obejmuje to metody łańcuchów znaków, działania na sekwencjach, a nawet dopasowywanie do wzorców z modułu `re`, jednak nie formatowanie łańcuchów znaków.

Obiekt `bytes` z Pythona 3.0 jest tak naprawdę sekwencją niewielkich liczb całkowitych, z których każda mieści się w przedziale od 0 do 255. Indeksowanie obiektu `bytes` zwraca obiekt `int`, wycinek zwraca inny obiekt `bytes`, natomiast wykonanie wbudowanej funkcji `list` zwraca listę liczb całkowitych, a nie znaków. Po przetworzeniu za pomocą operacji zakładających istnienie znaków w przypadku `bytes` zakłada się, że zawartość tych obiektów to bajty zakodowane w ASCII (metoda `isalpha` zakłada, że każdy bajt jest kodem znaku ASCII). Co więcej, obiekty `bytes` wyświetlane są dla wygody jako łańcuchy znaków, a nie liczby całkowite.

A skoro już przy tym jesteśmy, programiści Pythona dodali również w wersji 3.0 typ `bytearray`. Typ `bytearray` jest odmianą `bytes`, która jest *zmienna*, dzięki czemu obsługuje modyfikację w miejscu. Obsługuje on zwykłe operacje na łańcuchach znaków, te same co typy `str` i `bytes`, a także wiele z operacji modyfikacji w miejscu obsługiwanych przez listy (na przykład metody `append` i `extend`, przypisywanie do indeksów). Zakładając, że łańcuchy znaków mogą być traktowane jako surowe bajty, `bytearray` wreszcie dodaje możliwość dokonywania bezpośrednich zmian w miejscu w przypadku danych będących łańcuchami znaków — coś, co było niemożliwe do uzyskania w Pythonie 2 bez konwersji na zmienny typ danych i nie jest obsługiwane przez typy `str` oraz `bytes` z Pythona 3.0.

Choć wersje 2.6 i 3.0 oferują w dużej mierze tę samą funkcjonalność, w inny sposób ją łączą. Tak naprawdę odwzorowanie z typów łańcuchów znaków Pythona 2.6 na typy z Pythona 3.0 nie jest bezpośrednie — `str` z 2.6 równy jest `str` i `bytes` z 3.0, natomiast `str` z 3.0 równy jest `str` i `unicode` z 2.6. Co więcej, unikalna jest zmienność typu `bytearray` z wersji 3.0.

W praktyce ta asymetria nie jest aż tak skomplikowana, jak mogłyby się wydawać. Sprowadza się do następującej kwestii: w wersji 2.6 użyjemy `str` w przypadku prostego tekstu oraz danych binarnych, a `unicode` w przypadku bardziej zaawansowanych form tekstu. W Pythonie 3.0 użyjemy `str` dla dowolnego rodzaju tekstu (prostego i Unicode), a `bytes` lub `bytearray` dla danych binarnych. W praktyce wybór jest często determinowany przez wykorzystywane przez nas narzędzia — zwłaszcza w przypadku narzędzi do przetwarzania plików, czyli tematu kolejnego podrozdziału.

Pliki binarne i tekstowe

Wejście-wyjście plików zostało w Pythonie 3.0 przebudowane, by odzwierciedlać rozróżnienie między typami `str` i `bytes` oraz automatycznie obsługiwać kodowanie tekstu Unicode. Python tworzy teraz jasne, niezależne od platformy rozróżnienie między plikami tekstowymi a binarnymi:

Pliki tekstowe

Kiedy plik otwierany jest w *trybie tekstowym*, wczytanie jego danych automatycznie dekoduje jego zawartość (zgodnie z ustawieniem domyślnym platformy lub podaną nazwą kodowania) i zwraca ją w postaci obiektu `str`. Zapisanie do pliku odbywa się poprzez obiekt `str` i automatyczne kodowanie przed przeniesieniem do pliku. Pliki w trybie tekstowym obsługują również uniwersalne przekładanie końca wiersza oraz dodatkowe argumenty określające kodowanie. W zależności od nazwy kodowania pliki tekstowe mogą również automatycznie przetwarzać sekwencję znacznika kolejności bajtów (ang. *byte order mark*, *BOM*) na początku pliku (więcej na ten temat za moment).

Pliki binarne

Kiedy plik otwierany jest w *trybie binarnym* za pomocą dodania `b` (malej litery) do argumentu łańcucha znaków trybu we wbudowanym wywołaniu `open`, wczytanie jego danych nie powoduje ich zdekodowania, a po prostu zwraca zawartość pliku w surowej i nienazmienionej postaci, jako obiekt `bytes`. Zapisywanie do pliku odbywa się poprzez obiekt `bytes` i w podobny sposób zawartość przenoszona jest do pliku bez zmian. Pliki w trybie bajtowym przyjmują również obiekt `bytearray` w przypadku zawartości, która ma być do nich zapisana.

Ponieważ Python rozróżnia typy `str` i `bytes`, musimy zdecydować, czy nasze dane są z natury tekstowe, czy binarne, i użyć odpowiedniego obiektu w celu ich poprawnej reprezentacji w skrypcie. Wybór trybu otwarcia pliku decyduje o tym, jaki typ obiektu skrypt wykorzysta do reprezentowania swojej zawartości:

- Jeśli przetwarzamy pliki graficzne, spakowane dane utworzone za pomocą innych programów, których zawartość musimy poddać ekstrakcji, podobnie jak strumienie danych różnych urządzeń, istnieje spora szansa, że chcemy je potraktować, używając typu `bytes` oraz plików w *trybie binarnym*. Jeśli chcemy uaktualniać dane bez tworzenia ich kopii w pamięci, możemy także wybrać typ `bytearray`.
- Jeśli zamiast tego przetwarzamy coś, co z natury jest tekstowe, jak na przykład dane wyjściowe programu, kod HTML, tekst ze znakami międzynarodowymi, pliki CSV czy XML, najprawdopodobniej będziemy chcieli skorzystać z typu `str` i plików w *trybie tekstowym*.

Warto zauważyć, że argument *łańcucha znaków trybu* wbudowanej funkcji `open` (jej drugi argument) w Pythonie 3.0 stał się bardzo istotny — jego wartość nie tylko określa *tryb przetwarzania* pliku, ale także sugeruje *typ obiektu* Pythona. Dodając do łańcucha trybu `b`, wybieramy tryb binarny i otrzymamy (a także musimy dostarczyć) obiekt `bytes`, który będzie reprezentował zawartość pliku w czasie odczytywania i zapisywania. Bez znaku `b` plik przetwarzany jest w trybie tekstowym, a do reprezentowania jego zawartości w skrypcie posłuży nam obiekt `str`. Przykładowo tryby `rb`, `wb` oraz `rb+` wymuszają obiekt `bytes`, natomiast `r`, `w+` oraz `rt` (tryb domyślny) — obiekt `str`.

Pliki w trybie tekstowym mogą także obsługiwać sekwencję znacznika kolejności bajtów (*BOM*), która może pojawiać się na początku pliku w niektórych schematach kodowania. W kodowaniu UTF-16 oraz UTF-32 sekwencja *BOM* określa format *big-endian* lub *little-endian* (określający w przybliżeniu, który koniec łańcucha bitowego jest ważniejszy). Plik tekstowy UTF-8 może także zawierać sekwencję *BOM* w celu zadeklarowania, że jest zakodowany jako UTF-8, jednak nie ma gwarancji, że tak będzie. Przy zapisywaniu i odczytywaniu danych za pomocą tych schematów kodowania Python automatycznie pomija lub zapisuje *BOM*, jeśli wynika to z ogólnej nazwy kodowania lub gdy podamy bardziej szczegółową nazwę kodowania w celu wymuszenia

tego. Przykładowo sekwencja BOM zawsze jest przetwarzana dla „utf-16”, bardziej uszczegółowiona nazwa kodowania „utf-16-le” określa format UTF-16 *little-endian*, bardziej uszczegółowiona nazwa kodowania „utf-8-sig” zmusza Pythona do, odpowiednio, pominięcia i zapisania BOM dla danych wejściowych i wyjściowych dla tekstu UTF-8 (nie dzieje się tak w przypadku ogólnego nazwy „utf-8”).

Więcej informacji na temat sekwencji BOM oraz plików znajdzie się w podrozdziale „Obsługa BOM w Pythonie 3.0”. Najpierw jednak zastanówmy się nad konsekwencjami nowego modelu łańcuchów znaków Unicode Pythona.

Łańcuchy znaków Pythona 3.0 w akcji

Przyjrzyjmy się kilku przykładom demonstrującym wykorzystywanie nowego typu łańcuchów znaków z Pythona 3.0. Jedna uwaga: kod w niniejszym podrozdziale został wykonany i ma zastosowanie jedynie do wersji 3.0. Mimo to podstawowe działania na łańcuchach znaków są zasadniczo przenośne pomiędzy wersjami Pythona. Proste łańcuchy znaków ASCII reprezentowane za pomocą typu `str` działają tak samo w wersji 2.6 oraz 3.0 (i dokładnie tak samo, jak widzieliśmy to w rozdziale 7. niniejszej książki). Co więcej, choć w Pythonie 2.6 nie istnieje typ `bytes` (jest tam tylko ogólny typ `str`), zazwyczaj można wykonać w nim kod, który zakłada, że typ ten tam się znajduje. W wersji 2.6 wywołanie `bytes(X)` obecne jest jako synonim dla `str(X)`, a nowa forma literala `b'...'` traktowana jest tak samo jak normalny literał łańcucha znaków `'...'`. W niektórych wyizolowanych przypadkach nadal możemy natrafić na pewne różnice w zakresie wersji — wywołanie `bytes` z Pythona 2.6 nie pozwala na przykład na podanie drugiego argumentu (nazwy kodowania) wymaganego przez `bytes` z wersji 3.0.

Literały i podstawowe właściwości

Obiekty łańcuchów znaków Pythona 3.0 pojawiają się, kiedy wywołujemy funkcję wbudowaną, taką jak `str` lub `bytes`, przetwarzamy plik utworzony za pomocą wywołania `open` (więcej informacji w kolejnym podrozdziale) lub zapisujemy w kodzie skryptu składnię literatu. W tym ostatnim przypadku nowa postać literalu `b'xxx'` (i jej równoważnik `B'xxx'`) wykorzystywana jest do tworzenia obiektów typu `bytes` w Pythonie 3.0. Obiekty typu `bytearray` tworzy się, wywołując funkcję `bytearray` z różnymi argumentami.

Z formalnego punktu widzenia w Pythonie 3.0 wszystkie obecne formy literałów — `'xxx'`, `"xxx"` oraz bloki w potrójnych cudzysłowach lub apostrofach — generują obiekt `str`. Dodanie znaku `b` lub `B` przed dowolną z nich tworzy zamiast tego obiekty `bytes`. Nowy literał `b'...'` jest podobny w formie do surowego łańcucha znaków `r'...'` wykorzystywanego do powstrzymania znaków ucieczki z ukośnikami lewymi. Rozważmy poniższy, wykonany w Pythonie 3.0, kod:

```
C:\misc> c:\python30\python
>>> B = b'mielonka'          # Utworzenie obiektu typu bytes (8-bitowe bajty)
>>> S = 'jajka'              # Utworzenie obiektu typu str (znaki Unicode, 8-bitowe lub większe)

>>> type(B), type(S)
(<class 'bytes'>, <class 'str'>)
```

```
>>> B = b'melonka' # Wyświetla jako łańcuchy znaków, a tak naprawdę są to sekwencje liczb całkowitych
>>> S = 'ajka'
```

Obiekt bytes jest tak naprawdę sekwencją krótkich liczb całkowitych, choć swoją zawartość wyświetla jako znaki zawsze, gdy jest to możliwe:

```
>>> B[0], S[0]      # Indeksowanie zwraca typ int dla bytes, typ str dla str
(109, 'j')
>>> B[1:], S[1:]    # Wycinek tworzy inny obiekt bytes lub str
(b'melonka', 'ajka')
>>> list(B), list(S) # bytes to tak naprawdę liczby całkowite
([109, 105, 101, 108, 111, 110, 107, 97], ['j', 'a', 'j', 'k', 'a'])
```

Obiekt bytes jest niezmienny, podobnie jak str (choć opisany niżej bytearray już nie jest). Nie można przypisać obiektu str, bytes czy liczby całkowitej do wartości przesunięcia dla obiektu bytes. Przedrostek obiektu bytes działa także dla dowolnej postaci literała łańcucha znaków:

```
>>> B[0] = 'x'      # Oba są niezmienne
TypeError: 'bytes' object does not support item assignment
>>> S[0] = 'x'
TypeError: 'str' object does not support item assignment
>>> B = B'''      # Przedrostek bytes działa dla apostrofów, cudzysłówów i podwójnych apostrofów oraz cudzysłówów
...
... xxxx
... yyyy
...
...
>>> B
b'\nxxxx\nyyyy\n'
```

Jak wspomniano wcześniej, w Pythonie 2.6 literał b'xxx' jest obecny z uwagi na zgodność z wersją 3.0, jednak jest tym samym co 'xxx' i tworzy obiekt str; bytes jest z kolei po prostu synonimem str. Jak widzieliśmy, w wersji 3.0 obie formy odnoszą się do odrębnych typów łańcuchów znaków. Warto także zauważyć, że formy literala łańcucha znaków Unicode u'xxx' oraz U'xxx' z wersji 2.6 w Pythonie 3.0 zniknęły; zamiast tego należy użyć 'xxx', ponieważ wszystkie łańcuchy znaków są Unicode, nawet jeśli zawierają tylko znaki ASCII (więcej informacji na temat zapisywania tekstu Unicode spoza ASCII znajduje się w podrozdziale „Kod tekstu spoza zakresu ASCII”).

Konwersje

Choć Python 2.X pozwalał na swobodne mieszanie obiektów typu str i unicode (jeśli łańcuchy znaków zawierały jedynie 7-bitowy tekst ASCII), wersja 3.0 zachowuje o wiele bardziej ścisły podział. Obiekty typów str i bytes nigdy automatycznie nie mieszają się ze sobą w wyrażeniach i nigdy nie są wzajemnie konwertowane przy przekazywaniu do funkcji. Funkcja oczekująca, że argument będzie obiektem str, nie przyjmie obiektu bytes — i odwrotnie.

Z tego powodu Python 3.0 wymaga zdecydowania się na jeden z typów lub wykonania ręcznego i jawnego przekształcenia:

- Funkcje `str.encode()` oraz `bytes(S, kodowanie)` przekładają łańcuch znaków na jego postać bajtową i z obiektu `str` tworzą w rezultacie obiekt `bytes`.
- Funkcje `bytes.decode()` oraz `str(B, kodowanie)` przekładają łańcuch znaków na jego postać bajtową i z obiektu `bytes` tworzą w rezultacie obiekt `str`.

Powyższe metody `encode` i `decode` (a także opisane w następnym podrozdziale obiekty plików) wykorzystują albo domyślne kodowanie platformy, albo nazwę kodowania przekazaną w sposób jawnego. Przykładowo w Pythonie 3.0:

```
>>> S = 'jajka'  
>>> S.encode()                                     # Ze str na bytes: kodowanie tekstu na bajty  
b'jajka'  
  
>>> bytes(S, encoding='utf-8')                   # Ze str na bytes, alternatywa  
b'jajka'  
  
>>> B = b'mielonka'  
>>> B.decode()                                     # Z bytes do str: zdekodowanie bajtów na tekst  
'mielonka'  
  
>>> str(B, encoding='utf-8')                      # Z bytes do str, alternatywa  
'mielonka'
```

Dwie uwagi. Po pierwsze, kodowanie domyślne platformy dostępne jest w module `sys`, jednak argument kodowania dla `bytes` nie jest opcjonalny, nawet jeśli jest tak w przypadku `str.encode` (oraz `bytes.decode`).

Po drugie, choć wywołania `str` nie wymagają podania argumentu kodowania, tak jak to jest w przypadku `bytes`, opuszczenie go w wywołaniach `str` nie oznacza wartości domyślnej — wywołanie `str` bez kodowania zwraca łańcuch wyświetlania obiektu bajtowego, a nie postać po konwersji do `str` (i nie jest to zazwyczaj efekt, którego oczekujemy!). Zakładając, że zmienne `B` oraz `S` są tym samym co w poprzednim kodzie:

```
>>> import sys  
>>> sys.platform                                    # Wykorzystywana platforma  
'win32'  
>>> sys.getdefaultencoding()                      # Domyślne kodowanie dla obiektów str  
'utf-8'  
  
>>> bytes(S)  
TypeError: string argument without an encoding  
  
>>> str(B)                                         # Obiekt str bez kodowania  
"b'mielonka"                                       # Łąńcuch znaków wyświetlania, a nie konwersja!  
>>> len(str(B))  
11  
>>> len(str(B, encoding='utf-8'))                 # Użycie kodowania w celu konwersji na str  
8
```

Kod łańcuchów znaków Unicode

Kodowanie i dekodowanie mają większe znaczenie, kiedy zaczniemy zajmować się prawdziwym tekstem Unicode spoza zakresu ASCII. By umieścić w kodzie łańcucha znaków znaki Unicode, z których części pewnie nawet nie możemy wpisać na posiadanej klawiaturze, literał łańcuchów znaków Pythona obsługują zarówno szesnastkowe wartości bajtowe z sekwencjami ucieczki "`\xNN`", jak i sekwencje ucieczki Unicode w literałach łańcuchów znaków "`\uNNNN`"

oraz "\UNNNNNNNN". W przypadku sekwencji ucieczki Unicode pierwsza postać podaje cztery cyfry szesnastkowe w celu zakodowania 2-bajtowego (16-bitowego) kodu znaku, natomiast druga forma podaje osiem cyfr szesnastkowych dla kodu 4-bajtowego (32-bitowego).

Kod tekstu z zakresu ASCII

Przyjrzyjmy się kilku przykładom demonstrującym podstawy kodowania tekstu. Jak widzimy, tekst ASCII jest prostym typem Unicode, przechowanym jako sekwencja wartości bajtowych reprezentujących znaki:

```
C:\misc> c:\python30\python

>>> ord('X')                                # 'X' ma wartość binarną 88 w kodowaniu domyślnym
88
>>> chr(88)                                 # 88 oznacza znak 'X'
'X'

>>> S = 'XYZ'                               # Łąćuch znaków Unicode dla tekstu ASCII
>>> S
'XYZ'
>>> len(S)                                  # Długość 3 znaków
3
>>> [ord(c) for c in S]                     # 3 bajty z liczbowymi wartościami porządkowymi
[88, 89, 90]
```

Taki normalny 7-bitowy tekst ASCII reprezentowany jest za pomocą jednego znaku na bajt w każdym ze schematów Unicode opisanych wcześniej w rozdziale:

```
>>> S.encode('ascii')                      # Wartości 0..127, każde w 1 bajcie (7 bitach)
b'XYZ'
>>> S.encode('latin-1')                    # Wartości 0..255, każde w 1 bajcie (8 bitach)
b'XYZ'
>>> S.encode('utf-8')                      # Wartości 0..127, każde w 1 bajcie, 128..2047 w 2, inne w 3 lub 4
b'XYZ'
```

Tak naprawdę obiekty zwieracane dzięki zakodowaniu tekstu ASCII w ten sposób są sekwencjami krótkich liczb całkowitych, które — tak się składa — gdy jest to możliwe, wyświetlane są jako znaki ASCII:

```
>>> S.encode('latin-1')[0]
88
>>> list(S.encode('latin-1'))
[88, 89, 90]
```

Kod tekstu spoza zakresu ASCII

By zapisać w kodzie znaki spoza zakresu ASCII, możemy w łańcuchach znaków użyć sekwencji ucieczki, szesnastkowych bądź Unicode. Szesnastkowe sekwencje ucieczki ograniczone są do wartości jednobajtowych, natomiast sekwencje ucieczki Unicode mogą wymieniać znaki o wartościach dwu- i czterobajtowych. Wartości szesnastkowe 0xCD i 0xE8 są na przykład kodami dwóch specjalnych znaków akcentowanych spoza 7-bitowego zakresu ASCII, jednak możemy je osadzać w obiektach str Pythona 3.0, ponieważ obsługuje on obecnie standard Unicode:¹

¹ Uwaga! Przykłady ze znakiem è oraz innymi znakami spoza ASCII mogą nie działać w wierszu poleceń systemu Windows, zwracając błąd `UnicodeEncodeError`, natomiast bez problemu powinno się je dać uruchomić w konsoli IDLE — *przyp. tłum.*

```

>>> chr(0xc4)                                # 0xC4, 0xE8: znaki spoza zakresu ASCII
'Ã'
>>> chr(0xe8)
'è'

>>> S = '\xc4\xe8'                           # Jednobajtowe 8-bitowe szesnastkowe sekwencje ucieczki
>>> S
'Ãè'

>>> S = '\u00c4\u00e8'                         # 16-bitowe sekwencje ucieczki Unicode
>>> S
'Ãè'

>>> len(S)                                    # 2 znaki d³ugo¶ci (nie liczba bajtów!)
2

```

Kodowanie i dekodowanie tekstu spoza zakresu ASCII

Jeśli teraz spróbujemy *zakodować* łańcuch znaków spoza ASCII na łańcuch bajtowy wykorzystujący ASCII, otrzymamy błąd. Kodowanie jako Latin-1 działa i alokuje po jednym bajcie na znak. Kodowanie jako UTF-8 działa i alokuje zamiast tego po dwa bajty na znak. Jeśli zapiszemy ten łańcuch znaków do pliku, pokazany poniżej obiekt bytes jest tym, co zostanie przechowane w pliku dla podanych typów kodowania:

```

>>> S = '\u00c4\u00e8'
>>> S
'Ãè'
>>> len(S)
2

>>> S.encode('ascii')
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not
in range(128)

>>> S.encode('latin-1')                      # Jeden bajt na znak
b'\xc4\xe8'

>>> S.encode('utf-8')                        # Dwa bajty na znak
b'\xc3\x84\xc3\xaa'

>>> len(S.encode('latin-1'))                 # 2 bajty w latin-1, 4 w utf-8
2
>>> len(S.encode('utf-8'))
4

```

Warto zauważyć, że możemy także postąpić odwrotnie, wczytując surowe bajty z pliku i *dekodując* je z powrotem na łańcuchach znaków Unicode. Jak jednak zobaczymy później, tryb kodowania podawany do wywołań open powoduje automatyczne dekodowanie danych wejściowych (i uniknięcie problemów, które mogą się pojawić w związku z wczytaniem części sekwencji znaków przy wczytywaniu bloków bajtów):

```

>>> B = b'\xc4\xe8'
>>> B
b'\xc4\xe8'
>>> len(B)                                    # 2 surowe bajty, 2 znaki
2
>>> B.decode('latin-1')                      # Zdekodowanie do tekstu latin-1
'Ãè'

>>> B = b'\xc3\x84\xc3\xaa'
>>> len(B)                                    # 4 surowe bajty
4

```

```

4
>>> B.decode('utf-8')
'Àè'
>>> len(B.decode('utf-8'))           # 2 znaki Unicode
2

```

Inne techniki kodowania łańcuchów Unicode

Niektóre schematy kodowania wykorzystują jeszcze większe sekwencje bajtów do reprezentowania znaków. Jeśli jest to potrzebne, możemy podać 16- i 32-bitowe wartości Unicode dla znaków w łańcuchu — należy użyć zapisu "\u...." z czterema cyframi dla tego pierwszego i "\U...." z ośmioma cyframi szesnastkowymi dla tego drugiego:

```

>>> S = 'A\u00c4B\u000000e8C'
>>> S                               # A, B, C i dwa znaki spoza ASCII
'AÄBèC'
>>> len(S)                         # 5 znaków długości
5

>>> S.encode('latin-1')
b'A\xc4B\xe8C'
>>> len(S.encode('latin-1'))       # 5 bajtów w latin-1
5

>>> S.encode('utf-8')
b'A\xc3\x84B\xc3\x8c'
>>> len(S.encode('utf-8'))         # 7 bajtów w utf-8
7

```

Co ciekawe, inne typy kodowania mogą wykorzystywać bardzo odmienne formaty bajtowe. Przykładowo kodowanie cp500 EBCDIC nie koduje nawet ASCII w ten sam sposób jak pozostałe (ponieważ Python koduje i dekoduje za nas, musimy się tym przejmować jedynie wtedy, gdy podajemy nazwy kodowania):

```

>>> S
'AÄBèC'
>>> S.encode('cp500')             # Dwa inne kodowania zachodnioeuropejskie
b'\xc1\x84\x20\x8c'
>>> S.encode('cp850')             # 5 bajtów każdy
b'A\x8eB\x8aC'

>>> S = 'mielonka'                # Tekst ASCII jest taki sam w większości typów kodowania
>>> S.encode('latin-1')
b'mielonka'
>>> S.encode('utf-8')
b'mielonka'
>>> S.encode('cp500')              # Ale nie w cp500: IBM EBCDIC!
b'\x94\x89\x85\x93\x96\x95\x92\x81'
>>> S.encode('cp850')
b'mielonka'

```

Łańcuchy znaków Unicode można także budować stopniowo za pomocą `chr` zamiast sekwencji ucieczki (Unicode czy szesnastkowych), jednak w przypadku większych łańcuchów znaków może się to stać dość żmudne:

```

>>> S = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'
>>> S
'AÄBèC'

```

Dwie uwagi. Po pierwsze, Python 3.0 pozwala na kodowanie znaków specjalnych za pomocą sekwencji ucieczki szesnastkowych i Unicode w łańcuchach znaków str, jednak w łańcuchach bytes tylko z użyciem wersji szesnastkowej. Sekwencje ucieczki Unicode są po cichu uznawane za dosłowne literały obiektów bytes, a nie sekwencje ucieczki. Tak naprawdę w celu poprawnego wyświetlenia znaków spoza zakresu ASCII obiekty bytes trzeba zdekodować do łańcuchów str:

```
>>> S = 'A\xC4B\xE8C'                                # str rozpoznaje sekwencje ucieczki Unicode i szesnastkowe
>>> S
'AÄBëC'

>>> S = 'A\u00C4B\u000000E8C'
>>> S
'AÄBëC'

>>> B = b'A\xC4B\xE8C'                                # bytes rozpoznaje szesnastkowe sekwencje ucieczki, Unicode nie
>>> B
b'A\xc4B\xe8C'

>>> B = b'A\u00C4B\u000000E8C'                        # Sekwencje ucieczki potraktowane dosłownie!
>>> B
b'A\\u00C4B\\U000000E8C'

>>> B = b'A\xC4B\xE8C'                                # W przypadku bytes trzeba użyć szesnastkowych sekwencji ucieczki
>>> B
b'A\xc4B\xe8C'
>>> print(B)
b'A\xc4B\xe8C'
>>> B.decode('latin-1')                            # Wyświetla znaki spoza ASCII jako szesnastkowe
'AÄBëC'                                              # Dekoduje jako latin-1, by zinterpretować jako tekst
```

Po drugie, literały łańcuchów bytes wymagają, aby znaki były albo z zakresu ASCII, albo — jeśli ich wartość jest większa od 127 — by były opatrzone znakami ucieczki. Łańcuchy str z kolei zezwalają na stosowanie w literałach dowolnych znaków ze źródłowego zbioru znaków (którym, jak omówimy to później, domyślnie jest UTF-8, o ile w pliku źródłowym nie ma deklaracji kodowania):

```
>>> S = 'AÄBëC'                                    # Znaki z UTF-8, jeśli nie ma deklaracji kodowania
>>> S
'AÄBëC'

>>> B = b'AÄBëC'                                 # SyntaxError: bytes can only contain ASCII literal characters.

>>> B = b'A\xC4B\xE8C'                          # Znaki muszą być ASCII lub z sekwencjami ucieczki
>>> B
b'A\xc4B\xe8C'
>>> B.decode('latin-1')
'AÄBëC'

>>> S.encode()                                     # Kod źródłowy domyślnie zakodowany jako UTF-8
b'A\xc3\x84B\xc3\xa8C'                           # Wykorzystuje ustawienie domyślne systemu do zdekodowania,
                                                # o ile nie przekazano kodowania

>>> S.encode('utf-8')                            # Surowe bajty nie odpowiadają utf-8
b'A\xc3\x84B\xc3\xa8C'

>>> B.decode()                                    UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: ...
                                                # Są to znaki, dla których nie istnieje kodowanie utf-8
```

Konwersja kodowania

Dotychczas kodowaliśmy i dekodowaliśmy łańcuchy znaków w celu zbadania ich struktury. Mówiąc bardziej ogólnie, możemy zawsze przekształcić łańcuch znaków na kodowanie inne od ustawienia domyślnego zbioru znaków źródłowych, jednak musimy w jawnym sposobie podać nazwę kodowania:

```
>>> S = 'AÄBëC'  
>>> S  
'AÄBëC'  
>>> S.encode() # Domyślne kodowanie utf-8  
b'A\xc3\x84B\xc3\xa8C'  
  
>>> T = S.encode('cp500') # Konwersja na EBCDIC  
>>> T  
b'\xc1c\x2T\xc3'  
  
>>> U = T.decode('cp500') # Konwersja z powrotem na Unicode  
>>> U  
'AÄBëC'  
  
>>> U.encode() # Znowu domyślne kodowanie utf-8  
b'A\xc3\x84B\xc3\xa8C'
```

Należy pamiętać, że specjalne sekwencje ucieczki Unicode i szesnastkowe są niezbędne jedynie wówczas, kiedy ręcznie piszemy kod łańcuchów znaków Unicode spoza ASCII. W praktyce często zamiast tego taki tekst ładujemy z plików. Jak zobaczymy później w niniejszym rozdziale, obiekt pliku z Pythona 3.0 (tworzony za pomocą wbudowanej funkcji open) automatycznie dekoduje tekstowe łańcuchy znaków w miarę ich wczytywania i koduje je w czasie zapisywania. Z tego powodu skrypty mogą często obsługiwać łańcuchy znaków w sposób ogólny, bez konieczności bezpośredniego zapisywania w kodzie znaków specjalnych.

W dalszej części rozdziału zobaczymy również, że można także przekształcać tekst z jednego kodowania na inne przy przenoszeniu łańcuchów znaków z plików i do nich, wykorzystując do tego technikę bardzo podobną do tej z ostatniego przykładu. Choć nadal będziemy musieli w jawnym sposobie udostępnić nazwy kodowania przy otwieraniu pliku, interfejs plików wykonuje większość zadań związanych z konwersją w sposób automatyczny.

Łańcuchy znaków Unicode w Pythonie 2.6

Skoro już zaprezentowałem podstawy łańcuchów znaków Unicode z Pythona 3.0, powinienem wyjaśnić, że prawie to samo można uzyskać w wersji 2.6, choć za pomocą innych narzędzi. Typ `unicode` dostępny jest w Pythonie 2.6, jednak jest typem danych odrębnym od `str` i pozwala na swobodne mieszanie zwykłych łańcuchów znaków i łańcuchów Unicode, kiedy są one ze sobą zgodne. Tak naprawdę możemy udawać, że typ `str` z wersji 2.6 jest typem `bytes` z wersji 3.0, jeśli chodzi o dekodowanie surowych bajtów na łańcuchy Unicode, o ile są one w poprawnej formie. Poniżej znajduje się przykład działania Pythona 2.6. Znaki `unicode` wyświetlane są w wersji 2.6 jako szesnastkowe, o ile ich w jawnym sposobie nie wyświetlimy, natomiast wyświetlanie znaków spoza ASCII wygląda różnie w różnych powłokach (większa część przykładów z tego podrozdziału wykonana została w IDLE):

```
C:\misc> c:\python26\python  
>>> import sys  
>>> sys.version  
'2.6 (r26:66721, Oct 2 2008, 11:35:03) [MSC v.1500 32 bit (Intel)]'
```

```

>>> S = 'A\xC4B\xE8C'          # Łąćuch 8-bitowych bajtów
>>> print S                  # Niektóre z nich są spoza ASCII
AÄBëC

>>> S.decode('latin-1')       # Zdekodowanie bajta do Unicode w latin-1
u'A\xc4B\xe8C'

>>> S.decode('utf-8')         # Nie jest sformatowane jak utf-8
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 1-2: invalid data

>>> S.decode('ascii')         # Poza zakresem ASCII
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1: ordinal not in
range(128)

```

By przechować dowolnie zakodowany tekst Unicode, należy utworzyć obiekt `unicode` za pomocą literała w postaci `u'xxx'` (literał ten nie jest już dostępny w wersji 3.0, ponieważ wszystkie łańcuchy znaków obsługują w tej wersji Unicode):

```

>>> U = u'A\xC4B\xE8C'      # Utworzenie łańcucha Unicode, szesnastkowe sekwencje ucieczki
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBëC

```

Po utworzeniu możemy przekształcić tekst Unicode na różne typy kodowania surowych bajtów, podobnie do kodowania obiektów `str` na obiekty `bytes` w Pythonie 3.0:

```

>>> U.encode('latin-1')       # Kodowanie latin-1: 8-bitowe bajty
'A\xc4B\xe8C'
>>> U.encode('utf-8')        # Kodowanie utf-8: wielobajtowe
'A\xc3\x84B\xc3\xa8C'

```

Znaki spoza zakresu ASCII można zapisywać w Pythonie 2.6 w kodzie literałów łańcuchów znaków za pomocą sekwencji ucieczki szesnastkowych bądź Unicode, tak samo jak w wersji 3.0. Tak samo jednak jak w przypadku typu `bytes` w 3.0, w wersji 2.6 sekwencje ucieczki "`\u...`" oraz "`\U...`" rozpoznawane są jedynie w przypadku łańcuchów znaków `unicode`, a nie 8-bitowych łańcuchów `str`:

```

C:\misc> c:\python26\python
>>> U = u'A\xC4B\xE8C'      # Szesnastkowe sekwencje ucieczki dla znaków spoza ASCII
>>> U
u'A\xc4B\xe8C'
>>> print U
AÄBëC

>>> U = u'A\u00C4B\u000000E8C'  # Sekwencje ucieczki Unicode dla znaków spoza ASCII
>>> U
u'A\xc4B\xe8C'
# u'' = 16 bitów, U'' = 32 bity
>>> print U
AÄBëC

>>> S = 'A\xC4B\xE8C'        # Szesnastkowe sekwencje ucieczki działają
>>> S
'A\xc4B\xe8C'
>>> print S                  # Niektóre są dziwnie wyświetlane, jeśli się ich nie zdekoduje
A-BFC
>>> print S.decode('latin-1')
AÄBëC

>>> S = 'A\u00C4B\u000000E8C'  # Nie sekwencje ucieczki Unicode — traktowane są dosłownie!
>>> S
'A\\u00C4B\\U000000E8C'

```

```
>>> print S
A\u00C4B\U000000E8C
>>> len(S)
19
```

Tak jak typy str i bytes z Pythona 3.0, typy unicode i str z wersji 2.6 współdzielą prawie identyczny zbiór operacji, dlatego o ile nie musimy przekształcać ich na inne kodowanie, często możemy traktować obiekty unicode tak, jakby były obiektami str. Jedną z głównych różnic między 2.6 a 3.0 jest jednak to, że unicode i obiekty str poza Unicode można ze sobą swobodnie *mieszać* w wyrażeniach, a dopóki obiekt str jest zgodny z kodowaniem unicode, Python automatycznie przekształci go na unicode. W Pythonie 3.0 obiekty str i bytes nigdy nie mieszają się automatycznie i wymagają ręcznej konwersji.

```
>>> u'ab' + 'cd'                                # Mogą się w 2.6 mieszać, jeśli są zgodne
u'abcd'                                         # 'ab' + 'cd' nie jest dozwolone w wersji 3.0
```

Tak naprawdę różnica w typach jest często dla kodu napisanego w wersji 2.6 trywialna. Tak jak normalne łańcuchy znaków, łańcuchy znaków Unicode można poddawać konkatenacji, indeksować, tworzyć z nich wycinki czy dopasowywać za pomocą modułu re; nie można ich także modyfikować w miejscu. Jeśli kiedykolwiek będziemy musieli dokonać jawniej konwersji między tymi dwoma typami, możemy wykorzystać wbudowane funkcje str i unicode:

```
>>> str(u'mielonka')                           # Z Unicode na normalny łańcuch znaków
'mielonka'
>>> unicode('mielonka')                         # Z normalnego łańcucha znaków na Unicode
u'mielonka'
```

Tak liberalne podejście do mieszania typów łańcuchów znaków w Pythonie 2.6 działa jednak tylko wtedy, gdy łańcuch znaków jest zgodny z typem kodowania obiektu unicode:

```
>>> S = 'A\xC4B\xE8C'                           # Nie można mieszać, jeśli są niezgodne
>>> U = u'A\xC4B\xE8C'
>>> S + U
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc4 in position 1: ordinal not in
range(128)

>>> S.decode('latin-1') + U                      # Nadal wymagana ręczna konwersja
u'A\xc4B\xe8CA\xc4B\xe8C'

>>> print S.decode('latin-1') + U
AÄBéCAÄBéC
```

Wreszcie, jak zobaczymy bardziej szczegółowo później w niniejszym rozdziale, wywołanie open z Pythona 2.6 obsługuje jedynie pliki bajtów 8-bitowych, zwracając ich zawartość jako łańcuchy znaków str. To od nas zależy interpretacja zawartości jako tekstu lub danych binarnych i ewentualne zdekodowanie. By wczytać i zapisać pliki Unicode, a także automatycznie zakodować lub zdekodować ich zawartość, należy użyć wywołania codecs.open z Pythona 2.6, opisanego w dokumentacji biblioteki tej wersji. Wywołanie to udostępnia prawie tę samą funkcjonalność co open z wersji 3.0 i wykorzystuje obiekty unicode z 2.6 do reprezentowania zawartości pliku. Wczytanie pliku przekłada zakodowane bajty na zdekodowane znaki Unicode, a zapisywanie przekłada łańcuchy znaków na pożąданie kodowanie określone przy otwarciu pliku.

Deklaracje typu kodowania znaków pliku źródłowego

Sekwencje ucieczki Unicode są w porządku w przypadku okazyjnego zastosowania znaku Unicode w literale łańcucha znaków, jednak ich użycie może stać się żmudne, jeśli często musimy osadzać tekst spoza zakresu ASCII w łańcuchach znaków. W przypadku łańcuchów znaków zapisywanych w kodzie plików skryptów Python domyślnie wykorzystuje kodowanie UTF-8, jednak pozwala je zmienić tak, by obsługiwać dowolny typ kodowania, umieszczając komentarz z nazwami pożądanego kodowania. Komentarz ten musi mieć następującą postać i musi się pojawiać w pierwszym lub drugim wierszu skryptu zarówno w Pythonie 2.6, jak i 3.0:

```
# -*- coding: latin-1 -*-
```

Kiedy komentarz w tej formie jest obecny w pliku, Python rozpozna łańcuchy znaków reprezentowane w podanym kodowaniu. Oznacza to, że możemy edytować plik skryptu w edytorze tekstu przyjmującym i poprawnie wyświetlanym znaki akcentowane i inne znaki spoza zakresu ASCII, a Python poprawnie je zdekoduje w literałach łańcuchów znaków. Przykładowo warto zwrócić uwagę, jak komentarz umieszczony na górze poniższego pliku *text.py* pozwala na osadzanie znaków Latin-1 w łańcuchach znaków:

```
# -*- coding: latin-1 -*-

# Każda z poniższych form literalu łańcucha znaków działa w latin-1.
# Zmiana kodowania wyżej na ascii albo utf-8 nie powiedzie się, ponieważ znaki 0xc4 i 0xe8 z myStr1 nie są w nich
# poprawne.

myStr1 = 'aÄBèC'

myStr2 = 'A\u00c4B\u00e8C'

myStr3 = 'A' + chr(0xC4) + 'B' + chr(0xE8) + 'C'

import sys
print('Kodowanie systemowe:', sys.getdefaultencoding())

for aStr in myStr1, myStr2, myStr3:
    print('{0}, strlen={1}'.format(aStr, len(aStr)), end='')

bytes1 = aStr.encode()          # Zgodnie z domyślnym utf-8: 2 bajty dla znaków spoza ASCII
bytes2 = aStr.encode('latin-1')  # Jeden bajt na znak
#bytes3 = aStr.encode('ascii')   # ASCII nie działa: poza zakresem 0..127

print('byteslen1={0}, byteslen2={1}'.format(len(bytes1), len(bytes2)))
```

Po wykonaniu powyższy skrypt daje następujące wyniki:

```
C:\misc> c:\python30\python text.py
Kodowanie domyślne: utf-8
aÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
AÄBèC, strlen=5, byteslen1=7, byteslen2=5
```

Ponieważ większość programistów będzie polegała na standardowym kodowaniu UTF-8, osoby zainteresowane szczegółowymi informacjami dotyczącymi tej opcji, a także innymi zaawansowanymi zagadnieniami związanymi z Unicode, takimi jak właściwości i sekwencje ucieczki znaków w łańcuchach, odsyłam do dokumentacji biblioteki standardowej Pythona.

Wykorzystywanie obiektów bytes z Pythona 3.0

W rozdziale 7. omówiliśmy szeroki zakres operacji dostępnych w ogólnym typie łańcucha znaków Pythona 3.0 str. Podstawowy typ danych łańcucha znaków działa w wersjach 2.6 oraz 3.0 identycznie, dlatego nie będziemy wracać do tego zagadnienia. Zamiast tego zagłębimy się nieco w zbiór operacji udostępnianych przez nowy typ bytes z Pythona 3.0.

Jak wspomniano wcześniej, obiekt bytes z wersji 3.0 jest sekwencją niewielkich liczb całkowitych, z których każda mieści się w przedziale od 0 do 255; zaprojektowano go tak, by po wyświetleniu pokazywał znaki ASCII. Obsługuje on operacje na sekwencjach i większość metod dostępnych dla obiektów str (a także obecnych w typie str z Pythona 2.X). Obiekt bytes nie obsługuje jednak metody format ani wyrażenia formatującego ze znakiem %. Nie można także mieszać i dopasowywać obiektów bytes i str bez jawnego przekształcenia ich. Zazwyczaj obiekty typu str wykorzystuje się w plikach tekstowych z przeznaczeniem na *dane tekstowe*, natomiast obiekty typu bytes służą w plikach binarnych do reprezentowania *danych binarnych*.

Wywołania metod

Jeśli naprawdę chcemy zobaczyć, jakie atrybuty, których nie ma typ bytes, znajdują się w typie str, zawsze możemy sprawdzić wyniki wywołania funkcji wbudowanej dir. Jej dane wyjściowe powiedzą nam także coś o obsługiwanych operatorach wyrażeń (przykładowo __mod__ oraz __rmod__ implementują operator %).

```
C:\misc> c:\python30\python

# Atrybuty unikalne dla typu str

>>> set(dir('abc')) - set(dir(b'abc'))
{'isprintable', 'format', '__mod__', 'encode', 'isidentifier',
 '_formatter_field_name_split', 'isnumeric', '__rmod__', 'isdecimal',
 '_formatter_parser', 'maketrans'}

# Atrybuty unikalne dla typu bytes

>>> set(dir(b'abc')) - set(dir('abc'))
{'decode', 'fromhex'}
```

Jak widać, typy str i bytes mają prawie identyczną funkcjonalność. Ich unikalne atrybuty to zazwyczaj metody, które nie mają zastosowania do tego drugiego typu. Przykładowo metoda decode przekłada surowe bajty na ich reprezentację str, natomiast encode tłumaczy łańcuch znaków na jego reprezentację w postaci surowych bajtów. Większość metod jest taka sama, choć metody bytes wymagają przekazania argumentów w postaci obiektów bytes (znów: typy łańcuchów znaków w Pythonie 3.0 nie mieszają się ze sobą). Warto także przypomnieć, że obiekty typu bytes są niezmienne, tak samo jak obiekty str, zarówno w wersji 2.6, jak i 3.0 (komunikaty o błędach zostały skrócone w imię oszczędności miejsca).

```
>>> B = b'mielonka'                      # Literal obiektu bytes b'...'
>>> B.find(b'ie')
1

>>> B.replace(b'ie', b'XY')                # Metody bytes oczekują argumentów bytes
b'mXYlonka'
```

```

>>> B.split(b'ie')
[b'm', b'lonka']

>>> B
b'mielonka'

>>> B[0] = 'x'
TypeError: 'bytes' object does not support item assignment

```

Jedyną istotną różnicą jest to, że *formatowanie łańcuchów znaków* działa w Pythonie 3.0 jedynie na obiektach str, natomiast na obiektach bytes już nie (więcej informacji na temat wyrażeń i metod formatowania łańcuchów znaków znajduje się w rozdziale 7.):

```

>>> b'%s' % 99
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'

>>> '%s' % 99
'99'

>>> b'{0}'.format(99)
AttributeError: 'bytes' object has no attribute 'format'

>>> '{0}'.format(99)
'99'

```

Operacje na sekwencjach

Oprócz wywołań metod wszystkie znane nam (i przez nas kochane) ogólne operacje na sekwencjach z łańcuchów znaków i list Pythona 2.X działają zgodnie z oczekiwaniemi na obiektach str oraz bytes w wersji 3.0. Obejmuje to między innymi indeksowanie, wycinki czy konkatenację. Warto zwrócić uwagę na to, jak w poniższym kodzie indeksowanie obiektu bytes zwraca liczbę całkowitą podającą wartość binarną bajta. Obiekt bytes jest tak naprawdę *sekwencją 8-bitowych liczb całkowitych*, jednak gdy jest wyświetlany w całości, dla wygody prezentuje się jako łańcuch znaków w kodzie ASCII. By sprawdzić wartość podanego bajta, należy użyć funkcji wbudowanej chr w celu przekształcenia jej z powrotem na znak — jak w poniższym kodzie:

```

>>> B = b'mielonka'                      # Sekwencja niewielkich liczb całkowitych
>>> B                                     # Wyświetlana jako znaki ASCII
b'mielonka'

>>> B[0]                                    # Indeksowanie zwraca liczbę całkowitą
109
>>> B[-1]
97

>>> chr(B[0])                             # Pokazanie znaku dla liczby całkowitej
'm'
>>> list(B)                                # Pokazanie wszystkich wartości liczb całkowitych obiektu bytes
[109, 105, 101, 108, 111, 110, 107, 97]

>>> B[1:], B[:-1]
(b'ielonka', b'mielonk')

>>> len(B)
8

>>> B + b'lmn'
b'mielonkalmn'
>>> B * 4
b'mielonkamielonkamielonkamielonka'

```

Inne sposoby tworzenia obiektów bytes

Dotychczas większość obiektów bytes tworzyliśmy za pomocą składni literała 'b...'. Można je jednak tworzyć także za pomocą wywołania konstruktora `bytes` z obiektem `str` i nazwą kodowania, wywołania konstruktora `bytes` z obiektem liczb całkowitych reprezentującym wartości bajtowe, na którym można wykonywać iterację, a także kodując obiekt `str` zgodnie z domyślnym (lub przekazanym) typem kodowania. Jak widzieliśmy, operacja kodowania pobiera obiekt `str` i zwraca wartość bajtową łańcucha znaków zgodnie ze specyfikacją tego kodowania. I odwrotnie, operacja dekodowania pobiera sekwencję surowych bajtów i koduje ją na jej reprezentację łańcucha znaków — serię znaków. Oba działania tworzą nowe obiekty łańcucha znaków.

```
>>> B = b'abc'  
>>> B  
b'abc'  
  
>>> B = bytes('abc', 'ascii')  
>>> B  
b'abc'  
  
>>> ord('a')  
97  
>>> B = bytes([97, 98, 99])  
>>> B  
b'abc'  
  
>>> B = 'mielonka'.encode()          # Lub bytes()  
>>> B  
b'mielonka'  
>>>  
>>> S = B.decode()                  # Lub str()  
>>> S  
'mielonka'
```

Z szerszej perspektywy dwie ostatnie z przedstawionych operacji są tak naprawdę narzędziami służącymi do *konwersji* pomiędzy obiektami `str` i `bytes`, czyli tematu wprowadzonego wcześniej i rozszerzonego w kolejnym podrozdziale.

Mieszanie typów łańcuchów znaków

W wywoaniu z `replace` w podrozdziale „Wywołania metod” musieliśmy przekazać dwa obiekty `bytes` — obiekty `str` by tam nie zadziałyły. Choć Python 2.X dokonuje automatycznej konwersji między typami `str` i `unicode`, gdy jest to możliwe (to jest gdy `str` jest 7-bitowym tekstem ASCII), Python 3.0 wymaga w pewnych kontekstach określonych typów łańcuchów znaków, a kiedy jest to potrzebne — ręcznych konwersji:

```
# Musi przekazywać oczekiwane typy do wywołań funkcji oraz metod  
  
>>> B = b'mielonka'  
  
>>> B.replace('ie', 'XY')  
TypeError: expected an object with the buffer interface  
  
>>> B.replace(b'ie', b'XY')  
b'mYlonka'  
  
>>> B = B'mielonka'
```

```

>>> B.replace(bytes('ie'), bytes('xy'))
TypeError: string argument without an encoding

>>> B.replace(bytes('ie', 'ascii'), bytes('xy', 'utf-8'))
b'myylonka'

# Musi dokonywać ręcznej konwersji w wyrażeniach z mieszanymi typami

>>> b'ab' + 'cd'
TypeError: can't concat bytes to str

>>> b'ab'.decode() + 'cd'                      # Z bytes na str
'abcd'

>>> b'ab' + 'cd'.encode()                      # Ze str na bytes
b'abcd'

>>> b'ab' + bytes('cd', 'ascii')               # Ze str na bytes
b'abcd'

```

Choć można samemu tworzyć obiekty bytes w celu reprezentowania spakowanych danych binarnych, są one także tworzone automatycznie po wczytaniu plików otwartych w trybie binarnym, co zobaczymy w szczegółach nieco później w niniejszym rozdziale. Najpierw jednak powinniśmy zapoznać się z bardzo bliskim i zmiennym krewnym obiektu bytes.

Wykorzystywanie obiektów bytearray z Pythona 3.0 (i 2.6)

Dotychczas koncentrowaliśmy się na typach str i bytes, ponieważ można je podciągnąć pod typy unicode i str z Pythona 2.X. Python 3.0 udostępnia jednak także trzeci typ łańcucha znaków — bytearray, czyli zmienna sekwencja liczb całkowitych w przedziale od 0 do 255, która jest zasadniczo zmiennym wariantem typu bytes. Tym samym obsługuje te same metody łańcuchów znaków i działania na sekwencjach co bytes, a także wiele z działań modyfikacji w miejscu obsługiwanych przez listy. Typ bytearray jest dostępny również w Pythonie 2.6 jako przeniesienie z wersji 3.0, jednak nie wymusza tam ścisłego rozróżnienia między danymi tekstowymi a binarnymi z Pythona 3.0.

Zajmijmy się teraz szybkim omówieniem tego typu. Obiekty bytearray można tworzyć, wywołując funkcję wbudowaną bytearray. W Pythonie do inicjalizacji można wykorzystać dowolny łańcuch znaków:

```

# Utworzenie w Pythonie 2.6: zmienna sekwencja małych (0..255) liczb całkowitych

>>> S = 'mielonka'
>>> C = bytearray(S)                         # Przeniesienie z wersji 3.0 do 2.6
>>> C                                         # b'..' == '..' w wersji 2.6 (str)
bytearray(b'mielonka')

```

W Pythonie 3.0 wymagana jest nazwa kodowania lub łańcuch bajtowy, ponieważ łańcuchy tekstowe i binarne nie mieszają się ze sobą, choć łańcuchy bajtowe mogą odzwierciedlać tekst zakodowany w Unicode:

```

# Utworzenie w Pythonie 2.6: tekst i dane binarne nie mieszają się ze sobą

>>> S = 'mielonka'
>>> C = bytearray(S)

```

```
TypeError: string argument without an encoding

>>> C = bytearray(S, 'utf-8')           # Typ odpowiedni dla zawartości w wersji 3.0
>>> C
bytearray(b'mielonka')

>>> B = b'mielonka'                   # b'..' != '..' w wersji 3.0 (bytes/str)
>>> C = bytearray(B)
>>> C
bytearray(b'mielonka')
```

Po utworzeniu obiekty bytearray stają się sekwencjami małych liczb całkowitych, tak jak typ bytes, i są zmienne, tak jak listy, choć dla operacji przypisania do indeksu wymagają liczb całkowitych, a nie łańcuchów znaków. Poniższy listing jest w całości kontynuacją powyższej sesji interaktywnej i wykonywany jest w Pythonie 3.0, o ile nie jest zaznaczone inaczej — komentarze dotyczące wersji 2.6 znajdują się w uwagach.

```
# Obiekt zmienny, jednak musi przypisywać liczby całkowite, a nie łańcuchy znaków

>>> C[0]
109

>>> C[0] = 'x'                      # Ta i poniższa operacja działają w wersji 2.6
TypeError: an integer is required

>>> C[0] = b'x'
TypeError: an integer is required

>>> C[0] = ord('x')
>>> C
bytearray(b'xielonka')

>>> C[1] = b'Y'[0]
>>> C
bytearray(b'xYelonka')
```

Przetwarzanie obiektów bytearray zapożycza zarówno z łańcuchów znaków, jak i list, ponieważ obiekty te są zmiennymi łańcuchami bajtów. Oprócz nazwanych metod metody `__iadd__` oraz `__setitem__` obiektu bytearray implementują, odpowiednio, konkatenację w miejscu `+=` oraz przypisanie do indeksu.

```
# Metody pokrywają się z działaniami dla typów str oraz bytes, jednak bytearray obsługuje także metody zmienne list

>>> set(dir(b'abc')) - set(dir(bytearray(b'abc')))
{'__getnewargs__'}

>>> set(dir(bytearray(b'abc'))) - set(dir(b'abc'))
{'__insert__', '__alloc__', 'reverse', 'extend', '__delitem__', 'pop', '__setitem__',
 '__iadd__', 'remove', 'append', '__imul__'}
```

Obiekt bytearray można zmodyfikować w miejscu za pomocą zarówno przypisania do indeksu, co właśnie widzieliśmy, jak i zaprezentowanych niżej metod podobnych do list. By zmodyfikować tekst w miejscu, w wersji 2.6 musielibyśmy przekształcić go na listę i z powrotem za pomocą `list(str)` i `''.join(list)`.

```
# Wywołania zmiennych metod

>>> C
bytearray(b'xYelonka')

>>> C.append(b'LMN')                # Python 2.6 wymaga łańcucha znaków o wielkości 1
```

```
TypeError: an integer is required
```

```
>>> C.append(ord('L'))
>>> C
bytearray(b'xYelonkaL')

>>> C.extend(b'MNO')
>>> C
bytearray(b'xYelonkaLMNO')
```

Na obiektach bytearray zgodnie z oczekiwaniami działają wszystkie zwykłe operacje na sekwencjach oraz metody łańcuchów znaków (warto zauważyć, że tak jak w przypadku typu bytes, wyrażenia i metody oczekują argumentów będących obiektami bytes, a nie str).

Operacje na sekwencjach i metody łańcuchów znaków

```
>>> C + b'#!'
bytearray(b'xYelonkaLMNO#!')

>>> C[0]
120

>>> C[1:]
bytearray(b'YelonkaLMNO')

>>> len(C)
12

>>> C
bytearray(b'xYelonkaLMNO')

>>> C.replace('xY', 'mi')          # Ten kod działa w wersji 2.6
TypeError: Type str doesn't support the buffer API

>>> C.replace(b'xY', b'mi')
bytearray(b'mielonkaLMNO')

>>> C
bytearray(b'xYelonkaLMNO')

>>> C * 4
bytearray(b'xYelonkaLMNOxYelonkaLMNOxYelonkaLMNOxYelonkaLMNO')
```

Wreszcie, słowem podsumowania, poniższe przykłady demonstrują, że obiekty bytes oraz bytearray są sekwencjami liczb całkowitych, natomiast obiekty str — sekwencjami znaków.

Dane binarne a dane tekstowe

```
>>> B                      # B jest takie samo jak S w wersji 2.6
b'mielonka'
>>> list(B)
[109, 105, 101, 108, 111, 110, 107, 97]

>>> C
bytearray(b'xYelonkaLMNO')
>>> list(C)
[120, 89, 101, 108, 111, 110, 107, 97, 76, 77, 78, 79]

>>> S
'mielonka'
>>> list(S)
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a']
```

Choć wszystkie trzy typy łańcuchów znaków z Pythona 3.0 zawierają wartości będące znakami i obsługują wiele z tych samych operacji, należy pamiętać, by zawsze:

- używać typu `str` dla danych tekstowych,
- używać typu `bytes` dla danych binarnych,
- używać typu `bytearray` dla danych binarnych, które chcemy modyfikować w miejscu.

Powiązane narzędzia, takie jak pliki, czyli temat kolejnego podrozdziału, często decydują o tym za nas.

Wykorzystywanie plików tekstowych i binarnych

Niniejszy rozdział rozszerza zagadnienia związane z wpływem modelu łańcuchów znaków z Pythona 3.0 na przetwarzanie plików, którego podstawy omówiliśmy we wcześniejszej części książki. Jak wspomniano wcześniej, kluczowy jest tu tryb otwierania pliku — określa on, jaki typ obiektu zostanie wykorzystany do reprezentowania w skrypcie zawartości pliku. Tryb tekstowy narzuca obiekty `str`, natomiast tryb binarny — obiekty `bytes`.

- *Pliki w trybie tekstowym* interpretują zawartość zgodnie z *kodowaniem Unicode* — albo zgodnie z domyślnym ustawieniem platformy, albo według przekazanej nazwy kodowania. Przekazując do `open` nazwę kodowania, możemy wymusić konwersję na różne typy plików Unicode. Pliki w trybie tekstowym wykonują również uniwersalne *translacje końca wiersza*. Domyślnie wszystkie formy końca wiersza odwzorowywane są w skrypcie na pojedynczy znak '`\n`', bez względu na platformę, na której skrypt ten wykonujemy. Jak wspomniano wcześniej, pliki tekstowe obsługują również odczytywanie i zapisywanie *znacznika kolejności bajtów* (BOM) przechowywanego na początku pliku w niektórych schematach kodowania Unicode.
- *Pliki w trybie binarnym* zwracają zamiast tego zawartość w postaci *surowej*, jako sekwencję liczb całkowitych reprezentujących wartości bajtowe, bez kodowania i dekodowania, a także bez translacji końca wiersza.

Drugi argument przekazany do funkcji `open` określa, czy chcemy plik przetwarzać jako tekstowy, czy binarny, tak samo jak w Pythonie 2.X. Dodanie „`b`” do tego łańcucha wymusza tryb binarny, jak na przykład w „`rb`” wczytującym binarne pliki z danymi. Trybem domyślnym jest „`rt`” — jest to to samo co „`r`” i oznacza tekstowe dane wejściowe (tak samo jak w wersji 2.X).

W Pythonie 3.0 ten argument funkcji `open` określa także *typ obiektu* dla reprezentacji zawartości pliku, bez względu na platformę. Pliki tekstowe zwracają obiekt `str` dla odczytu i oczekują tego obiektu dla zapisu, natomiast pliki binarne zwracają obiekt `bytes` dla odczytu i oczekują tego samego (bądź obiektu `bytearray`) dla zapisu.

Podstawy plików tekstowych

W celach demonstracyjnych zaczniemy od prostego pliku wejścia-wyjścia. Dopóki będziemy przetwarzać proste pliki tekstowe (na przykład ASCII), nie będziemy się przejmować obchodem domyślnego kodowania łańcuchów znaków platformy. Pliki w Pythonie 3.0 wyglądają i działają w dużej mierze tak samo jak w 2.X (w tym zakresie podobnie zachowują się łańcuchy znaków). Poniższy kod zapisuje na przykład jeden wiersz tekstu do pliku i wczytuje

go z powrotem w wersji 3.0. Dokładnie tak samo działałoby to w wersji 2.6 (warto zauważyć, że `file` w Pythonie 3.0 nie jest już nazwą wbudowaną, dlatego można jej użyć w kodzie jako zmiennej).

```
C:\misc> c:\python30\python

# Proste pliki tekstowe (i łańcuchy znaków) działają tak samo jak w wersji 2.X

>>> file = open('temp', 'w')                                # Zwraca liczbę zapisanych bajtów
>>> size = file.write('abc\n')                            # Ręczne zamknięcie w celu wyczyszczenia bufora wyjścia
>>> file.close()

>>> file = open('temp')                                 # Domyslnym trybem jest "r" (== "rt"): tekstowe dane wejściowe
>>> text = file.read()
>>> text
'abc\n'
>>> print(text)
abc
```

Tryby tekstowy i binarny w Pythonie 3.0

W Pythonie 2.6 nie ma większej różnicy między plikami tekstowymi a binarnymi — oba typy przyjmują i zwracają zawartość w postaci łańcuchów znaków `str`. Jedyną główną różnicą jest to, że pliki tekstowe automatycznie odwzorowują znaki końca wiersza `\n` na i z `\r\n` w systemie Windows, podczas gdy pliki binarne tego nie robią. Poniższe operacje umieszczałam tutaj połączone w wierszach z uwagi na zwięzłość takiego zapisu.

```
C:\misc> c:\python26\python

>>> open('temp', 'w').write('abd\n')                      # Zapis w trybie tekstowym — dodaje \r
>>> open('temp', 'r').read()                             # Wczytanie w trybie tekstowym — opuszcza \r
'abd\n'
>>> open('temp', 'rb').read()                           # Wczytanie w trybie binarnym — dosłownie
'abd\r\n'

>>> open('temp', 'wb').write('abc\n')                      # Zapis w trybie binarnym
>>> open('temp', 'r').read()                             # \n nie jest rozszerzane do \r\n
'abc\n'
>>> open('temp', 'rb').read()                           # Dane wyjściowe trybu binarnego
```

W Pythonie 3.0 wszystko jest nieco bardziej skomplikowane z uwagi na rozróżnienie pomiędzy `str` (dane tekstowe) i `bytes` (dane binarne). By to zademonstrować, zapiszemy plik tekstowy i wczytamy go z powrotem w obu trybach wersji 3.0. Warto zauważyć, że dla zapisu musimy przekazać obiekt `str`, natomiast wczytanie da nam obiekt `str` lub `bytes`, w zależności od trybu otwarcia pliku.

```
C:\misc> c:\python30\python

# Zapisanie i wczytanie pliku tekstowego

>>> open('temp', 'w').write('abc\n')                      # Dane wyjściowe trybu tekstopowego, przekazanie obiektu str
4

>>> open('temp', 'r').read()                            # Dane wejściowe trybu tekstopowego, zwrócenie obiektu str
'abc\n'

>>> open('temp', 'rb').read()                           # Dane wejściowe trybu binarnego, zwrócenie obiektu bytes
b'abc\r\n'
```

Warto zwrócić uwagę na to, jak w systemie Windows pliki w trybie tekstowym przekładają znak końca wiersza `\n` na `\r\n` dla danych wyjściowych. W przypadku danych wejściowych tryb tekstowy przekłada `\r\n` z powrotem na `\n`, jednak tryb binarny tego nie robi. Tak samo jest w wersji 2.6 i tak właśnie chcemy, by działały dane binarne (nie powinny mieć miejsca żadne translacje), choć działanie to można w Pythonie 3.0 kontrolować za pomocą dodatkowych argumentów funkcji `open`, jeśli jest to pożądane.

Teraz zróbcmy to samo, ale z *plikiem binarnym*. W tym przypadku do zapisania dostarczamy obiekt `bytes` i nadal otrzymujemy z powrotem obiekt `str` lub `bytes`, w zależności od trybu wejścia.

```
# Zapisanie i wczytanie pliku binarnego
>>> open('temp', 'wb').write(b'abc\n')          # Dane wejściowe trybu binarnego, przekazanie obiektu bytes
4
>>> open('temp', 'r').read()                  # Dane wejściowe trybu tekstowego, zwrócenie obiektu str
'abc\n'
>>> open('temp', 'rb').read()                # Dane wejściowe trybu binarnego, zwrócenie obiektu bytes
b'abc\n'
```

Warto zwrócić uwagę na to, że znak końca wiersza `\n` nie jest w przypadku danych wejściowych trybu binarnego rozszerzany do `\r\n` — znów, jest to pożądany wynik dla danych binarnych. Wymagania w zakresie typów oraz zachowanie pliku są te same, nawet jeśli dane zapisywane do pliku binarnego są z natury prawdziwie binarne. Przykładowo w poniższym kodzie "`\x00`" to binarny bajt zerowy, który nie jest znakiem wyświetlanym:

```
# Zapisanie i wczytanie prawdziwie binarnych danych
>>> open('temp', 'wb').write(b'a\x00c')        # Przekazanie obiektu bytes
3
>>> open('temp', 'r').read()                  # Otrzymanie obiektu str
'a\x00c'
>>> open('temp', 'rb').read()                # Otrzymanie obiektu bytes
b'a\x00c'
```

Pliki w trybie binarnym zawsze zwracają zawartość w postaci obiektu `bytes`, jednak do zapisu przyjmują obiekty `bytes` lub `bytearray`, co jest rozsądne, ponieważ `bytearray` jest po prostu zmienną odmianą typu `bytes`. Tak naprawdę większość API w Pythonie 3.0 przyjmujących obiekt `bytes` pozwala także na przekazanie typu `bytearray`.

```
# Obiekty bytearray także działają
>>> BA = bytearray(b'\x01\x02\x03')
>>> open('temp', 'wb').write(BA)
3
>>> open('temp', 'r').read()
'\x01\x02\x03'
>>> open('temp', 'rb').read()
b'\x01\x02\x03'
```

Brak dopasowania typu i zawartości

Warto wiedzieć, że gdy dochodzimy do plików, nie upieczę się nam łamanie reguł związanych z rozróżnieniem typów `str` i `bytes` Pythona. Jak ilustruje to poniższy przykład, jeśli spróbujemy zapisać obiekt `bytes` do pliku tekstowego lub obiekt `str` do pliku binarnego, otrzymamy błędy (skrócone z braku miejsca).

W przypadku zawartości plików typy nie są elastyczne

```
>>> open('temp', 'w').write('abc\n')          # Tryb tekstowy tworzy i wymaga obiektu str
4
>>> open('temp', 'w').write(b'abc\n')
TypeError: can't write bytes to text stream

>>> open('temp', 'wb').write(b'abc\n')         # Tryb binarny tworzy i wymaga obiektu bytes
4
>>> open('temp', 'wb').write('abc\n')
TypeError: can't write str to binary stream
```

Ma to sens — tekst z punktu widzenia danych binarnych nie ma znaczenia, dopóki nie zostanie zakodowany. Choć często można się przemieszczać między tymi typami, kodując obiekty `str` i dekodując `bytes`, zgodnie z wcześniejszymi informacjami z tego rozdziału, zazwyczaj chcemy się trzymać *albo* typu `str` dla danych tekstowych, *albo* typu `bytes` dla danych binarnych. Ponieważ zbiory operacji na obiektach `str` i `bytes` w dużej mierze się pokrywają, wybór ten w przypadku większości programów nie będzie wielkim dilemma (kilka świetnych tego przykładów można znaleźć w omówieniu narzędzi łańcuchów znaków na końcu niniejszego rozdziału).

Poza ograniczeniami związanymi z typem danych w Pythonie 3.0 znaczenie ma także *zawartość pliku*. Pliki wyjściowe w trybie tekstowym wymagają dla swojej zawartości obiektu `str`, a nie `bytes`, dlatego nie da się w tej wersji zapisać prawdziwie binarnych danych do pliku w trybie tekstowym. W zależności od reguł dotyczących kodowania bajty spoza domyślnego zbioru znaków mogą czasami być osadzone w normalnym łańcuchu znaków i zawsze mogą zostać zapisane w trybie binarnym. Ponieważ jednak pliki wejściowe w trybie tekstowym w Pythonie 3.0 muszą być w stanie zdekodować zawartość zgodnie z kodowaniem Unicode, nie istnieje żaden sposób wczytania prawdziwie binarnych danych w trybie tekstowym.

Nie da się wczytać prawdziwie binarnych danych w trybie tekstowym

```
>>> chr(0xFF)                                # FF jest poprawnym znakiem, FE nie
'ÿ'
>>> chr(0xFE)
UnicodeEncodeError: 'charmap' codec can't encode character '\xfe' in position 1...
>>> open('temp', 'w').write(b'\xFF\xFE\xFD')    # Nie można użyć dowolnych bajtów!
TypeError: can't write bytes to text stream
>>> open('temp', 'w').write('\xFF\xFE\xFD')      # Można zapisać, jeśli da się osadzić w obiekcie str
3
>>> open('temp', 'wb').write(b'\xFF\xFE\xFD')      # Można także zapisać w trybie binarnym
3
>>> open('temp', 'rb').read()                  # Zawsze można wczytać jako binarny obiekt bytes
b'\xff\xfe\xfd'
>>> open('temp', 'r').read()                   # Nie da się wczytać tekstu, jeśli nie da się go zdekodować!
UnicodeEncodeError: 'charmap' codec can't encode characters in position 2-3: ...
```

Ostatni z powyższych błędów bierze się z faktu, że wszystkie pliki tekstowe w Pythonie 3.0 są tak naprawdę plikami tekstowymi Unicode, co opisane jest w kolejnym podrozdziale.

Wykorzystywanie plików Unicode

Dotychczas wczytywaliśmy i zapisywaliśmy proste pliki tekstowe oraz binarne, jednak jak wygląda przetwarzanie plików Unicode? Okazuje się, że odczytanie i zapisanie tekstu Unicode zapisanego w pliku jest proste, ponieważ w Pythonie 3.0 wywołanie `open` przyjmuje kodowanie plików tekstowych i w trakcie transferu danych automatycznie je dla nas koduje i dekoduje. Pozwala to na przetwarzanie tekstu Unicode utworzonego z wykorzystaniem schematu kodowania innego od domyślnego dla platformy, a także przechowywanie go w innym kodowaniu w celu konwersji.

Odczyt i zapis Unicode w Pythonie 3.0

Tak naprawdę możemyłańcuch znaków *przekształcić* na inne kodowanie zarówno ręcznie, zapomocą wywołań metod, jak i automatycznie przy wczytaniu z pliku i zapisie do niego. W niniejszym podrozdziale w celach demonstracyjnych wykorzystamy następującyłańcuch Unicode:

```
C:\misc> c:\python30\python
>>> S = 'A\xc4B\xe8C'                                # Łańcuch pięcioznakowy spoza ASCII
>>> S
'AÄBëC'
>>> len(S)
5
```

Kodowanie ręczne

Jak już wiemy, takiłańcuch znaków możemy zawsze ręcznie zakodować na surowe bajty zgodnie z docelową nazwą kodowania:

Ręczne kodowanie za pomocą metod

```
>>> L = S.encode('latin-1')                         # 5 bajtów, jeśli zakodowany jako latin-1
>>> L
b'A\xc4B\xe8C'
>>> len(L)
5

>>> U = S.encode('utf-8')                            # 7 bajtów, jeśli zakodowany jako utf-8
>>> U
b'A\xc3\x84B\xc3\xa8C'
>>> len(U)
7
```

Kodowanie danych wyjściowych pliku

By teraz zapisać naszłańcuch znaków do pliku tekstowego z określonym schematem kodowania, możemy po prostu przekazać pożądaną nazwę kodowania do funkcji `open`. Choć moglibyśmy najpierw ręcznie zakodowaćłańcuch, a następnie zapisać go w trybie binarnym, nie musimy tego robić.

```

# Automatycznie kodowanie przy zapisie

>>> open('latindata', 'w', encoding='latin-1').write(s)      # Zapisane jako latin-1
5
>>> open('utf8data', 'w', encoding='utf-8').write(s)        # Zapisane jako utf-8

>>> open('latindata', 'rb').read()                          # Wczytanie surowych bajtów
b'A\xc4B\xe8C'

>>> open('utf8data', 'rb').read()                          # Różnice w plikach
b'A\xc3\x84B\xc3\xa8C'

```

Dekodowanie danych wejściowych pliku

W podobny sposób, by wczytać dowolne dane Unicode, przekazujemy po prostu nazwę typu kodowania pliku do funkcji open, która automatycznie dekoduje go z surowych bajtów do łańcuchów znaków. Moglibyśmy również wczytać surowe bajty i ręcznie je zdekodować, ale przy wczytywaniu w blokach może to być podchwytniwe (możemy wczytać niepełny znak) i na dodatek nie jest niezbędne.

```

# Automatycznie dekodowanie przy odczycie

>>> open('latindata', 'r', encoding='latin-1').read()          # Dekodowane przy wczytaniu
'AÄBëC'
>>> open('utf8data', 'r', encoding='utf-8').read()            # Zgodnie z typem kodowania
'AÄBëC'

>>> X = open('latindata', 'rb').read()                         # Ręczne dekodowanie:
>>> X.decode('latin-1')                                         # Nie jest konieczne
'AÄBëC'
>>> X = open('utf8data', 'rb').read()                           # utf-8 to kodowanie domyślne
>>> X.decode()
'AÄBëC'

```

Dekodowanie błędnych dopasowań

Wreszcie należy pamiętać, że takie zachowanie plików w Pythonie 3.0 ogranicza rodzaj zawartości, którą możemy załadować jako tekst. Zgodnie z informacjami z poprzedniego podrozdziału Python 3.0 musi tak naprawdę być w stanie zdekodować dane z plików tekstowych do łańcucha znaków str zgodnie z nazwą kodowania Unicode — domyślną lub przekazaną. Przykładowo próba otwarcia pliku z prawdziwie binarnymi danymi w trybie tekstowym raczej w wersji 3.0 nie zadziała, nawet jeśli użyjemy poprawnego typu obiektu:

```

>>> file = open('python.exe', 'r')
>>> text = file.read()
UnicodeDecodeError: 'charmap' codec can't decode byte 0x90 in position 2: ...
>>> file = open('python.exe', 'rb')
>>> data = file.read()
>>> data[:20]
b'MZ\x90\x00\x03\x00\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00\x00'

```

Pierwszy z powyższych przykładów mógłby w Pythonie 2.X zadziałać (normalne pliki nie dekodują tekstu), nawet jeśli najprawdopodobniej nie powinien. Wczytanie tego pliku może zwrócić w łańcuchu znaków niepoprawne dane z uwagi na automatyczne translacje znaków końca wiersza w trybie tekstowym (wszystkie osadzone bajty \r\n zostaną w systemie Windows

przetłumaczone na \n). By potraktować zawartość pliku w Pythonie 2.6 jako tekst Unicode, musimy użyć specjalnych narzędzi zamiast ogólnej funkcji wbudowanej open, co zobaczymy za moment. Najpierw jednak zajmijmy się nieco bardziej wybuchowym zagadnieniem...

Obsługa BOM w Pythonie 3.0

Zgodnie z opisem z wcześniejszej części niniejszego rozdziału niektóre schematy kodowania przechowują specjalną sekwencję znacznika kolejności bajtów (BOM) na początku plików w celu określenia kolejności little- lub big-endian danych bądź zadeklarowania typu kodowania. Python pomija ten znacznik przy odczytce i zapisuje go przy zapisie, jeśli sugeruje to nazwa kodowania, jednak czasami musimy użyć specjalnej nazwy kodowania w celu jawnego wymuszenia przetwarzania sekwencji BOM.

Przykładowo jeśli zapiszemy plik tekstowy w Notatniku systemu Windows, możemy określić jego typ kodowania zgodnie z listą rozwijaną — zwykły tekst ASCII, UTF-8 lub UTF-16 w wersji little- lub big-endian. Jeśli jednowierszowy plik tekstowy o nazwie *spam.txt* zapiszemy w Notatniku jako typ kodowania ANSI, zostanie on zapisany jako prosty tekst ASCII bez znacznika BOM. Kiedy plik ten zostanie wczytany w Pythonie w trybie binarnym, zobaczymy prawdziwe bajty w nim przechowane. Po wczytaniu jako tekst Python domyślnie wykonuje translacje końca wierszy. Możemy zdekodować plik jako tekst UTF-8, ponieważ ASCII jest podziorem tego schematu (a UTF-8 jest kodowaniem domyślnym Pythona 3.0).

```
c:\misc> C:\Python30\python                                # Plik zapisany w Notatniku
>>> import sys
>>> sys.getdefaultencoding()
'utf-8'
>>> open('spam.txt', 'rb').read()                         # Plik tekstowy ASCII (utf-8)
b'mielonka\r\nMIELONKA\r\n'
>>> open('spam.txt', 'r').read()                           # Tryb tekstowy tłumaczy znaki końca wiersza
'mielonka\nMIELONKA\n'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'mielonka\nMIELONKA\n'
```

Jeśli zamiast tego plik ten zapiszemy w Notatniku jako UTF-8, zostanie on poprzedzony trzybajtową sekwencją BOM dla UTF-8 i w celu pominięcia znacznika przez Pythona musimy podać bardziej uszczegółowaną nazwę kodowania („utf-8-sig”).

```
>>> open('spam.txt', 'rb').read()                         # utf-8 z 3-bajtowym znacznikiem BOM
b'\xef\xbb\xbfmielonka\r\nMIELONKA'
>>> open('spam.txt', 'r').read()
'd\x7mielonka\nMIELONKA'
>>> open('spam.txt', 'r', encoding='utf-8').read()
'\ufe0fmielonka\nMIELONKA'
>>> open('spam.txt', 'r', encoding='utf-8-sig').read()
'mielonka\nMIELONKA\n'
```

Jeśli plik przechowamy w Notatniku jako „Unicode big-endian”, otrzymamy w pliku dane w formacie UTF-16, poprzedzone dwubajtową sekwencją BOM. Nazwa kodowania „utf-16” w Pythonie pomija BOM, ponieważ sekwencja ta jest znana (gdyż wszystkie pliki UTF-16 zawierają BOM), natomiast nazwa „utf-16-be” obsługuje format big-endian, ale nie pomija znacznika BOM.

```
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00m\x00i\x00e\x00l\x00o\x00n\x00k\x00a\x00\r\x00\n\x00M\x00I\x00E\x00O\x00L\
\x00O\x00N\x00K\x00A'
>>> open('spam.txt', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\xfe' in position 1:...
```

```
>>> open('spam.txt', 'r', encoding='utf-16').read()
'mielonka\nMIELONKA\n'
>>> open('spam.txt', 'r', encoding='utf-16-be').read()
'\ufeffmielonka\nMIELONKA'
```

Tak samo będzie zazwyczaj w przypadku *danych wyjściowych*. Przy zapisywaniu pliku Unicode w kodzie Pythona potrzebna nam bardziej uszczegółowiona nazwa kodowania, by wymusić w UTF-8 znacznik BOM. Zwykle „utf-8” nie zapisuje znacznika BOM (lub go pomija), natomiast „utf-8-sig” to robi:

```
>>> open('temp.txt', 'w', encoding='utf-8').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()                                # Brak BOM
b'mielonka\r\nMIELONKA\r\n'

>>> open('temp.txt', 'w', encoding='utf-8-sig').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()                                # Zapisanie BOM
b'\xef\xbb\xbfmielonka\r\nMIELONKA\r\n'

>>> open('temp.txt', 'r').read()
'd\x9azmielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()                # Zachowanie BOM
'\ufeffmielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()            # Pomini\u0144cie BOM
'mielonka\nMIELONKA\n'
```

Warto zauważyc, że choć „utf-8” nie pomija BOM, dane *bez* BOM można wczytać zarówno za pomocą nazwy kodowania „utf-8”, jak i „utf-8-sig”. To drugie rozwiązanie należy zastosować w przypadku danych wejściowych, jeśli nie jesteśmy pewni, czy sekwencja BOM obecna jest w pliku.

```
>>> open('temp.txt', 'w').write('mielonka\nMIELONKA\n')
18
>>> open('temp.txt', 'rb').read()                                # Dane bez BOM
b'mielonka\r\nMIELONKA\r\n'
>>> open('temp.txt', 'r').read()                                # Działa dowolna nazwa kodowania utf-8
'mielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8').read()
'mielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-8-sig').read()
'mielonka\nMIELONKA\n'
```

Wreszcie w przypadku nazwy kodowania „utf-16” sekwencja BOM obsługiwana jest automatycznie. W przypadku *danych wyjściowych* dane te zapisywane są zgodnie z ustawieniem kolejności bajtów platformy, a znacznik BOM jest zawsze zapisywany. Dla *danych wejściowych* są one dekodowane zgodnie z sekwencją BOM, a sekwencja ta jest zawsze wycinana. Bardziej uszczegółowione nazwy kodowania UTF-16 mogą określać różne ustawienia kolejności bajtów, choć w niektórych sytuacjach możemy być zmuszeni ręcznie zapisywać lub pomijać BOM, jeśli znacznik ten jest wymagany bądź obecny.

19

```
>>> open('spam.txt', 'rb').read()
b'\xfe\xff\x00m\x00i\x00e\x001\x00o\x00n\x00k\x00a\x00\r\x00\n\x00M\x00I\x00E\x00L\
\x00O\x00N\x00K\x00A'
>>> open('temp.txt', 'r', encoding='utf-16').read()
'mielonka\nMIELONKA\n'
>>> open('temp.txt', 'r', encoding='utf-16-be').read()
'\ufe0fmielonka\nMIELONKA\n'
```

Bardziej uszczegółowione nazwy kodowania UTF-16 działają dobrze w przypadku plików bez BOM, jednak nazwa kodowania „utf-16” wymaga tej sekwencji w celu ustalenia kolejności bajtów:

```
>>> open('temp.txt', 'w', encoding='utf-16-le').write('MIELONKA')
8
>>> open('temp.txt', 'rb').read()                      # OK, jeśli BOM nie ma lub nie oczekiwano
b'M\x00I\x00E\x00L\x00O\x00N\x00K\x00A\x00'
>>> open('temp.txt', 'r', encoding='utf-16-le').read()
'MIELONKA'
>>> open('temp.txt', 'r', encoding='utf-16').read()
UnicodeError: UTF-16 stream does not start with BOM
```

Warto poeksperymentować samodzielnie z tymi typami kodowania lub sprawdzić dokumentację biblioteki Pythona pod kątem szczegółowych informacji o znaczniku kolejności bajtów BOM.

Pliki Unicode w Pythonie 2.6

Powyższe omówienie ma zastosowanie do typów łańcuchów znaków oraz plików z Pythona 3.0. Podobny efekt można uzyskać dla plików Unicode w wersji 2.6, jednak interfejs jest nieco odmienny. Jeśli zastąpimy typ `str` za pomocą `unicode`, a funkcję `open` funkcją `codecs.open`, wynik będzie w wersji 2.6 mniej więcej taki sam.

```
C:\misc> c:\python26\python
>>> S = u'\xc4B\xe8C'
>>> print S
AÄBëC
>>> len(S)
5
>>> S.encode('latin-1')
'A\xc4B\xe8C'
>>> S.encode('utf-8')
'A\xc3\x84B\xc3\xa8C'

>>> import codecs
>>> codecs.open('latindata', 'w', encoding='latin-1').write(S)
>>> codecs.open('utfdata', 'w', encoding='utf-8').write(S)

>>> open('latindata', 'rb').read()
'A\xc4B\xe8C'
>>> open('utfdata', 'rb').read()
'A\xc3\x84B\xc3\xa8C'

>>> codecs.open('latindata', 'r', encoding='latin-1').read()
u'A\xc4B\xe8C'
>>> codecs.open('utfdata', 'r', encoding='utf-8').read()
u'A\xc3\x84B\xc3\xa8C'
```

Inne zmiany narzędzi łańcuchów znaków w Pythonie 3.0

Niektóre z popularnych narzędzi służących w bibliotece standardowej Pythona do przetwarzania łańcuchów znaków zostały zmodyfikowane pod kątem dychotomii nowych typów str i bytes. Nie będziemy omawiać tych skoncentrowanych na aplikacjach narzędzi zbyt szczegółowo w książce poświęconej jądru języka, jednak na zakończenie niniejszego rozdziału przedstawię krótki przegląd czterech najważniejszych narzędzi, na które zmiana ta miała wpływ: modułu dopasowywania wzorców re, modułu danych binarnych struct, modułu serializacji obiektów pickle oraz pakietu xml służącego do analizy składniowej tekstu XML.

Moduł dopasowywania wzorców re

Moduł dopasowywania wzorców Pythona re obsługuje przetwarzanie tekstu w sposób bardziej ogólny niż za pomocą prostych metod łańcuchów znaków, takich jak `find`, `split` oraz `replace`. Dzięki re łańcuchy znaków wyznaczające cele wyszukiwania i dzielenia można opisać za pomocą ogólnych wzorców, a nie tekstu. Moduł ten został uogólniony w taki sposób, by działać na obiektach dowolnego typu łańcucha znaków z Pythona 3.0 — str, bytes oraz bytearray — i zwracać wynikowe podłańcuchy znaków tego samego typu co podmiot operacji.

Oto przykład działania tego modułu w wersji 3.0, dokonujący ekstrakcji podłańcuchów znaków z wiersza tekstu. W przypadku łańcuchów wzorców `(.*)` oznacza dowolny znak `(.)`, zero lub więcej razy `(*)`, zapisany jako dopasowany podłańcuch znaków `(())`. Części łańcucha znaków dopasowane za pomocą wzorca zawartego w nawiasach dostępne są po dopasowaniu za pomocą metod `group` lub `groups`.

```
C:\misc> c:\python30\python
>>> import re
>>> S = 'Dzielny sir Robin nawiewa raz po raz'
      # Wiersz tekstu
      # Zazwyczaj z pliku
>>> B = b'Dzielny sir Robin nawiewa raz po raz'

>>> re.match('(.*) sir (.*) nawiewa (.*)', S).groups()
      ('Dzielny', 'Robin', 'raz po raz')           # Dopasowanie wiersza do wzorca
                                                       # Dopasowane podłańcuchy znaków

>>> re.match(b'(.*) sir (.*) nawiewa (.*)', B).groups()
      (b'Dzielny', b'Robin', b'raz po raz')        # Podłańcuchy znaków bytes
                                                       # Dopasowane podłańcuchy bytes
```

W Pythonie 2.6 wyniki są podobne, jednak dla tekstu spoza zakresu ASCII wykorzystywany jest typ `unicode`, a typ `str` obsługuje zarówno tekst 8-bitowy, jak i binarny:

```
C:\misc> c:\python26\python
>>> import re
>>> S = 'Dzielny sir Robin nawiewa raz po raz'
      # Prosty tekst
      # Binarny tekst Unicode
>>> U = u'Dzielny sir Robin nawiewa raz po raz'

>>> re.match('(.*) sir (.*) nawiewa (.*)', S).groups()
      ('Dzielny', 'Robin', 'raz po raz')

>>> re.match('(.*) sir (.*) nawiewa (.*)', U).groups()
      (u'Dzielny', u'Robin', u'raz po raz')
```

Ponieważ typy bytes i str obsługują w zasadzie ten sam zbiór operacji, rozróżnienie pomiędzy nimi jest praktycznie niezauważalne. Warto jednak zauważyć, że tak jak w innych API, nie

można mieszać typów str i bytes w argumentach ich wywołań w Pythonie 3.0 (choć jeśli nie planujemy dopasowywać wzorców na danych binarnych, najprawdopodobniej nie musimy się tym przejmować).

```
C:\misc> c:\python30\python
>>> import re
>>> S = 'Dzielny sir Robin nawiewa raz po raz'
>>> B = b'Dzielny sir Robin nawiewa raz po raz'

>>> re.match('(.*) sir (.*) nawiewa (.*)', B).groups()
TypeError: can't use a string pattern on a bytes-like object

>>> re.match(b'(.*) sir (.*) nawiewa (.*)', S).groups()
TypeError: can't use a bytes pattern on a string-like object

>>> re.match(b'(.*) sir (.*) nawiewa (.*)', bytearray(B)).groups()
(bytearray(b'Dzielny'), bytearray(b'Robin'), bytearray(b'raz po raz'))

>>> re.match('(.*) sir (.*) nawiewa (.*)', bytearray(B)).groups()
TypeError: can't use a string pattern on a bytes-like object
```

Moduł danych binarnych struct

Moduł struct Pythona, wykorzystywany do tworzenia i ekstrakcji spakowanych danych binarnych z łańcuchów znaków, także działa w Pythonie 3.0 tak samo jak w 2.X, jednak spakowane dane reprezentowane są jedynie w postaci obiektów bytes i bytearray, a nie typu str (co ma sens, biorąc pod uwagę, że przeznaczony jest on do przetwarzania danych binarnych, a nie dowolnie zakodowanego tekstu).

Poniżej znajduje się przykład działania obu wersji Pythona, pakujący trzy obiekty do łańcucha znaków zgodnie ze specyfikacją typu binarnego (tworzą one czterobajtową liczbę całkowitą, czterobajtowy łańcuch znaków oraz dwubajtową liczbę całkowitą).

```
C:\misc> c:\python30\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'jajo', 8)           # Typ bytes w wersji 3.0 (8-bitowy łańcuch znaków)
b'\x00\x00\x00\x07jajo\x00\x08'

C:\misc> c:\python26\python
>>> from struct import pack
>>> pack('>i4sh', 7, 'jajo', 8)           # Typ str w wersji 2.6 (8-bitowy łańcuch znaków)
'\x00\x00\x00\x07jajo\x00\x08'
```

Ponieważ obiekt bytes ma interfejs prawie identyczny jak str w wersjach 3.0 oraz 2.6, większość programistów nie będzie najprawdopodobniej musiała sobie tym zatrzymać głowy — zmiany są dla większości istniejącego kodu nieistotne, zwłaszcza że wczytanie z pliku binarnego automatycznie tworzy obiekt bytes. Choć ostatni test z poniższego przykładu nie działa przy braku dopasowania typów, większość skryptów wczyta dane binarne z pliku, a nie utworzy je w postaci łańcucha znaków.

```
C:\misc> c:\python30\python
>>> import struct
>>> B = struct.pack('>i4sh', 7, 'jajo', 8)
>>> B
b'\x00\x00\x00\x07jajo\x00\x08'

>>> vals = struct.unpack('>i4sh', B)
>>> vals
```

```
(7, b'jajo', 8)

>>> vals = struct.unpack('>i4sh', B.decode())
TypeError: 'str' does not have the buffer interface
```

Poza nową składnią obiektu bytes tworzenie i wczytywanie plików binarnych działa w wersji 3.0 prawie tak samo jak w Pythonie 2.X. Kod taki jak poniższy jest jednym z głównych miejsc, w których programiści zauważają typ obiektu bytes.

```
C:\misc> c:\python30\python
```

```
# Zapisanie wartości do spakowanego pliku binarnego
```

```
>>> F = open('data.bin', 'wb')                                # Otwarcie binarnego pliku wyjścia
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'jajo', 8)               # Utworzenie spakowanych danych binarnych
>>> data
b'\x00\x00\x00\x07jajo\x00\x08'
>>> F.write(data)                                         # W wersji 3.0 typ bytes 3.0, nie str
10
>>> F.close()                                              # Zapisanie do pliku

# Wczytanie wartości ze spakowanego pliku binarnego
```



```
>>> F = open('data.bin', 'rb')                                # Otwarcie binarnego pliku wejścia
>>> data = F.read()                                         # Wczytanie obiektu bytes
>>> data
b'\x00\x00\x00\x07jajo\x00\x08'
>>> values = struct.unpack('>i4sh', data)                  # Ekstrakcja spakowanych danych binarnych
>>> values
(7, b'jajo', 8)                                            # Powrót do obiektów Pythona
```

Po ekstrakcji spakowanych danych binarnych do obiektów Pythona, jak wyżej, jeśli musimy, możemy jeszcze bardziej zagłębić się w świat binarny — łańcuchy znaków można indeksować czy tworzyć z nich wycinki w celu pobrania wartości poszczególnych bajtów. Poszczególne bity można z kolei pobierać z liczb całkowitych za pomocą operatorów poziomu bitowego (więcej informacji na temat zastosowanych tutaj działań można znaleźć we wcześniejszej części książki).

```
>>> values                                              # Wynik struct.unpack
(7, b'jajo', 8)

# Dostęp do bitów przeanalizowanych składowo liczb całkowitych

>>> bin(values[0])                                       # Może dojść do bitów w liczbach całkowitych
'0b111'
>>> values[0] & 0x01                                     # Sprawdzenie pierwszego (najniższego) bitu w liczbie całkowitej
1
>>> values[0] | 0b1010                                  # Bitowe or: załączanie bitów
15
>>> bin(values[0] | 0b1010)                            # Dziesiętnie 15 to binarne 1111
'0b1111'
>>> bin(values[0] ^ 0b1010)                            # Bitowe xor: wyłączenie, jeśli oba są prawdziwe
'0b1101'
>>> bool(values[0] & 0b100)                           # Sprawdzenie, czy bit 3 jest włączony
True
>>> bool(values[0] & 0b1000)                           # Sprawdzenie, czy bit 4 jest ustawiony
False
```

Ponieważ łańcuchy znaków bytes po analizie składowej są sekwencjami niewielkich liczb całkowitych, w podobny sposób możemy przetwarzać ich poszczególne bajty:

```

# Dostęp do bajtów łańcuchów znaków po analizie składniowej, a także do znajdujących się w nich bitów

>>> values[1]
b'jajo'
>>> values[1][0]                                # Łąćuch znaków bytes: sekwencja liczba całkowitych
106
>>> values[1][1:]                               # Wyświetlane jako znaki ASCII
b' ajo'
>>> bin(values[1][0])                          # Może dostać się do bitów bajtów z łańcuchów znaków
'0b1101010'
>>> bin(values[1][0] | 0b1100)                 # Załączenie bitów
'0b1101110'
>>> values[1][0] | 0b1100
110

```

Oczywiście większość programistów Pythona nie zajmuje się bitami binarnymi. Python ma typy obiektów wyższego poziomu, takie jak listy i słowniki, które są najczęściej lepszą metodą reprezentowania informacji w skryptach napisanych w tym języku. Jeśli jednak musimy wykorzystywać bądź przetwarzanie dane niskopoziomowe używane w programach w języku C, bibliotekach sieciowych czy innych interfejsach, Python udostępnia narzędzia mogące nas w tym wspomóc.

Moduł serializacji obiektów pickle

Z modelem pickle spotkaliśmy się krótko w rozdziałach 9. oraz 30. W rozdziale 27. używaliśmy także modułu shelve, który wewnętrznie wykorzystuje moduł pickle. Dla pełności obrazu pamiętajmy, że wersja modułu pickle z Pythona 3.0 zawsze tworzy obiekt bytes, bez względu na domyślny bądź przekazany „protokół” (poziom formatu danych). Możemy się o tym przekonać, wykorzystując wywołanie `dumps` z tego modułu w celu zwrócenia łańcucha znaków pickle dla obiektu.

```

C:\misc> C:\Python30\python
>>> import pickle                                # dumps() zwraca łańcuch znaków pickle

>>> pickle.dumps([1, 2, 3])                      # Domyślny protokół Pythona 3.0: protocol=3, czyli binarny
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'

>>> pickle.dumps([1, 2, 3], protocol=0)          # Protokół ASCII (0), ale nadal obiekt bytes!
b'(lp0\nL1L\naL2L\naL3L\na.'

```

Wynika z tego, że pliki wykorzystywane do przechowywania zserializowanych za pomocą modułu pickle obiektów w Pythonie 3.0 muszą zawsze być otwierane w *trybie binarnym*, ponieważ pliki tekstowe wykorzystują do reprezentowania danych łańcuchy znaków str, a nie bytes. Wywołanie `dumps` próbuje po prostu zapisać zserializowany łańcuch do otwartego pliku wyjściowego.

```

>>> pickle.dump([1, 2, 3], open('temp', 'w'))      # Pliki tekstowe nie działają na obiektach bytes!
TypeError: can't write bytes to text stream        # Pomimo wartości protokołu

>>> pickle.dump([1, 2, 3], open('temp', 'w'), protocol=0)
TypeError: can't write bytes to text stream

>>> pickle.dump([1, 2, 3], open('temp', 'wb'))     # W wersji 3.0 zawsze wykorzystuje tryb binarny

>>> open('temp', 'r').read()
UnicodeEncodeError: 'charmap' codec can't encode character '\u20ac' in ...

```

Ponieważ zserializowane dane nie są tekstem Unicode, który można zdekodować, tak samo jest w przypadku danych wejściowych — poprawne użycie w Pythonie 3.0 wymaga zawsze zapisywania i wczytywania zserializowanych danych w trybach binarnych:

```
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))
>>> pickle.load(open('temp', 'rb'))
[1, 2, 3]
>>> open('temp', 'rb').read()
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

W Pythonie 2.6 (oraz wcześniejszych wersjach) możemy dla zserializowanych danych wykorzystywać także pliki w trybie tekstowym, o ile protokół ustawiony jest na poziom 0 (ustawienie domyślne w Pythonie 2.6) i w spójny sposób wykorzystujemy tryb tekstowy do przekształcania końca wierszy.

```
C:\misc> c:\python26\python
>>> import pickle
>>> pickle.dumps([1, 2, 3])                                # W Pythonie 2.6 domyślny protokół to 0, czyli ASCII
'(lp0\nI1\naI2\naI3\na.'

>>> pickle.dumps([1, 2, 3], protocol=1)
'']q\x00(K\x01K\x02K\x03e.

>>> pickle.dump([1, 2, 3], open('temp', 'w'))    # W Pythonie 2.6 działa tryb tekstowy
>>> pickle.load(open('temp'))
[1, 2, 3]
>>> open('temp').read()
'(lp0\nI1\naI2\naI3\na.'
```

Jeśli ma dla nas znaczenie neutralność względem wersji Pythona albo nie chcemy się przejmować protokołami czy innymi wartościami domyślnymi określonej wersji, należy w przypadku zserializowanych danych zawsze używać plików w trybie binarnym. Poniższy kod działa tak samo w Pythonie 3.0 oraz 2.6.

```
>>> import pickle
>>> pickle.dump([1, 2, 3], open('temp', 'wb'))    # Neutralne względem wersji
>>> pickle.load(open('temp', 'rb'))                # I wymagane w Pythonie 3.0
[1, 2, 3]
```

Ponieważ prawie wszystkie programy pozwalają Pythonowi na automatyczną serializację oraz deserializację obiektów i nie zajmują się zawartością samych zserializowanych danych, wymaganie użycia binarnego trybu plików jest jedyną istotną niezgodnością nowego modelu serializacji Pythona 3 z poprzednimi wersjami. Więcej informacji na temat serializacji obiektów można znaleźć w innych publikacjach książkowych oraz dokumentacji Pythona.

Narzędzia do analizy składniowej XML

XML to język znaczników służący do definiowania ustrukturyzowanych informacji, wykorzystywany powszechnie do definiowania dokumentów oraz danych w Internecie. Choć część informacji można pobierać z tekstu XML za pomocą podstawowych metod łańcuchów znaków lub modułu dopasowywania wzorców `re`, zagnieźdzanie konstrukcji XML i dowolne nazwy atrybutów umożliwiają dokładniejsze przetwarzanie dokumentów w tym języku.

Ponieważ XML jest tak wszechobecnym formatem, sam Python zawiera pełny pakiet narzędzi do analizy składniowej XML obsługujący modele SAX oraz DOM, a także pakiet znany jako `ElementTree` („drzewo elementów”) — specyficzne dla Pythona API służące do analizy skład-

niowej oraz konstruowania XML. Poza podstawową analizą składniową obsługę dodatkowych narzędzi XML, takich jak XPath, Xquery i XSLT, można znaleźć w domenie open source.

XML z definicji reprezentuje tekst w formacie Unicode w celu obsłużenia internacjonalizacji. Choć większość narzędzi do przetwarzania XML w Pythonie zawsze zwracała łańcuchy znaków Unicode, w Pythonie 3.0 ich wyniki zmieniły się z typu `unicode` Pythona 2.X na ogólny typ łańcuchów znaków `str` wersji 3.0. Ma to sens, biorąc pod uwagę, że łańcuch znaków `str` z Pythona 3.0 jest w formacie Unicode, bez względu na to, czy kodowanie to ASCII, czy jakieś inne.

Nie mamy tutaj możliwości za bardzo zagłębić się w szczegóły, ale by nieco zasmakować tej dziedziny wykorzystania Pythona, założymy, że mamy prosty tekstowy plik XML o nazwie `mybooks.xml`:

```
<books>
    <date>2009</date>
    <title>Learning Python</title>
    <title>Programming Python</title>
    <title>Python Pocket Reference</title>
    <publisher>O'Reilly Media</publisher>
</books>
```

i chcemy wywołać skrypt w celu pobrania i wyświetlenia zawartości wszystkich osadzonych znaczników `title`, jak poniżej:

```
Learning Python
Programming Python
Python Pocket Reference
```

Istnieją przynajmniej cztery proste sposoby dokonania tego (nie licząc narzędzi bardziej zaawansowanych, jak XPath). Po pierwsze, moglibyśmy wykonać podstawowe *dopasowanie wzorców* na tekście pliku, choć jeśli tekst jest nieprzewidywalny, może to być niedokładne. Tam, gdzie ma to zastosowanie, można tego dokonać za pomocą omówionego wcześniej modułu `re` — jego metoda `match` szuka dopasowania na początku łańcucha znaków, `search` szuka dopasowania dalej, a wykorzystana tutaj metoda `findall` lokalizuje wszystkie miejsca, w których wzorzec dopasowany zostaje w łańcuchu znaków (wynik zwracany jest w postaci listy dopasowanych podłańcuchów znaków odpowiadających grupom wzorców w nawiasach lub w postaci krotki takich list w przypadku większej liczby grup).

```
# Plik patternparse.py
```

```
import re
text = open('mybooks.xml').read()
found = re.findall('<title>(.*)</title>', text)
for title in found: print(title)
```

Po drugie, by mieć większe możliwości, moglibyśmy wykonać pełną analizę składniową XML za pomocą obsługi *analizy składniowej drzewa DOM* z biblioteki standardowej Pythona. DOM przetwarza tekst XML w drzewo obiektów i udostępnia interfejs do nawigowania w drzewie w celu ekstrakcji atrybutów i wartości znaczników. Interfejs ten jest oficjalną i niezależną od Pythona specyfikacją.

```
# Plik domparse.py
```

```
from xml.dom.minidom import parse, Node
xmltree = parse('mybooks.xml')
for node1 in xmltree.getElementsByTagName('title'):
    for node2 in node1.childNodes:
        if node2.nodeType == Node.TEXT_NODE:
            print(node2.data)
```

Trzecią opcją jest obsługa *analizy składniowej SAX* dla XML w bibliotece standardowej Pythona. W modelu SAX metody klas otrzymują wywołania zwrotne w miarę postępu analizy i wykorzystują informacje o stanie do zapamiętywania, w którym miejscu dokumentu się znajdują, oraz zbierania danych.

```
# Plik saxparse.py

import xml.sax.handler
class BookHandler(xml.sax.handler.ContentHandler):
    def __init__(self):
        self.inTitle = False
    def startElement(self, name, attributes):
        if name == 'title':
            self.inTitle = True
    def characters(self, data):
        if self.inTitle:
            print(data)
    def endElement(self, name):
        if name == 'title':
            self.inTitle = False

import xml.sax
parser = xml.sax.make_parser()
handler = BookHandler()
parser.setContentHandler(handler)
parser.parse('mybooks.xml')
```

Wreszcie system *ElementTree* dostępny w pakiecie etree biblioteki standardowej pozwala uzyskać te same rezultaty co narzędzia do analizy składniowej DOM, jednak z wykorzystaniem mniejszej ilości kodu. To specyficzny dla Pythona sposób analizy składniowej oraz generowania tekstu XML. Po przetworzeniu tekstu API tego systemu daje dostęp do komponentów dokumentu.

```
# Plik etreeparse.py

from xml.etree.ElementTree import parse
tree = parse('mybooks.xml')
for E in tree.findall('title'):
    print(E.text)
```

Po wykonaniu w Pythonie 2.6 lub 3.0 wszystkie cztery skrypty wyświetlają ten sam wynik:

```
C:\misc> c:\python26\python domparse.py
Learning Python
Programming Python
Python Pocket Reference
```

```
C:\misc> c:\python30\python domparse.py
Learning Python
Programming Python
Python Pocket Reference
```

W Pythonie 2.6 część z powyższych skryptów zwraca obiekty łańcuchów znaków `unicode`, podczas gdy w wersji 3.0 wszystkie zwracają łańcuchy znaków `str`, ponieważ typ ten zawiera tekst Unicode (ASCII bądź inny).

```
C:\misc> c:\python30\python
>>> from xml.dom.minidom import parse, Node
>>> xmmtree = parse('mybooks.xml')
>>> for node in xmmtree.getElementsByTagName('title'):
...     for node2 in node.childNodes:
...         if node2.nodeType == Node.TEXT_NODE:
```

```
...
node2.data
...
'Learning Python'
'Programming Python'
'Python Pocket Reference'

C:\misc> c:\python26\python
>>> ...ten sam kod...
...
u'Learning Python'
u'Programming Python'
u'Python Pocket Reference'
```

Programy, które muszą zajmować się wynikami analizy składniowej na nietypowe sposoby, będą musiały uwzględnić zmieniony typ obiektu z Pythona 3.0. I znów, ponieważ wszystkie łańcuchy znaków obsługują prawie identyczne interfejsy w wersjach 2.6 oraz 3.0, na większość skryptów zmiana ta nie będzie miała żadnego wpływu. Narzędzia dostępne dla obiektów unicode z Pythona 2.6 są zazwyczaj swobodnie dostępne dla obiektów str z wersji 3.0.

Niestety, dalsze zagłębianie się w szczegóły analizy składniowej XML wykracza poza zakres niniejszej książki. Osoby zainteresowane analizą składniową tekstu oraz kodu XML odsyłam do bardziej szczegółowego omówienia tego zagadnienia w książce poświęconej dziedzinie aplikacji — *Programming Python*. Więcej informacji na temat modułów re, struct, pickle oraz ogólnie narzędzi do obsługi XML można znaleźć w Internecie, wspomnianej wyżej książce, innych publikacjach oraz dokumentacji biblioteki standardowej Pythona.

Podsumowanie rozdziału

Niniejszy rozdział omawiał zaawansowane typy łańcuchów znaków dostępne w Pythonie 3.0 oraz 2.6 i służące do przetwarzania tekstu Unicode oraz danych binarnych. Jak widzieliśmy, wielu programistów wykorzystuje tekst ASCII — dla nich wystarczający będzie podstawowy typ łańcucha znaków oraz jego wbudowane operacje. W przypadku zastosowań bardziej zaawansowanych model łańcuchów znaków Pythona w pełni obsługuje zarówno tekst Unicode (w wersji 3.0 za pomocą normalnego typu łańcucha znaków, w wersji 2.6 za pomocą specjalnego typu), jak i dane bajtowe (reprezentowane w wersji 3.0 za pomocą typu bytes, a w wersji 2.6 — normalnych łańcuchów znaków).

Dodatkowo zobaczyliśmy, w jaki sposób obiekt pliku Pythona został w wersji 3.0 przekształcony w taki sposób, by automatycznie kodować i dekodować tekst Unicode i radzić sobie z łańcuchami bajtowymi w plikach w trybie binarnym. Wreszcie krótko omówiliśmy niektóre narzędzia do obsługi danych tekstowych oraz binarnych z biblioteki Pythona, a także wypróbowaliśmy ich działanie w wersji 3.0.

W kolejnym rozdziale skoncentrujemy się na zagadnieniach dotyczących budowy narzędzi, przyglądając się sposobom zarządzania dostępem do atrybutów obiektów za pomocą wstawiania automatycznie wykonywanego kodu. Zanim jednak przejdziemy dalej, poniżej znajduje się zbiór pytań sprawdzających to, czego dowiedzieliśmy się tutaj.

Sprawdź swoją wiedzę — quiz

1. Jakie są nazwy i role typów obiektów łańcuchów znaków z Pythona 3.0?
2. Jakie są nazwy i role typów obiektów łańcuchów znaków z Pythona 2.6?
3. W jaki sposób można na siebie odwzorować typy łańcuchów znaków z Pythona 2.6 oraz 3.0?
4. W jaki sposób typy łańcuchów znaków z Pythona 3.0 różnią się od siebie w zakresie obsługiwanych operacji?
5. W jaki sposób można zapisać w kodzie znaki Unicode spoza zakresu ASCII w łańcuchy znaków Pythona 3.0?
6. Jaka są główne różnice pomiędzy plikami w trybie tekstowym a binarnym w Pythonie 3.0?
7. W jaki sposób można wczytać plik z tekstem Unicode zawierający tekst z kodowaniem innym od domyślnego dla naszej platformy?
8. W jaki sposób można utworzyć plik tekstowy Unicode z określonym formatem kodowania?
9. Dlaczego tekst ASCII uznawany jest za rodzaj tekstu Unicode?
10. Jak duży wpływ zmiana typów łańcuchów znaków z Pythona 3.0 ma na nasz kod?

Sprawdź swoją wiedzę — odpowiedzi

1. Python 3.0 zawiera trzy typy łańcuchów znaków: `str` (przeznaczony dla tekstu Unicode, w tym ASCII), `bytes` (dla danych binarnych z bezwzględnymi wartościami bajtów) oraz `bytearray` (zmienną odmianę typu `bytes`). Typ `str` zazwyczaj reprezentuje zawartość przechowywaną w plikach tekstowych, natomiast pozostałe dwa typy reprezentują zawartość przechowywaną w plikach binarnych.
2. Python 2.6 zawiera dwa podstawowe typy łańcuchów znaków: `str` (przeznaczony dla tekstu 8-bitowego oraz danych binarnych) oraz `unicode` (tekst Unicode). Typ `str` wykorzystywany jest zarówno w przypadku zawartości plików tekstowych, jak i binarnych; typ `unicode` wykorzystywany jest dla zawartości plików, która jest bardziej złożona od 8 bitów. Python 2.6 (ale wersje wcześniejsze nie) zawiera również typ `bytearray` z wersji 3.0, jednak został on tam przeniesiony z nowszej wersji i nie przejawią tak ostrego rozróżnienia pomiędzy tekstem a danymi binarnymi, jak ma to miejsce w Pythonie 3.0.
3. Odwzorowanie z typów łańcuchów znaków Pythona 2.6 na typy z wersji 3.0 nie jest bezpośrednie, ponieważ typ `str` z 2.6 równy jest typom `str` i `bytes` z 3.0, a typ `str` z 3.0 równy jest typom `str` i `unicode` z 2.6. Zmienność typu `bytearray` z Pythona 3.0 jest unikalna.
4. Typy łańcuchów znaków Pythona 3.0 wspólnie zasadą prawie wszystkie operacje — wywoływanie metod, działania na sekwencjach, a nawet większe narzędzia, takie jak dopasowywanie wzorców, działają w ten sam sposób. Z drugiej strony, jedynie typ `str` obsługuje operacje formatowania łańcuchów znaków, a typ `bytearray` ma zbiór dodatkowych operacji wykonujących modyfikacje w miejscu. Typy `str` oraz `bytes` mają także metody służące, odpowiednio, do kodowania i dekodowania tekstu.

5. Znaki spoza zakresu ASCII można zapisać w kodzie łańcucha znaków za pomocą sekwencji ucieczki, zarówno szesnastkowych (`\xNN`), jak i Unicode (`\uUNNNN`, `\UNNNNNNNN`). Na niektórych klawiaturach pewne znaki spoza zakresu ASCII — na przykład niektóre znaki z kodowania Latin-1 — można także wpisać w sposób bezpośredni.
6. W Pythonie 3.0 pliki w trybie tekstowym zakładają, że ich zawartość jest tekstem Unicode (nawet jeśli tekst ten jest w kodowaniu ASCII) i automatycznie są one dekodowane przy wczytywaniu oraz kodowane przy zapisywaniu. W przypadku plików w trybie binarnym bajty są przenoszone do pliku i z niego bez zmian. Zawartość plików tekstowych zazwyczaj reprezentowana jest w skrypcie jako obiekty `str`, natomiast zawartość plików binarnych jako obiekty `bytes` (lub `bytearray`). Pliki w trybie tekstowym obsługują również sekwencję znacznika kolejności bajtów BOM dla pewnych typów kodowania i dokonują automatycznie translacji sekwencji końca wiersza z pojedynczego znaku `\n` i na niego na wejściu i wyjściu, o ile opcja ta nie jest w sposób jawnny wyłączona. Pliki w trybie binarnym nie wykonują żadnego z tych kroków.
7. W celu wczytania plików zakodowanych w kodowaniu innym od domyślnego dla naszej platformy wystarczy przekazać nazwę kodowania pliku do wbudowanej funkcji `open` Pythona 3.0 (w wersji 2.6 — `codecs.open()`). Po wczytaniu dane zostaną zdekodowane zgodnie z określonym schematem kodowania. Można również wczytać pliki w trybie binarnym i ręcznie zdekodować bajty na łańcuchy znaków, podając nazwę kodowania, jednak wymaga to większego nakładu pracy i jest nieco bardziej podatne na błędy w przypadku znaków wielobajtowych (możemy przypadkowo wczytać część sekwencji znaków).
8. W celu utworzenia pliku tekstowego Unicode o określonym formacie kodowania należy w Pythonie 3.0 przekazać pożądaną nazwę kodowania do funkcji `open` (w wersji 2.6 — do `codecs.open()`). Przy zapisie do pliku łańcuchy znaków zostaną zakodowane zgodnie z podanym schematem. Można również ręcznie zakodować łańcuch znaków na bajty i zapisać go w trybie binarnym, jednak wymaga to większego nakładu pracy.
9. Tekst ASCII uznawany jest za typ tekstu Unicode, ponieważ jego 7-bitowy przedział wartości jest podzbiorem większości schematów kodowania Unicode. Przykładowo poprawny tekst ASCII jest także poprawnym tekstem Latin-1 (Latin-1 po prostu przypisuje pozostałe możliwe wartości w 8-bitowym bajcie dodatkowym znakom) oraz poprawnym tekstem UTF-8 (UTF-8 definiuje schemat o zmiennej liczbie bajtów reprezentujących więcej znaków, jednak znaki ASCII nadal reprezentowane są przez te same kody, w pojedynczym bajcie).
10. Wpływ zmian modelu łańcuchów znaków z Pythona 3.0 uzależniony jest od wykorzystywanych łańcuchów znaków. W przypadku skryptów wykorzystujących prosty tekst ASCII najprawdopodobniej nie ma to żadnego znaczenia — typ łańcucha znaków `str` działa w tym przypadku tak samo w wersjach 2.6 oraz 3.0. Co więcej, choć narzędzia biblioteki standardowej wykorzystujące łańcuchy znaków, takie jak moduły `re`, `struct`, `pickle` oraz `xml`, mogą w wersji 3.0 korzystać z innych typów niż w Pythonie 2.6, zmiany są dla większości programów bez znaczenia, ponieważ typy `str` i `bytes` z Pythona 3.0 oraz typ `str` z 2.6 obsługują prawie identyczne interfejsy. Jeśli przetwarzamy dane Unicode, potrzebny nam zbiór narzędzi przeniesiony został z `unicode` i `codecs.open()` Pythona 2.6 do `str` i `open` wersji 3.0. Jeśli mamy do czynienia z plikami danych binarnych, musimy obsługiwać ich zawartość w postaci obiektów `bytes`. Ponieważ jednak mają one interfejs podobny do łańcuchów znaków z wersji 2.6, wpływ tych zmian powinien być minimalny.

Zarządzane atrybuty

Niniejszy rozdział rozszerza informacje na temat zaprezentowanych wcześniej technik *przechwytywania atrybutów*, wprowadza kolejne i wykorzystuje je w kilku większych przykładach. Tak jak wszystkie pozostałe rozdziały z tej części książki, został on zaklasyfikowany do tematów bardziej zaawansowanych i opcjonalnych, ponieważ większość programistów aplikacji nie musi się przejmować omawianymi tutaj zagadnieniami — mogą oni pobierać i ustawiać atrybuty obiektów bez martwienia się o samą implementację atrybutów. Zarządzanie dostępem do atrybutów może jednak być szczególnie istotne dla twórców narzędzi — jako część elastycznego API.

Po co zarządza się atrybutami?

Atrybuty obiektów są kluczowym elementem większości programów napisanych w Pythonie — to w nich często przechowujemy informacje o jednostkach przetwarzanych przez nasz skrypt. Normalnie atrybuty są po prostu zmiennymi obiektów — atrybut `name` osoby może na przykład być prostym łańcuchem znaków, pobieranym i ustawianym za pomocą podstawowej składni atrybutów:

```
person.name          # Pobranie wartości atrybutu  
person.name = wartość # Modyfikacja wartości atrybutu
```

W większości przypadków atrybut znajduje się w samym obiekcie lub jest dziedziczony po klasie, z której obiekt ten pochodzi. Ten podstawowy model będzie wystarczający na potrzeby większości programów pisanych w trakcie naszej kariery programisty Pythona.

Czasami jednak wymagana jest większa elastyczność. Przypuśćmy, że napisaliśmy program korzystający z atrybutu `name` w sposób bezpośredni, jednak później wymagania się zmieniają — na przykład decydujemy o tym, że dane te powinny być w jakiś sposób sprawdzane bądź modyfikowane przy pobraniu. Napisanie kodu metod zarządzającego dostępu do wartości atrybutów jest stosunkowo proste (`valid` i `transform` są tutaj wartościami abstrakcyjnymi).

```
class Person:  
    def getName(self):  
        if not valid():  
            raise TypeError('nie można pobrać danych')  
        else:  
            return self.name.transform()  
    def setName(self, value):  
        if not valid(value):
```

```
        raise TypeError('nie można zmienić danych')
    else:
        self.name = transform(value)

person = Person()
person.getName()
person.setName('value')
```

Taka modyfikacja wymaga jednak wprowadzenia zmian w całym programie — we wszystkich miejscach, w których wykorzystywane są dane osobowe ze zmiennej `name`, co może nie być trywialnym zadaniem. Co więcej, takie rozwiązanie wymaga, by program był świadom sposobu eksportowania wartości — w postaci prostych zmiennych lub wywoływanych metod. Jeśli zaczniemy od interfejsu danych opartego na metodach, klient jest odporny na zmiany. Jeśli tak jednak nie będzie, może się to stać problematyczne.

Takie problemy pojawiają się częściej, niż można by tego oczekwać. Wartość komórki w programie przypominającym arkusz kalkulacyjny może na przykład rozpocząć swój żywot jako prosta, niezależna wartość, ale później może ona przemienić się w dowolne obliczenia. Ponieważ interfejs obiektu powinien być na tyle elastyczny, by obsługiwać tego typu przyszłe zmiany bez zakłócania działania istniejącego kodu, późniejsze przełączenie się na metody jest dalekie od ideału.

Wstawianie kodu wykonywanego w momencie dostępu do atrybutów

Lepszym rozwiązaniem byłoby pozwolenie na automatyczne wykonywanie kodu w momencie dostępu do atrybutów, jeśli jest to potrzebne. W różnych miejscach książki spotkaliśmy narzędzia Pythona pozwalające skryptom automatycznie obliczać wartości atrybutów przy ich pobieraniu i sprawdzające poprawność lub modyfikujące wartości atrybutów przy przechowywaniu. W niniejszym rozdziale rozszerzymy informacje na temat wprowadzonych już narzędzi, poznamy inne dostępne, a także przyjrzymy się większym przykładom przypadków użycia z tej dziedziny. W szczególności rozdział ten prezentuje:

- Metody `__getattribute__` oraz `__setattr__` służące do przekierowywania niewydefiniowanych pobrań atrybutów, a także wszystkich przypisań atrybutów do ogólnych metod programów obsługi.
- Metodę `__getattribute__` służącą do przekierowywania wszystkich pobrań atrybutów do ogólnej metody programu obsługi w klasach w nowym stylu z Pythona 2.6 oraz wszystkich klasach wersji 3.0.
- Wbudowaną funkcję `property` służącą do przekierowywania dostępu do uszczegółowionych atrybutów do funkcji programów obsługi pobierania i ustawiania, znanych jako *właściwości*.
- *Protokół deskryptora* służący do przekierowywania dostępu do uszczegółowionych atrybutów do instancji klas z własnymi metodami programów obsługi pobierania i ustawiania.

Z pierwszą i trzecią opcją spotkaliśmy się krótko w części VI książki, pozostałe są nowymi zagadnieniami wprowadzonymi i omówionymi tutaj.

Jak zobaczymy, wszystkie cztery techniki do pewnego stopnia mają wspólne cele i zazwyczaj można rozwiązać określony problem w kodzie za pomocą dowolnej z nich. Istnieją jednak

między nimi pewne istotne różnice. Przykładowo dwie ostatnie z wymienionych technik mają zastosowanie do uszczegółowionych atrybutów, natomiast dwie pierwsze są na tyle ogólne, by móc je wykorzystywać w klasach opartych na delegacji, które muszą przekierowywać dowolne atrybuty do opakowanych obiektów. Jak będziemy mogli się przekonać, wszystkie cztery rozwiązania różnią się także stopniem skomplikowania oraz estetyką, co trzeba zobaczyć w praktyce, by móc dokonać ich samodzielnej oceny.

Poza omówieniem szczegółów leżących u podstaw czterech wymienionych wyżej technik przechwytywania atrybutów w niniejszym rozdziale będziemy mieli okazję zapoznać się z programami większymi od widzianych wcześniej w książce. Studium przypadku `CardHolder` z końca rozdziału powinno służyć jako przykład działania większej klasy do samodzielnego studiowania. Części technik wykorzystanych tutaj użyjemy również w kolejnym rozdziale, w kodzie dekoratorów, dlatego przed przejściem dalej należy upewnić się, że omawiane tutaj zagadnienia zrozumiało się przynajmniej w ogólnym zarysie.

Właściwości

Protokół właściwości pozwala przekierowywać operacje pobierania i ustawiania określonego atrybutu do udostępnianych przez nas funkcji lub metod, pozwalając nam wstawiać kod, który zostanie wykonany automatycznie w momencie dostępu do atrybutów, a także przechwytywać usuwanie atrybutów i udostępniać ich dokumentację, jeśli jest to pożądane.

Właściwości tworzone są za pomocą funkcji wbudowanej `property` i przypisywane są do atrybutów klas, podobnie jak funkcje metod. Tym samym są także dziedziczone przez klasy podległe oraz instancje, podobnie jak wszystkie inne atrybuty klas. Do ich przechwytyujących dostęp funkcji przekazywany jest argument instancji `self`, który umożliwia dostęp do informacji o stanie i do atrybutów klas dostępnych w podmiotowej instancji.

Właściwość zarządza pojedynczym, określonym atrybutem. Choć nie jest w stanie przechwytywać wszystkich dostępów do atrybutów w sposób ogólny, pozwala na kontrolowanie dostępu związanego z pobieraniem i przypisaniem, a także umożliwia zmianę atrybutu z prostych danych na swobodnie obliczany bez zakłócania działania istniejącego kodu. Jak zobaczymy, właściwości są mocno powiązane z deskryptorami — są właściwie ich ograniczoną postacią.

Podstawy

Właściwość tworzona jest za pomocą przypisania wyniku wbudowanej funkcji do atrybutu klasy:

```
attribute = property(fget, fset, fdel, doc)
```

Żaden z argumentów funkcji wbudowanej nie jest wymagany i w razie ich nieprzekazania wszystkie otrzymują wartość domyślną `None`. Takie operacje nie będą wtedy obsługiwane, a próba ich wykonania spowoduje zwrócenie wyjątku. Przy użyciu do `fget` przekazujemy funkcję służącą do przechwytywania pobrania atrybutów, do `fset` — funkcję do przypisań, do `fdel` — funkcję do usuwania atrybutów, natomiast argument `doc` otrzymuje łańcuch znaków dokumentacji dla atrybutu, jeśli jest to pożądane (w przeciwnym razie właściwość kopiuje łańcuch znaków dokumentacji `fget`, o ile jest on podany, który ma wartość domyślną `None`). Funkcja `fget` zwraca obliczoną wartość atrybutu, natomiast `fset` i `fdel` nie zwracają nic (a tak naprawdę zwracają `None`).

Powyzsza funkcja wbudowana zwraca obiekt właściwości przypisywany do nazwy atrybutu, którym chcemy zarządzać w zakresie klasy, gdzie zostanie on odziedziczony przez wszystkie instancje.

Pierwszy przykład

W celu zademonstrowania, jak przekłada się to na działający kod, poniższa klasa wykorzystuje właściwość do śledzenia dostępu do atrybutu o nazwie `name`. Same przechowywane dane noszą nazwę `named_name` w celu uniknięcia konfliktu nazw z właściwością.

```
class Person:                                # W wersji 2.6 należy użyć (object)
    def __init__(self, name):
        self._name = name
    def getName(self):
        print('pobieranie...')
        return self._name
    def setName(self, value):
        print('modyfikacja...')
        self._name = value
    def delName(self):
        print('usunięcie...')
        del self._name
    name = property(getName, setName, delName, "Dokumentacja właściwości name")

bob = Person('Robert Zielony')                # bob ma zarządzany atrybut
print(bob.name)                               # Wykonuje getName
bob.name = 'Robert A. Zielony'               # Wykonuje setName
print(bob.name)                               # Wykonuje delName

print('*'*20)
anna = Person('Anna Czerwona')               # anna także dziedziczy właściwość
print(anna.name)                             # Lub help(Person.name)
```

Właściwości dostępne są w Pythonie 2.6 oraz 3.0, jednak w wersji 2.6 wymagają pochodzenia obiektów w nowym stylu, by przypisania działały poprawnie. By wykonać powyższy kod w Pythonie 2.6, należy dodać `object` jako klasę nadzrędną (w wersji 3.0 również można dodać klasę nadzrędną, jednak jest to domyślne i nie jest wymagane).

Ta akurat właściwość niewiele robi — po prostu przechwytuje i śledzi atrybut — jednak służy do zademonstrowania protokołu. Kiedy powyższy kod zostanie wykonany, dwie instancje dziedziczą właściwość — w ten sam sposób jak odziedziczyłyby dowolny inny atrybut dołączony do swojej klasy. Przechwytywane są jednak operacje dostępu do atrybutów:

```
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja właściwości name
```

Tak jak wszystkie atrybuty klas, właściwości są *dziedziczone* zarówno przez instancje, jak i klasy podrzędne znajdujące się niżej w hierarchii. Jeśli zmodyfikujemy nasz przykład w następujący sposób:

```

class Super:
    ...kod oryginalnej klasy Person...
    name = property(getName, setName, delName, 'Dokumentacja właściwości name')

class Person(Super):
    pass                                # Właściwości są dziedziczone

bob = Person('Robert Zielony')
...reszta bez zmian...

```

wynik będzie taki sam — klasa podzielona Person dziedziczy właściwość name po klasie Super, natomiast instancja bob otrzymuje ją po Person. Jeśli chodzi o dziedziczenie, właściwości działyają tak samo jak normalne metody; ponieważ mają dostęp do argumentów instancji self, mogą uzyskiwać dostęp do informacji o stanie w instancji, podobnie jak metody, co zobaczymy niżej.

Obliczanie atrybutów

Przykład powyżej śledzi po prostu dostęp do atrybutów. Zazwyczaj jednak właściwości robią o wiele więcej — na przykład obliczają wartość atrybutu w sposób dynamiczny w momencie pobierania. Poniższy przykład ilustruje takie zastosowanie:

```

class PropSquare:
    def __init__(self, start):
        self.value = start
    def getX(self):                      # Przy pobraniu atrybutów
        return self.value ** 2
    def setX(self, value):               # Przy przypisaniu atrybutów
        self.value = value
    X = property(getX, setX)            # Brak usuwania i dokumentacji

P = PropSquare(3)                     # 2 instancje klasy z właściwościami
Q = PropSquare(32)                   # Każda ma inną informację o stanie

print(P.X)                           # 3 ** 2
P.X = 4                             # 4 ** 2
print(P.X)                           # 32 ** 2
print(Q.X)

```

Powyzsza klasa definiuje atrybut X, do którego dostęp odbywa się tak, jakby był on danymi statycznymi, jednak tak naprawdę wykonuje kod obliczający jego wartość przy pobraniu. Rezultat przypomina niejawne wywołanie metody. Kiedy kod jest wykonywany, wartość przechowywana jest w instancji w postaci informacji o stanie, jednak za każdym razem, gdy pobieramy ją za pomocą zarządzanego atrybutu, wartość ta jest automatycznie podnoszona do kwadratu.

```

9
16
1024

```

Warto zwrócić uwagę na to, że utworzyliśmy dwie różne instancje. Ponieważ metody właściwości automatycznie otrzymują argument self, mają dostęp do informacji o stanie przechowywanych w instancjach. W naszym przypadku oznacza to, że operacja pobrania oblicza kwadrat danych podmiotowej instancji.

Zapisywanie właściwości w kodzie za pomocą dekoratorów

Choć dodatkowe szczegóły zachowamy na kolejny rozdział, podstawy dekoratorów funkcji wprowadziliśmy wcześniej, w rozdziale 31. Przypomnijmy, że składnia dekoratorów:

```
@dekorator  
def funkcja(argumenty): ...
```

automatycznie przekładana jest przez Pythona na poniższy odpowiednik w celu ponownego dowiązania nazwy funkcji do wyniku obiektu wywołalnego *dekorator*:

```
def funkcja(argumenty): ...  
    funkcja = dekorator(funkcja)
```

Dzięki temu odwzorowaniu okazuje się, że wbudowana funkcja *property* może służyć jako dekorator, definiujący funkcję, która zostanie wykonana automatycznie, kiedy atrybut zostaje pobrany:

```
class Person:  
    @property  
    def name(self): ...  
        # Ponownie dowiązuje name = property(name)
```

Po wykonaniu do udekorowanej metody automatycznie przekazywany jest pierwszy argument funkcji wbudowanej *property*. Jest to tak naprawdę składnia alternatywna dla tworzenia właściwości i ręcznego, ponownego dowiązania nazwy atrybutu.

```
class Person:  
    def name(self): ...  
        name = property(name)
```

Od Pythona 2.6 obiekty właściwości mogą zawierać metody *getter*, *setter* i *deleter*, które przypisują odpowiednio metody akcesora właściwości i zwracają kopię samej właściwości. Możemy je wykorzystać do określenia komponentów właściwości, dekorując także normalne metody, choć komponent *getter* zazwyczaj wypełniany jest automatycznie przez sam fakt tworzenia właściwości.

```
class Person:  
    def __init__(self, name):  
        self._name = name  
  
    @property  
    def name(self):  
        "Dokumentacja właściwości name"  
        print('pobieranie...')  
        return self._name  
  
    @name.setter  
    def name(self, value):  
        print('modyfikacja...')  
        self._name = value  
  
    @name.deleter  
    def name(self):  
        print('usunięcie...')  
        del self._name  
  
    bob = Person('Robert Zielony')  
    print(bob.name)  
    bob.name = 'Robert A. Zielony'  
    print(bob.name)  
    del bob.name  
    # bob ma zarządzany atrybut  
    # Wykonuje komponent getter dla name (pierwszy dostęp do name)  
    # Wykonuje komponent setter dla name (drugi dostęp do name)  
    # Wykonuje komponent deleter dla name (trzeci dostęp do name)
```

```
print('*'*20)
anna = Person('Anna Czerwona')           # anna także dziedziczy właściwość
print(anna.name)                         # Lub help(Person.name)
print(Person.name.__doc__)
```

Tak naprawdę powyższy kod jest odpowiednikiem pierwszego przykładu z tego podrozdziału — dekoracja to w tym przypadku po prostu alternatywny sposób zapisania w kodzie właściwości. Po wykonaniu wyniki są takie same.

```
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja właściwości name
```

W porównaniu z ręcznym przypisywaniem wyników za pomocą `property` w tym przypadku użycie dekoratorów do tworzenia właściwości wymaga jedynie trzech dodatkowych wierszy kodu (różnica jest do pominięcia). Jak to często bywa w przypadku narzędzi alternatywnych, wybór pomiędzy dwoma technikami jest w dużej mierze kwestią subiektywną.

Deskryptory

Deskryptory stanowią alternatywny sposób przechwytywania dostępu do atrybutów i są mocno powiązane z właściwościami omówionymi w poprzednim podrozdziale. Tak naprawdę właściwość jest rodzajem deskryptora — funkcja wbudowana `property` jest uproszczonym sposobem tworzenia określonego typu deskryptora, który wykonuje funkcje metod w momencie dostępu do atrybutów.

Z punktu widzenia funkcjonalności protokół deskryptora pozwala nam przekierować operacje pobierania i ustawiania określonego atrybutu do metod osobnego obiektu klasy, który udostępnimy. Umożliwiają one wstawienie kodu wykonywanego automatycznie w momencie dostępu do atrybutu i pozwalają przechwytywać operacje usuwania atrybutów, a także udostępnić dokumentację, jeśli jest to pożądane.

Deskryptory tworzone są jako niezależne *klasy* i są przypisywane do atrybutów klas tak samo jak funkcje metod. Tak jak wszystkie inne atrybuty klas, są one dziedziczone przez klasy podzielone oraz instancje. Do ich metod przechwytyujących operacje dostępu przekazywane są zarówno sam deskryptor (w postaci argumentu `self`), jak i instancje klasy klienta. Z tego powodu zachowują i wykorzystują własne informacje o stanie, a także informacje o stanie podmiotowej instancji. Przykładowo deskryptor może wywoływać metody dostępne w klasie klienta, a także definiowane przez niego metody specyficzne dla deskryptora.

Podobnie jak właściwość, deskryptor zarządza pojedynczym, określonym atrybutem. Choć nie jest w stanie przechwytywać wszystkich dostępów do atrybutów w sposób uniwersalny, udostępnia kontrolę nad dostępem związanym zarówno z pobieraniem, jak i z przypisywaniem, a także pozwala dowolnie modyfikować atrybut z prostych danych na obliczenia bez zakłócania działania istniejącego kodu. Właściwości tak naprawdę są po prostu wygodnym sposobem tworzenia określonego typu deskryptora i, jak zobaczymy, można je tworzyć w kodzie bezpośrednio w postaci deskryptorów.

Podczas gdy właściwości mają stosunkowo wąski zakres, deskryptory są rozwiązaniem bardziej uniwersalnym. Przykładowo, ponieważ tworzone są w postaci normalnych klas, deskryptory mają swój własny stan, mogą brać udział w hierarchiach dziedziczenia deskryptorów, wykorzystują kompozycję do agregacji obiektów i stanowią naturalną strukturę dla tworzenia w kodzie metod wewnętrznych oraz łańcuchów znaków dokumentacji atrybutów.

Podstawy

Jak wspomniano wcześniej, deskryptory zapisywane są w kodzie jako odrębne klasy i udostępniają metody akcesorów o specjalnych nazwach, służące do operacji dostępu do atrybutów, które mają przechwytywać. Metody pobierania (`__get__`), ustawiania (`__set__`) oraz usuwania (`__delete__`) w klasie deskryptora są wykonywane automatycznie w momencie wystąpienia odpowiedniego typu dostępu do atrybutu przypisanego do instancji klasy deskryptora.

```
class Descriptor:  
    "miejsce na łańcuch znaków dokumentacji"  
    def __get__(self, instance, właściciel): ... # Zwraca wartość atrybutu  
    def __set__(self, instance, właściciel): ... # Nic nie zwraca (None)  
    def __delete__(self, instance): ... # Nic nie zwraca (None)
```

Klasy zawierające dowolną z powyższych metod uznawane są za deskryptory, a ich metody są specjalne, jeśli jedna z nich zostanie przypisana do atrybutu innej klasy — przy dostępie do atrybutu są one wywoływanie automatycznie. Jeśli któraś z metod jest nieobecna, oznacza to zazwyczaj, że odpowiadający jej typ dostępu nie jest obsługiwany. W przeciwieństwie do właściwości pominięcie metody `__set__` pozwala na ponowne zdefiniowanie nazwy w instancji, tym samym ukrywając deskryptor. By atrybut był *tylko do odczytu*, należy zdefiniować metodę `__set__` w taki sposób, aby przechwytywała przypisania i zgłaszała wyjątek.

Argumenty metod deskryptorów

Zanim zajmiemy się utworzeniem jakiegoś realistycznego przykładu, przyjrzymy się podstawom. Do wszystkich trzech opisanych powyżej metod deskryptorów przekazywana jest zarówno instancja klasy deskryptora (`self`), jak i instancja klasy klienta (`instancja`), do której dołączana jest instancja deskryptora.

Metoda dostępu `__get__` dodatkowo otrzymuje argument `właściciel` określający klasę, do której dołączana jest instancja deskryptora. Jej argument `instancja` jest albo instancją, przez którą odbył się dostęp do atrybutu (dla `instancja.atrybut`), lub `None`, jeśli dostęp do atrybutu odbywa się bezpośrednio za pomocą klasy właściciela (dla `klasa.atrybut`). Pierwsza forma zazwyczaj oblicza wartość dla dostępu do instancji, natomiast druga najczęściej zwraca `self`, jeśli obsługiwany jest dostęp do obiektu deskryptora.

Przykładowo w poniższym kodzie po pobraniu `X.attr` Python automatycznie wykonuje metodę `__get__` klasy `Descriptor`, do której przypisany jest atrybut klasy `Subject.attr`. Tak jak w przypadku właściwości, by móc używać tu deskryptorów w Pythonie 2.6, musimy dodać pochodzenie od `object`; w Pythonie 3.0 jest to domniemane, jednak jawnie dopisanie `object` nie zaszkoodzi.

```
>>> class Descriptor(object):  
...     def __get__(self, instance, owner):  
...         print(self, instance, owner, sep='\n')  
...  
>>> class Subject:
```

```

...     attr = Descriptor()                      # Instancja klasy Descriptor jest atrybutem klasy
...
>>> X = Subject()

>>> X.attr
<__main__.Descriptor object at 0x0281E690>
<__main__.Subject object at 0x028289B0>
<class '__main__.Subject'>

>>> Subject.attr
<__main__.Descriptor object at 0x0281E690>
None
<class '__main__.Subject'>

```

Warto zwrócić uwagę na argumenty przekazane automatycznie do metody `__get__` przy pierwszym pobraniu atrybutu. Gdy pobierane jest `X.attr`, działa to tak, jakby nastąpił poniższy przekład (choć `Subject.attr` nie wywołuje tutaj ponownie metody `__get__`):

```
X.attr -> Descriptor.__get__(Subject.attr, X, Subject)
```

Deskryptor wie o tym, że odbywa się do niego dostęp bezpośredni, jeśli argument instancji równy jest `None`.

Deskryptory tylko do odczytu

Jak wspomniano wcześniej, w przeciwieństwie do właściwości, w przypadku deskryptorów pominięcie metody `__set__` nie wystarczy, by atrybut stał się tylko do odczytu, ponieważ zmienną deskryptora można przypisać do instancji. W poniższym kodzie przypisanie atrybutu do `X.a` powoduje przechowanie `a` w instancji obiektu `X`, tym samym ukrywając deskryptor przechowywany w klasie `C`.

```

>>> class D:
...     def __get__(*args): print('pobranie')
...
>>> class C:
...     a = D()
...
>>> X = C()
>>> X.a                                     # Wykonuje metodę __get__ odziedziczonego deskryptora
pobranie
>>> C.a
pobranie
>>> X.a = 99                                # Przechowane w X, ukrywa C.a
>>> X.a
99
>>> list(X.__dict__.keys())
['a']
>>> Y = C()
>>> Y.a                                      # Y nadal dziedziczy deskryptor
pobranie
>>> C.a
pobranie

```

W ten sposób działa przypisywanie atrybutów instancji w Pythonie, co pozwala klasom na selektywne nadpisywanie wartości domyślnych z poziomu klasy w ich instancjach. By uczynić atrybut oparty na deskryptorze atrybutem tylko do odczytu, należy przechywić przypisanie w klasie deskryptora i zgłosić wyjątek, tak by zapobiec przypisaniu atrybutu. Przy przypisywaniu atrybutu będącego deskryptorem Python obchodzi normalne zachowanie na poziomie instancji i przekierowuje operację do obiektu deskryptora.

```

>>> class D:
...     def __get__(*args): print('pobranie')
...     def __set__(*args): raise AttributeError('nie można ustawić')
...
>>> class C:
...     a = D()
...
>>> X = C()                                     # Przekierowane do C.a.__get__
pobranie
>>> X.a                                         # Przekierowane do C.a.__set__
AttributeError: nie można ustawić

```



Należy także uważać, by nie pomylić metody deskryptora `__delete__` z ogólną metodą `__del__`. Ta pierwsza wywoływana jest przy próbach usunięcia nazwy zarządzanego atrybutu w instancji klasy właściciela. Ta druga jest ogólną metodą destruktora instancji, wykonywaną gdy instancja klasy dowolnego typu ma być wyczyszczona z pamięci. Metoda `__delete__` jest bliżej związana z ogólną metodą usuwania atrybutu `__delattr__`, z którą spotkamy się w dalszej części rozdziału. Więcej informacji na temat metod przeciążania operatorów można znaleźć w rozdziale 29.

Pierwszy przykład

By zobaczyć, jak to wszystko łączy się ze sobą w bardziej realistycznym kodzie, zacznijmy od tego samego pierwszego przykładu, jaki napisaliśmy dla właściwości. Poniższy kod definiuje deskryptor przechwytyjący dostęp do atrybutu o nazwie `name` w swoich klientach. Jego metody wykorzystują argument instancji w celu uzyskania dostępu do informacji o stanie z podmowej instancji, w której przechowywany jest łańcuch znaków imienia i nazwiska. Tak jak właściwości, deskryptory działają poprawnie jedynie dla klas w nowym stylu, dlatego w Pythonie 2.6 trzeba dopilnować pochodzenia obu klas od `object`.

```

class Name:                                     # W Pythonie 2.6 należy użyć (object)
    "Dokumentacja deskryptora name"
    def __get__(self, instance, owner):
        print('pobieranie...')
        return instance._name
    def __set__(self, instance, value):
        print('modyfikacja...')
        instance._name = value
    def __delete__(self, instance):
        print('usunięcie...')
        del instance._name

class Person:                                    # W Pythonie 2.6 należy użyć (object)
    def __init__(self, name):
        self._name = name
    name = Name()                                # Przypisanie deskryptora do atrybutu

bob = Person('Robert Zielony')                  # bob ma zarządzany atrybut
print(bob.name)                                 # Wykonuje Name.__get__
bob.name = 'Robert A. Zielony'                 # Wykonuje Name.__set__
print(bob.name)                                 # Wykonuje Name.__delete__

print('*'*20)
anna = Person('Anna Czerwona')                # anna także dziedziczy deskryptor
print(anna.name)                               # Lub help(Name)
print(Name.__doc__)

```

Warto zwrócić uwagę na to, jak w powyższym kodzie przypisujemy instancję klasy deskryptora do *atrybutu klasy* w klasie klienta. Z tego powodu jest on dziedziczony przez wszystkie instancje klasy, tak samo jak jej metody. Tak naprawdę *musimy* przypisać deskryptor do atrybutu klasy w taki właśnie sposób — nie będzie on działał, jeśli zamiast tego przypiszemy go do atrybutu instancji `self`. Po wykonaniu metody `__get__` deskryptora przekazywane są do niej trzy obiekty w celu zdefiniowania kontekstu:

- `self` jest instancją klasy `Name`,
- `instance` jest instancją klasy `Person`,
- `owner` to klasa `Person`.

Po wykonaniu powyższego kodu metody deskryptora przechwytyują próby uzyskania dostępu do atrybutów, podobnie jak wersja kodu z właściwościami. Tak naprawdę wynik będzie znowu taki sam:

```
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
Dokumentacja deskryptora name
```

Podobnie jak w przykładzie z właściwościami, instancja klasy deskryptora jest atrybutem klasy i tym samym *dziedziczona* jest przez wszystkie instancje klasy klienta oraz klasy pod-rzędne. Jeśli w naszym przykładzie zmienimy klasę `Person` w następujący sposób, wynik skryptu będzie taki sam:

```
...
class Super:
    def __init__(self, name):
        self._name = name
    name = Name()

class Person(Super):                      # Deskryptory są dziedziczone
    pass
...
```

Warto również zauważyć, że kiedy klasa deskryptora nie jest przydatna poza klasą klienta, zupełnie rozsądne jest składniowe osadzenie definicji deskryptora wewnątrz klienta. Oto jak wyglądać będzie nasz przykład w przypadku zastosowania *klasy osadzonej*:

```
class Person:
    def __init__(self, name):
        self._name = name

class Name:                                # Zastosowanie klasy osadzonej
    "Dokumentacja deskryptora name"
    def __get__(self, instance, owner):
        print('pobieranie...')
        return instance._name
    def __set__(self, instance, value):
        print('modyfikacja...')
        instance._name = value
    def __delete__(self, instance):
```

```
    print('usuńcie...')  
    del instance._name  
name = Name()
```

W takim kodzie Name staje się zmienną lokalną w zakresie instrukcji klasy Person i tym samym nie będzie wchodziła w konflikt z żadnymi nazwami spoza klasy. Powyższa wersja działa tak samo jak oryginalna — przenieśliśmy po prostu definicję klasy deskryptora do zakresu klasy klienta — jednak ostatni wiersz kodu sprawdzającego musi się zmienić, by pobierać łańcuch znaków dokumentacji z nowej lokalizacji.

```
...  
print(Person.Name.__doc__) # Różnica: już nie Name.__doc__ poza klasą
```

Obliczone atrybuty

Tak jak było w przypadku zastosowania właściwości, nasz pierwszy przykład deskryptora z powyższego podrozdziału nie robił nic specjalnego — wyświetlał po prostu komunikaty śledzenia dla dostępu do atrybutów. W praktyce deskryptory można także wykorzystać do obliczania wartości atrybutów za każdym razem, gdy są one pobierane. Ilustruje to poniższy przykład — jest on inną wersją tego samego przykładu, jaki utworzyliśmy dla właściwości. Wykorzystuje on deskryptor do automatycznego obliczenia kwadratu wartości atrybutu z każdym pobraniem.

```
class DescSquare:  
    def __init__(self, start):  
        self.value = start # Każdy deskryptor ma własny stan  
    def __get__(self, instance, owner):  
        return self.value ** 2 # Przy pobieraniu atrybutów  
    def __set__(self, instance, value):  
        self.value = value # Przy przypisywaniu atrybutów  
                                # Brak usuwania i dokumentacji  
  
class Client1:  
    X = DescSquare(3) # Przypisanie instancji deskryptora do atrybutu klasy  
  
class Client2:  
    X = DescSquare(32) # Inna instancja w innej klasie klienta  
                        # Można także utworzyć kod dwóch instancji tej samej klasy  
  
c1 = Client1()  
c2 = Client2()  
  
print(c1.X) # 3 ** 2  
c1.X = 4  
print(c1.X) # 4 ** 2  
print(c2.X) # 32 ** 2
```

Po wykonaniu wynik powyższego przykładu będzie taki sam jak oryginalnej wersji opartej na właściwościach, jednak tym razem próby dostępu do atrybutów przechwytywane są przez obiekt deskryptora klas.

```
9  
16  
1024
```

Wykorzystywanie informacji o stanie w deskryptorach

Jeśli zastanowimy się chwilę nad dwoma utworzonymi dotychczas przykładami deskryptorów, możemy zauważyc, że swoje informacje pobierają one z różnych miejsc. Pierwszy (przykład z atrybutem `name`) wykorzystuje dane przechowywane w *instancji* klienta, natomiast drugi (przykład z kwadratem atrybutu) wykorzystuje dane dołączone do samego obiektu *deskryptora*. Tak naprawdę deskryptory mogą wykorzystywać stan *zarówno* instancji, jak i deskryptora lub też dowolną ich kombinację:

- stan deskryptora wykorzystywany jest do zarządzania danymi wewnętrznymi z punktu widzenia działania deskryptora,
- stan instancji zapisuje informacje powiązane z klasą klienta i prawdopodobnie przez nią utworzone.

Metody deskryptorów mogą wykorzystywać oba rozwiązania, jednak stan deskryptora często sprawia, że używanie specjalnych konwencji nazewnictwa w celu uniknięcia konfliktów między nazwami danych deskryptora przechowywanych w instancji nie jest konieczne. Poniższy deskryptor dołącza na przykład informacje do własnej instancji, dzięki czemu nie wchodzą one w konflikt z informacjami instancji klasy klienta.

```
class DescState:                                     # Wykorzystanie stanu deskryptora
    def __init__(self, value):
        self.value = value
    def __get__(self, instance, owner):      # Przy pobieraniu atrybutów
        print('pobranie DescState')
        return self.value * 10
    def __set__(self, instance, value):      # Przy przypisywaniu atrybutów
        print('ustawienie DescState')
        self.value = value

class CalcAttrs:
    X = DescState(2)                           # Atrybut klasy deskryptora
    Y = 3                                      # Atrybut klasy
    def __init__(self):                         # Atrybut instancji
        self.Z = 4

obj = CalcAttrs()                                # X jest obliczane, pozostałe nie są
print(obj.X, obj.Y, obj.Z)                      # Przypisanie X jest przechwytywane
obj.X = 5
obj.Y = 6
obj.Z = 7
print(obj.X, obj.Y, obj.Z)
```

Informacje o wartościach znajdują się w powyższym kodzie jedynie w *deskryptorze*, dzięki czemu nie wystąpi konflikt, jeśli ta sama nazwa zostanie użyta w instancji klienta. Warto zauważyć, że zarządzamy tutaj jedynie atrybutem deskryptora — przechwytywane są próby pobrania oraz ustawienia zmiennej `X`, jednak dostęp do `Y` oraz `Z` nie (zmienna `Y` dołączona jest do klasy klienta, natomiast `Z` — do instancji). Po wykonaniu powyższego kodu zmienna `X` jest po pobraniu obliczana.

```
pobranie DescState
20 3 4
ustawienie DescState
pobranie DescState
50 6 7
```

Deskryptor może także przechowywać lub wykorzystywać atrybut dołączony do instancji klasy klienta zamiast do siebie. Deskryptor z poniższego przykładu zakłada, że instancja ma atrybut `_Y` dołączony do klasy klienta, i wykorzystuje go do obliczenia reprezentowanej przez niego wartości.

```
class InstState:
    def __get__(self, instance, owner):
        print('pobranie InstState')
        return instance._Y * 100
    def __set__(self, instance, value):
        print('ustawienie InstState')
        instance._Y = value

# Klasa klienta

class CalcAttrs:
    X = DescState(2)                      # Wykorzystanie stanu instancji
    Y = InstState()                        # Założenie ustawienia przez klasę klienta
    def __init__(self):
        self._Y = 3                          # Atrybut deskryptora klasy
        self.Z = 4                           # Atrybut deskryptora klasy

    obj = CalcAttrs()
    print(obj.X, obj.Y, obj.Z)              # X oraz Y są obliczane, Z nie jest
    obj.X = 5                            # Przypisania X oraz Y są przechwytywane
    obj.Y = 6
    obj.Z = 7
    print(obj.X, obj.Y, obj.Z)
```

Tym razem `X` oraz `Y` przypisywane są do deskryptorów i obliczane przy pobraniu (w poprzednim przykładzie do `X` przypisywany był deskryptor). Nowy deskryptor z powyższego przykładu nie ma własnych informacji, jednak wykorzystuje atrybut, który zakłada, że istnieje w instancji. Atrybut ten nosi nazwę `_Y` w celu uniknięcia konfliktu z nazwą z samego deskryptora. Po wykonaniu tej wersji kodu wyniki będą podobne, jednak drugi atrybut jest zarządzany — wykorzystywany jest do tego stan znajdujący się w instancji, a w nie w deskryptorze.

```
pobranie DescState
pobranie InstState
20 300 4
ustawienie DescState
ustawienie InstState
pobranie DescState
pobranie InstState
50 600 7
```

Zarówno stan z deskryptora, jak i stan z instancji pełnią swoje role. Tak naprawdę na tym właśnie polega przewaga deskryptorów nad właściwościami — ponieważ mają one własny stan, mogą z łatwością przechowywać dane wewnętrznie, bez dodawania ich do przestrzeni nazw obiektu instancji klienta.

Powiązania pomiędzy właściwościami a deskryptorami

Jak wspomniano wcześniej, właściwości i deskryptory są ze sobą silnie powiązane — wbudowana funkcja `property` jest po prostu wygodnym sposobem tworzenia deskryptora. Skoro już wiemy, jak działają oba rozwiązania, powinniśmy być w stanie zobaczyć, że możliwe jest symulowanie funkcji wbudowanej `property` za pomocą klasę deskryptora, jak poniżej:

```

class Property:
    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc
                    # Zapisanie metod bez wiązania
                    # lub innych obiektów wywoływalnych

    def __get__(self, instance, instancetype=None):
        if instance is None:
            return self
        if self.fget is None:
            raise AttributeError("nie można pobrać atrybutu")
        return self.fget(instance)           # Przekazanie instancji do self
                                            # w akcesorach właściwości

    def __set__(self, instance, value):
        if self.fset is None:
            raise AttributeError("nie można ustawić atrybutu")
        self.fset(instance, value)

    def __delete__(self, instance):
        if self.fdel is None:
            raise AttributeError("nie można usunąć atrybutu")
        self.fdel(instance)

class Person:
    def getName(self): ...
    def setName(self, value): ...
    name = Property(getName, setName)      # Użyć jak property()

```

Powyższa klasa `Property` przechwytuje dostęp do atrybutów za pomocą protokołu deskryptora i przekierowuje żądania do funkcji lub metod przekazanych i zapisanych w stanie deskryptora, kiedy klasa jest tworzona. Pobranie atrybutu jest na przykład przekierowywane z klasy `Person` do metody `__get__` klasy `Property` i z powrotem do metody `getName` klasy `Person`. W przypadku deskryptorów wszystko „po prostu działa”.

Warto zauważyć, że przykład odpowiednika klasy deskryptora obsługuje jedynie proste zastosowania właściwości. By skorzystać ze składni dekoratorów z @ w celu określenia operacji ustawiania i usuwania, nasza klasa `Property` musiałaby zostać rozszerzona za pomocą metod setter oraz deleter, zapisujących udekorowaną funkcję akcesora i zwracających obiekt właściwości (powinno tu wystarczyć `self`). Ponieważ funkcja wbudowana `property` robi to za nas, pominiemy tutaj kod takiego rozszerzenia.

Należy także zwrócić uwagę na fakt, iż deskryptory wykorzystywane są do implementowania `__slots__` z Pythona. Możemy unikać słowników atrybutów instancji dzięki przechwyceniu nazw slotów za pomocą deskryptorów przechowywanych na poziomie klasy. Więcej informacji na temat slotów można znaleźć w rozdziale 31.



W rozdziale 38. będziemy także wykorzystywać deskryptory do implementowania dekoratorów funkcji mających zastosowanie zarówno do funkcji, jak i metod. Jak zobaczymy, ponieważ deskryptory otrzymują zarówno instancje deskryptora, jak i klasy podmiotowej, w tej roli dobrze się sprawdzają, choć funkcje zagnieżdżone są zazwyczaj prostszym rozwiązaniem.

Metody `__getattr__` oraz `__getattribute__`

Dotychczas omówiliśmy właściwości oraz deskryptory — narzędzia służące do zarządzania określonymi atrybutami. Metody przeciążania operatorów `__getattr__` oraz `__getattribute__` udostępniają jeszcze inne sposoby przechwytywania pobrań atrybutów dla instancji klas. Tak jak właściwości oraz deskryptory, pozwalają one wstawiać kod, który będzie wykonywany w momencie dostępu do atrybutów. Jak jednak zobaczymy, te dwie metody mogą być wykorzystywane w sposób bardziej uniwersalny.

Przechwytywanie pobierania atrybutów ma dwie odmiany, których kod tworzy się za pomocą dwóch różnych metod:

- Metoda `__getattr__` wykonywana jest dla atrybutów *niezdefiniowanych*, to znaczy atrybutów nieprzechowywanych w instancji lub dziedziczonych po jednej z jej klas.
- Metoda `__getattribute__` wykonywana jest dla *każdego* atrybutu, dlatego wykorzystując ją, trzeba uważać i unikać pętli rekurencyjnych przy przekazywaniu dostępu do atrybutów do klasy nadzędnej.

Z pierwszą z powyższych metod spotkaliśmy się w rozdziale 29. — jest ona dostępna we wszystkich wersjach Pythona. Druga dostępna jest w Pythonie 2.6 dla klas w nowym stylu oraz dla wszystkich klas (domniemane: w nowym stylu) w wersji 3.0. Dwie powyższe metody są reprezentantami zbioru metod przechwytywania atrybutów, do którego należą także `__setattr__` oraz `__delattr__`. Ponieważ metody te pełnią podobne role, potraktujemy je tutaj jako jedno zagadnienie.

W przeciwnieństwie do właściwości oraz deskryptorów metody te są częścią protokołu *przeciążania operatorów* Pythona — metod klas o specjalnych nazwach, dziedziczących przez klasy podrzędne i wykonywanych automatycznie, gdy instancje użyte zostają w domniemanej wbudowanej operacji. Tak jak wszystkie metody klasy, każda z nich po wywołaniu otrzymuje pierwszy argument `self`, co daje dostęp do wszelkich wymaganych informacji o stanie instancji lub innych metodach klasy.

Metody `__getattr__` oraz `__getattribute__` są także bardziej *ogólne* od właściwości i deskryptorów — można je wykorzystywać do przechwytywania dostępu do dowolnych (nawet wszystkich) pobrań atrybutów instancji, a nie tylko określonych nazw, do których zostały one przypisane. Z tego powodu metody te są dobrze przystosowane do ogólnych wzorców kodu opartego na *delegacji* — można je wykorzystywać do implementowania obiektów opakowujących, zarządzających wszystkimi dostępami do atrybutów dla obiektów osadzonych. Z kolei w przypadku właściwości bądź deskryptorów musimy zdefiniować po jednym z nich dla każdego atrybutu, który chcemy przechwytywać.

Wreszcie te dwie metody są o wiele *węższe* i bardziej *skoncentrowane* w swoim działaniu od wcześniej rozważanych alternatyw — przechwytyują jedynie pobrań atrybutów, a nie przy pisania. By przechwytywać również zmianę atrybutu przez jego przypisanie, musimy utworzyć kod `__setattr__` — metody przeciążania atrybutów wykonywanej dla każdego ich pobra nia, w przypadku której należy uważać, by uniknąć pętli rekurencyjnych poprzez przekierowanie przypisania atrybutów za pośrednictwem słownika przestrzeni nazw instancji.

Choć zdarza się to o wiele rzadziej, możemy także utworzyć kod metody przeciążania operatorów `__delattr__` (która w ten sam sposób musi unikać pętli) w celu przechwycenia operacji usunięcia atrybutów. W przeciwnieństwie do tego rozwiązania właściwości i deskryptory z założenia przechwytyują operacje pobierania, ustawiania oraz usuwania.

Większość metod przeciążania operatorów została wprowadzona we wcześniejszej części książki. Tutaj rozszerzymy jedynie informacje o ich zastosowaniach i omówimy ich rolę w szerszym kontekście.

Podstawy

Metody `__getattr__` oraz `__setattr__` wprowadzone zostały w rozdziałach 29. oraz 31.; o `__getattribute__` wspomnialiśmy krótko w rozdziale 31. W skrócie, jeśli klasa definiuje lub dziedziczy poniższe metody, zostaną one wykonane automatycznie, jeżeli instancja zostanie użyta w kontekście opisanyem za pomocą komentarzy znajdujących się z prawej strony.

```
def __getattr__(self, nazwa):          # Przy pobraniu niezdefiniowanego atrybutu [obiekt.nazwa]
def __getattribute__(self, nazwa):       # Przy pobraniu wszystkich atrybutów [obiekt.nazwa]
def __setattr__(self, nazwa, wartość):   # Przy przypisaniu wszystkich atrybutów [obiekt.nazwa=wartość]
def __delattr__(self, nazwa):           # Przy usunięciu wszystkich atrybutów[del obiekt.nazwa]
```

W każdej z metod `self` jest jak zwykle obiektem instancji docelowej, `nazwa` to nazwa atrybutu, do którego dostęp ma miejsce, a `wartość` to obiekt przypisywany do atrybutu. Dwie metody pobierania normalnie zwracają wartość atrybutu, natomiast pozostałe dwie nie zwracają niczego (`None`). Przykładowo w celu przechwycenia operacji pobrania każdego atrybutu możemy użyć dowolnej z dwóch pierwszych metod wymienionych wyżej, natomiast by przechwycić przypisanie każdego atrybutu, możemy skorzystać z trzeciej:

```
class Catcher:
    def __getattr__(self, name):
        print('Pobranie:', name)
    def __setattr__(self, name, value):
        print('Ustawienie:', name, value)

X = Catcher()
X.job                         # Wyświetla "Pobranie: job"
X.pay                          # Wyświetla "Pobranie: pay"
X.pay = 99                      # Wyświetla "Ustawienie: pay 99"
```

Takie struktury kodu można wykorzystać do implementowania wzorca kodu *delegacji*, z którym spotkaliśmy się wcześniej, w rozdziale 30. Ponieważ wszystkie atrybuty przekierowywane są do naszych metod przechwytyujących w sposób ogólny, możemy sprawdzać ich poprawność i przekazywać je do osadzonych, zarządzanych obiektów. Poniższa klasa (zapozyczona z rozdziału 30.) śledzi na przykład każdą operację pobrania atrybutu wykonaną dla innego obiektu przekazanego do klasy opakowującej.

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                         # Zapisanie obiektu
    def __getattr__(self, attrname):
        print('Śledzenie:', attrname)                # Śledzenie pobrania
        return getattr(self.wrapped, attrname)         # Delegacja pobrania
```

Taka analogia nie występuje w przypadku właściwości oraz deskryptorów, z wyjątkiem pisania kodu akcesorów dla każdego możliwego atrybutu w każdym możliwym opakowanym obiekcie.

Unikanie pętli w metodach przechwytyjących atrybuty

Metody te są stosunkowo proste w użyciu. Jedynym skomplikowanym elementem jest w nich potencjalna możliwość *zapętlania* (czyli rekurencyjności). Ponieważ metoda `__getattr__` wywoływana jest jedynie dla atrybutów niezdefiniowanych, może swobodnie pobierać inne atrybuty w ramach swojego kodu. Ponieważ jednak metody `__getattribute__` oraz `__setattr__` wykonywane są dla wszystkich atrybutów, musimy być ostrożni przy dostępie do innych atrybutów, by uniknąć ponownego wywoływanie ich wzajemnie i uruchomienia rekurencyjnej pętli.

Przykładowo kolejne pobranie atrybutu wewnętrz kodu metody `__getattribute__` powoduje ponowne wywołanie tej metody, a kod zapętli się, dopóki nie wyczerpie się pamięć.

```
def __getattribute__(self, name):
    x = self.other                                # PĘTLA!
```

By obejść to ograniczenie, należy przekierować operację pobrania za pośrednictwem klasy nadzędnej wyżej w hierarchii, zamiast przechodzić do wersji z klasy z tego poziomu — klasa `object` zawsze będzie klasą nadzędzienną i świetnie się sprawdza w tej roli:

```
def __getattribute__(self, name):
    x = object.__getattribute__(self, 'other')      # Wymuszenie klasy wyżej w celu uniknięcia siebie
```

W przypadku metody `__setattr__` sytuacja jest podobna. Przypisanie dowolnego atrybutu wewnętrz tej metody wywołuje ponownie `__setattr__` i tworzy podobną pętlę.

```
def __setattr__(self, name, value):
    self.other = value                            # PĘTLA!
```

By obejść ten problem, należy zamiast tego przypisać atrybut jako klucz do słownika przestrzeni nazw instancji `__dict__`. Pozwala to uniknąć bezpośredniego przypisania atrybutu.

```
def __setattr__(self, name, value):
    self.__dict__['other'] = value                # Użycie słownika atrybutów w celu uniknięcia siebie
```

Choć jest to rzadziej stosowane rozwiązanie, metoda `__setattr__` może także przekazywać własne przypisy atrybutów do klasy nadzędnej wyżej w hierarchii w celu uniknięcia pętli, tak samo jak metoda `__getattribute__`.

```
def __setattr__(self, name, value):
    object.__setattr__(self, 'other', value)       # Wymuszenie wyższej klasy nadzędnej w celu uniknięcia siebie
```

Inaczej sytuacja przedstawia się w przypadku metody `__getattribute__` — w celu uniknięcia pętli *nie możemy* użyć sztuczki ze słownikiem `__dict__`.

```
def __getattribute__(self, name):
    x = self.__dict__['other']                     # PĘTLA!
```

Pobranie samego atrybutu `__dict__` powoduje ponowne wywołanie metody `__getattribute__`, co wywołuje pętlę rekurencyjną. Dziwne, ale prawdziwe!

Metoda `__delattr__` w praktyce wykorzystywana jest rzadko, jeśli jednak tak jest, wywoływana jest dla każdego usunięcia atrybutu (tak samo jak `__setattr__` wywoływana jest dla każdego przypisania atrybutu). Tym samym należy uważać, by unikać pętli przy usuwaniu atrybutów, używając do tego tych samych technik — słowników przestrzeni nazw lub wywołań metod z klasy nadzędnej.

Pierwszy przykład

Wszystko to nie jest wcale aż tak skomplikowane, jak mogłoby wynikać z powyższych informacji. By zobaczyć, jak wykorzystać te koncepcje w praktyce, poniżej zamieszczamy ten sam pierwszy przykład, który wykorzystaliśmy dla właściwości i deskryptorów — tym razem zaimplementowany za pomocą metod przeciążania operatorów atrybutów. Ponieważ metody te są tak ogólne, sprawdzamy tutaj nazwy atrybutów, by wiedzieć, że odbywa się dostęp do atrybutu zarządzanego. Pozostałe mogą być przekazywane normalnie.

```
class Person:
    def __init__(self, name):
        self._name = name

    def __getattr__(self, attr):
        if attr == 'name':
            print('pobieranie...')
            return self._name
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):
        if attr == 'name':
            print('modyfikacja...')
            attr = '_name'
        self.__dict__[attr] = value

    def __delattr__(self, attr):
        if attr == 'name':
            print('usunięcie...')
            attr = '_name'
        del self.__dict__[attr]

bob = Person('Robert Zielony')
print(bob.name)
bob.name = 'Robert A. Zielony'
print(bob.name)
del bob.name

print('*'*20)
anna = Person('Anna Czerwona')
print(anna.name)
#print(Person.name.__doc__)

# Dla [Person()]
# Wywołuje __setattr__!
# Dla [obiekt.niezdefiniowany]
# Przechwytcenie name: nie przechowano
# Nie tworzy pętli: prawdziwy atrybut
# Pozostałe są błędami
# Dla [obiekt.dowolny = wartość]
# Ustawienie nazw wewnętrznych
# Tutaj unikanie pętli
# Dla [del obiekt.dowolny]
# Tutaj także unikanie pętli
# jednak jest to o wiele rzadsze
# bob ma zarządzany atrybut
# Wykonuje __getattr__
# Wykonuje __setattr__
# Wykonuje __delattr__
# anna także dziedziczy właściwość
# Tutaj brak odpowiednika
```

Warto zauważyć, że przypisanie atrybutu w konstruktorze `__init__` także wywołuje metodę `__setattr__` — metoda ta przechwytuje wszystkie przypisania atrybutów, nawet te wewnątrz samej klasy. Po wykonaniu kodu zwracany jest ten sam wynik, jednak tym razem jest to efekt normalnego mechanizmu przeciążania operatorów Pythona oraz naszych metod przechwytywania atrybutów:

```
pobieranie...
Robert Zielony
modyfikacja...
pobieranie...
Robert A. Zielony
usunięcie...
-----
pobieranie...
Anna Czerwona
```

Warto również zauważyc, że w przeciwnieństwie do właściwości i deskryptorów nie istnieje bezpośredni sposób podawania tutaj dokumentacji naszego atrybutu — zarządzane atrybuty istnieją wewnątrz kodu metod przechwytyjących, a nie jako odrębne obiekty.

By uzyskać dokładnie taki sam rezultat za pomocą metody `__getattribute__`, należy zastąpić `__getattr__` w przykładzie następującym kodem. Ponieważ przechwytuje on pobrania wszystkich atrybutów, wersja ta musi uważać na unikanie pętli, przekazując nowe operacje pobierania do klasy nadzędnej. Nie może także zakładać, że wszystkie nieznane nazwy są błędami.

Należy zastąpić metodę `__getattr__` za pomocą poniższej

```
def __getattribute__(self, attr):          # Dla [obiekt.dowolny]
    if attr == 'name':                     # Przechwycenie wszystkich nazw
        print('pobieranie...')
        attr = '_name'                    # Odwzorowanie na nazwę wewnętrzna
    return object.__getattribute__(self, attr) # Tutaj unikanie pętli
```

Powyzszy przykład jest odpowiednikiem rozwiązania z właściwościami i deskryptorami, jednak jest nieco sztuczny i tak naprawdę nie przedstawia działania tych narzędzi w praktyce. Metody `__getattr__` oraz `__getattribute__`, ponieważ są tak bardzo ogólne, częściej wykorzystywane są w kodzie opartym na mechanizmie delegacji (zgodnie z informacjami przedstawionymi wcześniej), gdzie dostęp do atrybutów jest sprawdzany i przekierowywany do osadzonego obiektu. Tam, gdzie musimy zarządzać pojedynczym atrybutem, właściwości i deskryptory sprawdzą się równie dobrze lub nawet lepiej.

Obliczanie atrybutów

Tak jak wcześniej, nasz poprzedni przykład nie robi nic specjalnego poza śledzeniem operacji pobrania atrybutów. Obliczenie wartości atrybutu przy jego pobraniu nie stanowi szczególnie większego wyzwania. Tak jak w przypadku właściwości i deskryptorów, poniższy kod tworzy wirtualny atrybut `X`, który po pobraniu wykonuje obliczenia.

```
class AttrSquare:
    def __init__(self, start):
        self.value = start                      # Wywołuje __setattr__!

    def __getattr__(self, attr):                # Przy pobraniu niezdefiniowanego atrybutu
        if attr == 'X':
            return self.value ** 2              # Wartość nie jest niezdefiniowana
        else:
            raise AttributeError(attr)

    def __setattr__(self, attr, value):         # Przy przypisaniu wszystkich atrybutów
        if attr == 'X':
            attr = 'value'
        self.__dict__[attr] = value

A = AttrSquare(3)                           # 2 instancje klasy z przeciążaniem operatorów
B = AttrSquare(32)                         # Każda ma inną informację o stanie

print(A.X)                                 # 3 ** 2
A.X = 4                                    # 4 ** 2
print(A.X)                                 # 32 ** 2
print(B.X)
```

Wykonanie powyższego kodu powoduje zwrócenie tego samego wyniku, jaki otrzymaliśmy wcześniej za pomocą właściwości i deskryptorów, jednak mechanizm tego skryptu oparty jest na metodach przechwytywania ogólnych atrybutów.

9
16
1024

Tak jak wcześniej, ten sam rezultat możemy uzyskać za pomocą użycia metody `__getattribute__` w miejsce `__getattr__`. Poniższy kod zastępuje metodę pobrania za pomocą `__getattribute__` i zmienia metodę przypisania `__setattr__` w taki sposób, by uniknąć pętli, wykorzystując do tego bezpośrednie wywołanie metod klasy nadzędnej zamiast kluczy słownika `__dict__`.

```
class AttrSquare:  
    def __init__(self, start):  
        self.value = start # Wywołuje __setattr__!  
  
    def __getattribute__(self, attr): # Przy pobraniu wszystkich atrybutów  
        if attr == 'X':  
            return self.value ** 2 # Znowu wywołuje __getattribute__!  
        else:  
            return object.__getattribute__(self, attr)  
  
    def __setattr__(self, attr, value): # Przy przypisaniu wszystkich atrybutów  
        if attr == 'X':  
            attr = 'value'  
            object.__setattr__(self, attr, value)
```

Po wykonaniu powyższej wersji kodu otrzymany wynik będzie ten sam. Warto zwrócić uwagę na niejawne przekierowanie, które występuje wewnątrz metod tej klasy:

- `self.value=start` wewnątrz konstruktora wywołuje metodę `__setattr__`,
- `self.value` wewnątrz metody `__getattribute__` wywołuje ponownie metodę `__getattribute__`.

Tak naprawdę metoda `__getattribute__` za każdym razem, gdy pobieramy atrybut `X`, wywoływana jest *dwa* razy. Takie zjawisko nie występuje w wersji z metodą `__getattr__` ponieważ atrybut `value` nie jest niezdefiniowany. Jeśli szybkość kodu ma dla nas znaczenie i chcemy tego uniknąć, należy zmodyfikować metodę `__getattribute__` w taki sposób, by do pobrania `value` również wykorzystywała klasę nadzędną:

```
def __getattribute__(self, attr):  
    if attr == 'X':  
        return object.__getattribute__(self, 'value') ** 2
```

Oczywiście to rozwiązanie nadal powoduje wywołanie metody z klasy nadzędnej, jednak bez dodatkowego wywołania rekurencyjnego, zanim tam dotrzemy. W celu prześledzenia tego, kiedy i jak wywoływanie są te metody, warto dodać wywołania `print`.

Porównanie metod `__getattr__` oraz `__getattribute__`

W celu podsumowania różnic w kodzie metod `__getattr__` oraz `__getattribute__` poniższy przykład wykorzystuje obydwie z nich w celu zaimplementowania trzech atrybutów: `attr1` jest atrybutem klasy, `attr2` jest atrybutem instancji, natomiast `attr3` jest wirtualnym atrybutem zarządzanym, obliczanym przy pobraniu.

```

class GetAttr:
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattr__(self, attr):
        print('pobieranie: ' + attr)
        return 3

X = GetAttr()
print(X.attr1)
print(X.attr2)
print(X.attr3)

print('*'*40)

class GetAttribute(object):           # (object) potrzebne jedynie w wersji 2.6
    attr1 = 1
    def __init__(self):
        self.attr2 = 2
    def __getattribute__(self, attr):      # Dla pobrania wszystkich atrybutów
        print('pobieranie: ' + attr)       # Tutaj użycie klasy nadzędnej w celu uniknięcia pętli
        if attr == 'attr3':
            return 3
        else:
            return object.__getattribute__(self, attr)

X = GetAttribute()
print(X.attr1)
print(X.attr2)
print(X.attr3)

```

Po wykonaniu wersja z metodą `__getattr__` przechwytyuje jedynie dostęp do atrybutu `attr3`, ponieważ jest on niezdefiniowany. Wersja z metodą `__getattribute__` przechwytyuje z kolei operacje pobierania wszystkich atrybutów i musi przekierowywać te, którymi nie zarządza, do klasy nadzędnej w celu uniknięcia pętli.

```

1
2
pobieranie: attr3
3
-----
pobieranie: attr1
1
pobieranie: attr2
2
pobieranie: attr3
3

```

Choć metoda `__getattribute__` może przechwytywać więcej operacji pobierania atrybutów od `__getattr__`, w praktyce często są one stosowane wymiennie — jeśli atrybuty nie są fizycznie przechowywane, obie metody dają ten sam efekt.

Porównanie technik zarządzania atrybutami

W celu podsumowania różnic w kodzie wszystkich czterech zaprezentowanych w niniejszym rozdziale technik zarządzania atrybutami przejdźmy szybko przez bardziej rozbudowany przykład z obliczaniem atrybutów z wykorzystaniem każdej z technik. Poniższa wersja wykorzystuje właściwości do przechwytywania i obliczania atrybutów o nazwie `square` oraz `cube`.

Warto zwrócić uwagę na to, jak ich wartości bazowe zostają przechowane w zmiennych rozpoczęjących się od znaku `_`, tak by nie wchodziły one w konflikt z samymi nazwami właściwości.

Dwa dynamicznie obliczane atrybuty z właściwościami

```
class Powers:
    def __init__(self, square, cube):
        self._square = square                      # _square to wartość bazowa
        self._cube = cube                          # cube to nazwa właściwości

    def getSquare(self):
        return self._square ** 2
    def setSquare(self, value):
        self._square = value
    square = property(getSquare, setSquare)

    def getCube(self):
        return self._cube ** 3
    cube = property(getCube)

X = Powers(3, 4)
print(X.square)                                # 3 ** 2 = 9
print(X(cube))                                 # 4 ** 3 = 64
X.square = 5
print(X.square)                                # 5 ** 2 = 25
```

By uzyskać to samo za pomocą deskryptorów, definiujemy atrybuty za pomocą pełnych klas. Warto zwrócić uwagę na to, że poniższe deskryptory przechowują wartości bazowe jako stan instancji, przez co znowu muszą korzystać z początkowych znaków `_` w celu uniknięcia konfliktu z nazwami deskryptorów. Jak zobaczymy w ostatnim przykładzie rozdziału, moglibyśmy uniknąć konieczności zmiany nazwy, przechowując zamiast tego wartości bazowe w postaci stanu deskryptora.

To samo, ale z wykorzystaniem deskryptorów

```
class DescSquare:
    def __get__(self, instance, owner):
        return instance._square ** 2
    def __set__(self, instance, value):
        instance._square = value

class DescCube:
    def __get__(self, instance, owner):
        return instance._cube ** 3

class Powers:                                     # W wersji 2.6 należy użyć (object)
    square = DescSquare()
    cube = DescCube()
    def __init__(self, square, cube):
        self._square = square                  # "self.square = square" także działa,
        self._cube = cube                     # ponieważ wywołuje metodę __set__ deskryptora!

X = Powers(3, 4)
print(X.square)                                # 3 ** 2 = 9
print(X(cube))                                 # 4 ** 3 = 64
X.square = 5
print(X.square)                                # 5 ** 2 = 25
```

By uzyskać ten sam rezultat za pomocą metody przechwytywania pobierania `__getattr__`, znów musimy przechować wartości bazowe za pomocą zmiennych poprzedzonych znakiem `_`.

tak by operacje dostępu do zarządzanych zmiennych pozostały niezdefiniowane i tym samym wywołyły naszą metodę. Musimy również utworzyć kod metody `__setattr__` w celu przechwycenia operacji przypisania, a także uważać na jej potencjalne zapętlenie.

To samo, ale z ogólnym przechwytywaniem niezdefiniowanego atrybutu `__getattr__`

```
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattr__(self, name):
        if name == 'square':
            return self._square ** 2
        elif name == 'cube':
            return self._cube ** 3
        else:
            raise TypeError('nieznany atrybut:' + name)

    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value
        else:
            self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)                                # 3 ** 2 = 9
print(X(cube))                                 # 4 ** 3 = 64
X.square = 5
print(X.square)                                # 5 ** 2 = 25
```

Ostatnie rozwiązańe — kod wykorzystujący metodę `__getattribute__` — jest podobne do wersji poprzedniej. Ponieważ jednak przechwytyujemy teraz każdy atrybut, musimy przekierować pobrania wartości bazowych do klasy nadzędnej w celu uniknięcia zapętlenia.

To samo, ale z ogólnym przechwytywaniem wszystkich atrybutów za pomocą `__getattribute__`

```
class Powers:
    def __init__(self, square, cube):
        self._square = square
        self._cube = cube

    def __getattribute__(self, name):
        if name == 'square':
            return object.__getattribute__(self, '_square') ** 2
        elif name == 'cube':
            return object.__getattribute__(self, '_cube') ** 3
        else:
            return object.__getattribute__(self, name)

    def __setattr__(self, name, value):
        if name == 'square':
            self.__dict__['_square'] = value
        else:
            self.__dict__[name] = value

X = Powers(3, 4)
print(X.square)                                # 3 ** 2 = 9
print(X(cube))                                 # 4 ** 3 = 64
X.square = 5
print(X.square)                                # 5 ** 2 = 25
```

Jak widać, każda technika przybiera w kodzie inną formę, jednak wszystkie cztery po wykonaniu dają ten sam rezultat:

Więcej informacji na temat porównania tych alternatyw, a także innych opcji, znajdziesz się w bardziej realistycznym ich zastosowaniu w przykładzie ze sprawdzaniem poprawności atrybutów w podrozdziale „Przykład — sprawdzanie poprawności atrybutów”. Najpierw jednak musimy omówić pewną pułapkę związaną z dwoma z powyższych narzędzi.

Przechwytywanie atrybutów wbudowanych operacji

Kiedy wprowadzałem metody `__getattr__` oraz `__getattribute__`, napisałem, że przechwytyują one operacje pobierania atrybutów, odpowiednio, niezdefiniowanych i wszystkich, co sprawia, że idealnie sprawdzają się we wzorcach kodu opartych na delegacji. Choć jest to prawda dla atrybutów o normalnych nazwach, ich działanie zasługuje na dodatkowe wyjaśnienie. W przypadku atrybutów o nazwach metod pobieranych w sposób niejawny przez operacje wbudowane metody te mogą *w ogóle nie być wykonane*. Oznacza to, że wywołania metod przeciążających operatory nie mogą być delegowane do opakowanych obiektów, o ile klasy opakowujące w jakiś sposób same nie zdefiniują tych metod ponownie.

Przykładowo pobranie atrybutów dla metod `__str__`, `__add__` oraz `__getitem__` wykonywane w sposób niejawny za pomocą, odpowiednio, wyświetlania, wyrażeń ze znakiem + oraz indeksowania nie jest przekierowywane do ogólnych metod przechwytywania atrybutów Pythona 3.0. A w szczególności:

- W Pythonie 3.0 dla takich atrybutów nie zostanie wykonana ani metoda `__getattr__`, ani `__getattribute__`.
- W Pythonie 2.6 metoda `__getattr__` jest wykonywana dla takich atrybutów, jeśli są one niezdefiniowane w klasie.
- W Pythonie 2.6 metoda `__getattribute__` jest dostępna jedynie dla klas w nowym stylu i działa tak samo jak w wersji 3.0.

Innymi słowy, w klasach Pythona 3.0 (oraz klasach w nowym stylu z wersji 2.6) nie ma żadnego bezpośredniego sposobu uniwersalnego przechwytywania operacji wbudowanych, takich jak wyświetlanie czy dodawanie. W Pythonie 2.X metody wywoływanie przez takie operacje są wyszukiwane w instancjach w czasie wykonywania, tak jak wszystkie pozostałe atrybuty. W Pythonie 3.0 takie metody są zamiast tego wyszukiwane w *klasach*.

Ta zmiana powoduje, że wzorce kodu oparte na delegacji w Pythonie 3.0 stały się bardziej skomplikowane, ponieważ nie są w stanie w sposób ogólny przechwytywać wywołań metod przeciążania operatorów i przekierować ich do osadzonego obiektu. Nie jest to problem nie do przejścia — klasy opakowujące są w stanie obejść to ograniczenie, redefiniując wszystkie niezbędne metody przeciążania operatorów w samej klasie opakowującej w celu delegowania wywołań. Te dodatkowe metody można dodawać albo ręcznie, za pomocą narzędzi, albo za pomocą definicji we wspólnych klasach nadrzędnych i dziedziczeniu po nich. Takie rozwiązanie sprawia jednak, że klasy opakowujące wymagają więcej pracy niż wcześniej, jeśli metody przeciążania operatorów są częścią interfejsu opakowanego obiektu.

Należy pamiętać, że problem ten występuje jedynie w przypadku metod `__getattr__` oraz `__getattribute__`. Ponieważ właściwości i deskryptory definiowane są jedynie dla określonych atrybutów, nie mają one tak naprawdę w ogóle zastosowania do klas opartych na delegacji.

Pojedynczej właściwości czy deskryptora nie można wykorzystać do przechwytywania dowolnych atrybutów. Co więcej, klasy definiujące zarówno metody przeciążania operatorów, jak i przechwytywanie atrybutów będą działały poprawnie bez względu na typ zdefiniowanego przechwytywania atrybutów. Nasze zaniepokojenie powinny wzbudzać jedynie klasy, które nie mają zdefiniowanych metod przeciążania operatorów, ale próbują je przechwytywać w sposób uniwersalny.

Rozważmy poniższy przykładowy plik *getattr.py*, który sprawdza różne typy atrybutów i operacji wbudowanych na instancjach klas zawierających metody `__getattr__` oraz `__getattribute__`.

```
class GetAttr:  
    eggs = 88  
    def __init__(self):  
        self.spam = 77  
    def __len__(self):  
        print('__len__: 42')  
        return 42  
    def __getattr__(self, attr):  
        print('getattr: ' + attr)  
        if attr == '__str__':  
            return lambda *args: '[GetAttr str]'  
        else:  
            return lambda *args: None  
  
class GetAttribute(object):  
    eggs = 88  
    def __init__(self):  
        self.spam = 77  
    def __len__(self):  
        print('__len__: 42')  
        return 42  
    def __getattribute__(self, attr):  
        print('getattribute: ' + attr)  
        if attr == '__str__':  
            return lambda *args: '[GetAttribute str]'  
        else:  
            return lambda *args: None  
  
for Class in GetAttr, GetAttribute:  
    print('\n' + Class.__name__.ljust(50, '=')  
  
    X = Class()  
    X.eggs  
    X.spam  
    X.other  
    len(X)  
  
    try:  
        X[0]  
    except:  
        print('niepowodzenie []')  
  
    try:  
        X + 99  
    except:  
        print('niepowodzenie +')  
  
    try:  
        X()  
    except:  
        print('niepowodzenie ()')  
  
    print('')  
    # W wersji 2.6 object jest wymagany, w 3.0 — domniemany  
    # W 2.6 wszystkie automatycznie są isinstance(object)  
    # Musi jednak pochodzić od niej, by uzyskać dostęp do narzędzi klas  
    # w nowym stylu,  
    # w tym __getattribute__, niektórych wartości domyślnych __X__
```

```

print('niepowodzenie ()')
X.__call__()                      # __call__? (jawne, nie dziedziczone)

print(X.__str__())                 # __str__? (jawne, dziedziczone po type)
print(X)                          # __str__? (niejawne , za pomocą funkcji wbudowanej)

```

Po wykonaniu w Pythonie 2.6 metoda `__getattribute__` otrzymuje różne niejawne pobrania atrybutów dla operacji wbudowanych, ponieważ Python normalnie szuka takich atrybutów w instancjach. Z kolei metoda `__getattribute__` nie jest wykonywana dla żadnych zmiennych przeciążania operatorów, ponieważ takie zmienne są wyszukiwane jedynie w klasach.

```
C:\misc> c:\python26\python getattr.py

GetAttr=====
getattr: other
__len__: 42
getattr: __getitem__
getattr: __coerce__
getattr: __add__
getattr: __call__
getattr: __call__
getattr: __str__
[Getattr str]
getattr: __str__
[Getattr str]

GetAttribute=====
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattr: __call__
getattr: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025EA1D0>
```

Warto zwrócić uwagę na to, jak metoda `__getattribute__` przechwytuje zarówno niejawne, jak i jawne pobrania `__call__` oraz `__str__` w Pythonie 2.6. Metoda `__getattribute__` nie jest natomiast w stanie przechwycić niejawnych pobrań żadnej z nazw atrybutów dla operacji wbudowanych.

Tak naprawdę przypadek metody `__getattribute__` działa w Pythonie 2.6 tak samo jak w 3.0, ponieważ by korzystać z tej metody, klasy z wersji 2.6 muszą być w nowym stylu przez pochodzenie od `object`. Pochodzenie klasy od `object` w powyższym kodzie jest w wersji 3.0 opcjonalne, ponieważ wszystkie klasy są tam w nowym stylu.

Po wykonaniu w Pythonie 3.0 wyniki dla metody `__getattribute__` różnią się jednak — żadna z wykonanych w sposób niejawnym metod przeciążania operatorów nie wywołuje żadnej z metod przechwytywania atrybutów, kiedy atrybuty te są pobierane za pomocą operacji wbudowanych. Python 3.0 przy wyszukiwaniu takich nazw pomija normalny mechanizm przeszukiwania instancji.

```
C:\misc> c:\python30\python getattr.py

GetAttr=====
getattr: other
__len__: 42
```

```
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattr: __call__
<__main__.GetAttr object at 0x025D17F0>
<__main__.GetAttr object at 0x025D17F0>

GetAttribute=====
getattribute: eggs
getattribute: spam
getattribute: other
__len__: 42
niepowodzenie []
niepowodzenie +
niepowodzenie ()
getattr: __call__
getattr: __str__
[GetAttribute str]
<__main__.GetAttribute object at 0x025D1870>
```

Powyższe rezultaty możemy prześledzić aż do wywołań `print` w skrypcie, by przekonać się, z czego wynikają:

- Próba przechwycenia `__str__` przez metodę `__getattr__` dwukrotnie kończy się niepowodzeniem w wersji 3.0 — raz dla wbudowanego wyświetlania, drugi raz dla jawnego pobrania, ponieważ zachowanie domyślne dziedziczone jest po klasie (a tak naprawdę wbudowanej klasie `object`, będącej klasą nadzczną dla wszystkich).
- Próba przechwycenia `__str__` przez metodę przechwytyującą wszystko `__getattribute__` kończy się niepowodzeniem tylko raz, podczas wbudowanej operacji wyświetlania. Pobranie w sposób jawnym pomija wersję odziedziczoną.
- W obu rozwiązańach w wersji 3.0 próba przechwycenia `__call__` kończy się niepowodzeniem dla wbudowanych wyrażeń wywołania, jednak przechwycenie jest możliwe w obu metodach w przypadku pobrania w sposób jawnym. W przeciwieństwie do `__str__` nie istnieje żadna domyślna odziedziczona metoda `__call__`, która miałaby przewagę nad `__getattr__`.
- Obie klasy przechwytują `__len__` — dlatego, że jest to metoda zdefiniowana w sposób jawnym w samych klasach. Jej nazwa nie jest w Pythonie 3.0 przekierowywana ani do `__getattr__`, ani do `__getattribute__`, jeśli usuniemy metody `__len__` klasy.
- Próba przechwycenia wszystkich pozostałych operacji wbudowanych w wersji 3.0 kończy się niepowodzeniem w obu rozwiązańach.

I znów, w rezultacie metody przeciążania operatorów wykonywane w sposób niejawny przez operacje wbudowane nigdy nie są w Pythonie 3.0 przekierowywane za pośrednictwem żadnej z metod przechwytywania atrybutów. Python 3.0 szuka takich atrybutów w *klasach* i całkowicie pomija wyszukiwanie w instancjach.

Sprawia to, że klasy opakowujące oparte na delegacji tworzy się o wiele trudniej w kodzie napisanym w Pythonie 3.0 — jeśli opakowane klasy mogą zawierać metody przeciążania operatorów, metody te muszą być raz jeszcze zdefiniowane w klasie opakowującej w celu dokonania delegacji do opakowanego obiektu. W uniwersalnych narzędziach służących do delegacji może się to wiązać z dodaniem kilku dodatkowych metod.

Oczywiście dodanie takich metod może być po części zautomatyzowane za pomocą narzędzi rozszerzających klasy za pomocą nowych metod (mogą w tym pomóc dekoratory klas oraz

metaklasy omówione w dwóch kolejnych rozdziałach). Co więcej, klasa nadzędna może być w stanie zdefiniować wszystkie te dodatkowe metody raz, tak by zostały one odziedziczone w klasach opartych na delegacji. Mimo to wzorce programowania oparte na delegacji wymagają w Pythonie 3.0 dodatkowej pracy.

By zobaczyć bardziej realistyczny przykład tego zjawiska wraz z obchodzącym ten problem rozwiązańiem, warto zajrzyć do przykładu z dekoratorem `Private` z kolejnego rozdziału. Zobaczmy tam także, że można wstawić metodę `__getattribute__` w klasie klienta w celu zachowania oryginalnego typu, choć metoda ta nadal nie będzie wywoływana dla metod przejęcia operatorów. Wyświetlanie będzie na przykład wciąż wykonywało metodę `__str__` zdefiniowaną w takiej klasie w sposób bezpośredni, zamiast przekierowywać żądanie za pośrednictwem metody `__getattribute__`.

W kolejnym przykładzie ożywimy ponownie kod z rozdziału o klasach. Skoro wiemy już, jak działa przechwytywanie atrybutów, będziemy w stanie wyjaśnić jego dziwniejsze aspekty.



Przykład wpływu tej zmiany z wersji 3.0 na samego Pythona znajduje się w omówieniu obiektu `os.popen` z tej wersji w rozdziale 14. Ponieważ implementowany jest on za pomocą klasy opakowującej, wykorzystując metodę `__getattr__` do delegacji pobrań atrybutów do osadzonego obiektu, nie przechwytuje w Pythonie 3.0 wbudowanej funkcji iteratora `next(X)`, zdefiniowanego w celu wywołania `__next__`. Przechwytuje jednak i deleguje jawnie wywołania `X.__next__()`, ponieważ nie są one przekierowywane za pośrednictwem funkcji wbudowanej i nie są dziedziczone po klasie nadzędnej, tak jak `__str__`.

Jest to odpowiednik wywołania `__call__` w naszym przykładzie — niejawne wywołania funkcji wbudowanych nie wywołują metody `__getattr__`, jednak wywołania jawnie nazw nieodziedziczonych po typie klasy to robią. Innymi słowy, zmiana ta wpłynęła nie tylko na nasz kod delegujący, ale także na kod z biblioteki standardowej Pythona! Biorąc pod uwagę zakres tej zmiany, możliwe jest, że takie zachowanie może w przyszłości jeszcze ewoluować, dlatego należy to sprawdzić w kolejnych wydaniach Pythona.

Powrót do menedżerów opartych na delegacji

Przykład programowania zorientowanego obiektowo z rozdziału 27. prezentował klasę `Manager` wykorzystującą osadzanie obiektów oraz delegację metod do dostosowania klasy nadzędnej do własnych potrzeb zamiast dziedziczenia. Poniżej znajduje się kod tego przykładu, z usuniętą częścią niepotrzebnego testowania.

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
    def __str__(self):
        return '[Person: %s, %s]' % (self.name, self.pay)

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay) # Osadzenie obiektu Person
    def giveRaise(self, percent, bonus=.10):
```

```

    self.person.giveRaise(percent + bonus)
def __getattr__(self, attr):
    return getattr(self.person, attr)
def __str__(self):
    return str(self.person)

if __name__ == '__main__':
    anna = Person('Anna Czerwona', job='programista', pay=100000)
    print(anna.lastName())
    anna.giveRaise(.10)
    print(anna)
    tom = Manager('Tomasz Czarny', 50000)
    print(tom.lastName())
    tom.giveRaise(.10)
    print(tom)

```

Przechwycenie i delegacja
 # Delegacja wszystkich pozostałych atrybutów
 # Ponownie musi przeciążyć operator (w wersji 3.0)

Manager.__init__
 # Manager.__getattr__ -> Person.lastName
 # Manager.giveRaise -> Person.giveRaise
 # Manager.__str__ -> Person.__str__

Komentarze na końcu pliku pokazują, które metody wywoływanie są w operacji z tego wiersza. W szczególności warto zwrócić uwagę na to, jak wywołania lastName w klasie Manager pozostają niezdefiniowane i tym samym przekierowane do ogólnej metody __getattr__, a stamtąd do osadzonego obiektu Person. Poniżej znajduje się wynik skryptu — obiekt anna otrzymuje podwyżkę dziesięcioprocentową z klasy Person, natomiast obiekt tom dostaje dwudziestoprocentową, ponieważ metoda giveRaise jest zmodyfikowana w klasie Manager.

```
C:\misc> c:\python30\python person.py
Czerwona
[Person: Anna Czerwona, 110000]
Czarny
[Person: Tomasz Czarny, 60000]
```

Zwróćmy jednak uwagę na to, co się stanie, gdy na końcu skryptu *wyświetlimy* obiekt Manager za pomocą wywołania print. Wywołana zostaje wtedy metoda __str__ klasy opakowującej, która wykonuje delegację do metody __str__ obiektu Person. Mając to na uwadze, spójrzmy, co stanie się, jeśli *usuniemy* metodę Manager.__str__ w poniższym kodzie.

```
# Usunięcie metody __str__ klasy Manager

class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay)      # Osadzenie obiektu Person
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)          # Przechwycenie i delegacja
    def __getattr__(self, attr):
        return getattr(self.person, attr)               # Delegacja wszystkich pozostałych atrybutów
```

Teraz w Pythonie 3.0 wyświetlanie *nie* przekierowuje pobrania atrybutu za pośrednictwem ogólnej metody przechwytyjącej w przypadku obiektów klasy Manager. Zamiast tego domyślna metoda wyświetlania __str__ odziedziczona po domniemanej klasie nadzędnej object zostaje odszukana i wykonana (obiekt anna nadal wyświetlany jest poprawnie, ponieważ klasa Person ma jawnie zdefiniowaną metodę __str__).

```
C:\misc> c:\python30\python person.py
Czerwona
[Person: Anna Czerwona, 110000]
Czarny
<__main__.Manager object at 0x02A5AE30>
```

Co ciekawe, takie wykonanie kodu bez metody __str__ *faktycznie* wywołuje w Pythonie 2.6 metodę __getattribute__, ponieważ atrybuty przeciążające operatory są przekierowywane za pośrednictwem tej metody, a klasy nie dziedziczą działania domyślnego metody __str__.

```
C:\misc> c:\python26\python person.py
Czerwona
[Person: Anna Czerwona, 110000]
Czarny
[Person: Tomasz Czarny, 60000]
```

Przełączanie kodu na wykorzystującą metodę `__getattribute__` w Pythonie 3.0 niewiele tutaj pomoże — tak jak `__getattr__`, *nie* jest ona wykonywana dla atrybutów przeciążania operatorów przyjmowanych w sposób niejawny przez wbudowane operacje Pythona 2.6 oraz 3.0.

```
# Zastąpienie __getattr__ za pomocą __getattribute__
```

```
class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay)
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus)
    def __getattribute__(self, attr):
        print('**', attr)
        if attr in ['person', 'giveRaise']:
            return object.__getattribute__(self, attr) # Pobranie moich atrybutów
        else:
            return getattr(self.person, attr) # Delegacja dla wszystkich pozostałych
```

Bez względu na to, której metody przechwytywania atrybutów użyjemy w Pythonie 3.0, nadal musimy ponownie zdefiniować metodę `__str__` w klasie `Manager` (jak powyżej) w celu przechwytywania operacji wyświetlania i przekierowania ich do osadzonego obiektu `Person`.

```
C:\misc> c:\python30\python person.py
Czerwona
[Person: Anna Czerwona, 110000]
** lastName
** person
Czarny
** giveRaise
** person
<__main__.Manager object at 0x028E0590>
```

Warto zwrócić uwagę na to, że `__getattribute__` wywoływanie jest tutaj *dwa* razy dla metod — raz dla nazwy metody, a drugi raz dla pobrania osadzonego obiektu `self.person`. Moglibyśmy tego uniknąć, wykorzystując nieco inny kod, ale nadal musielibyśmy ponownie zdefiniować metodę `__str__` w celu przechwytywania operacji wyświetlania — choć w inny sposób (`self.person` spowodowałoby niepowodzenie dla tej metody `__getattribute__`).

```
# Inny kod __getattribute__ minimalizujący dodatkowe wywołania
```

```
class Manager:
    def __init__(self, name, pay):
        self.person = Person(name, 'manager', pay)
    def __getattribute__(self, attr):
        print('**', attr)
        person = object.__getattribute__(self, 'person')
        if attr == 'giveRaise':
            return lambda percent: person.giveRaise(percent+.10)
        else:
            return getattr(person, attr)
    def __str__(self):
        person = object.__getattribute__(self, 'person')
        return str(person)
```

Po wykonaniu tego rozwiązania alternatywnego nasz obiekt wyświetlany jest poprawnie, jednak tylko dlatego, że dodaliśmy jawną metodę `__str__` w klasie opakowującej — atrybut nadal nie jest przekierowywany do naszej ogólnej metody przechwytywania atrybutów.

```
Czerwona
[Person: Anna Czerwona, 110000]
** lastName
Czarny
** giveRaise
[Person: Tomasz Czarny, 60000]
```

Sytuacja w skrócie wygląda tak, że klasy oparte na delegacji, takie jak `Manager`, muszą w Pythonie 3.0 ponownie definiować niektóre metody przeciążania operatorów (takie jak `__str__`) w celu przekierowania ich do osadzonych obiektów, jednak nie muszą tego robić w wersji 2.6, o ile nie korzysta się z klas w nowym stylu. Jedynym bezpośrednim rozwiązaniem wydaje się użycie metody `__getattr__` i Pythona 2.6 lub ponowne zdefiniowanie metod przeciążających klasy w klasach opakowujących Pythona 3.0.

I znów, nie jest to nieosiągalny cel — wiele klas opakowujących potrafi przewidzieć zbiór wymaganych metod przeciążania operatorów, a narzędzia oraz klasy nadzędne mogą zautomatyzować część tego zadania. Co więcej, nie wszystkie klasy wykorzystują metody przeciążające operatory (tak naprawdę większość klas aplikacji nie powinna tego zazwyczaj robić). Jest to jednak coś, o czym należy pamiętać w przypadku modeli kodu delegacji stosowanych w Pythonie 3.0. Kiedy metody przeciążania operatorów są częścią interfejsu obiektu, klasy opakowujące muszą obsługiwać je w sposób przenośny za pomocą lokalnej redefinicji metod.

Przykład — sprawdzanie poprawności atrybutów

Na zakończenie rozdziału przejdźmy do bardziej realistycznego przykładu, wykorzystującego w kodzie wszystkie cztery rozwiązania z zakresu zarządzania atrybutami. Przykład ten definiuje obiekt `CardHolder` z czterema atrybutami, z których trzy są zarządzane. Zarządzane atrybuty sprawdzają poprawność lub przekształcają dane po pobraniu albo przechowaniu. Wszystkie cztery wersje zwracają te same wyniki dla tego samego kodu testowego, jednak implementują atrybuty na bardzo różne sposoby. Przykłady są tutaj zamieszczone głównie z myślą o samodzielnym przestudiowaniu. Choć nie będę szczegółowo omawiał ich kodu, wykorzystują one koncepcje omawiane już w niniejszym rozdziale.

Wykorzystywanie właściwości do sprawdzania poprawności

Nasz pierwszy przykład do zarządzania trzema atrybutami wykorzystuje właściwości. Jak zwykle moglibyśmy zamiast atrybutów zarządzanych zastosować proste metody, jednak właściwości są pomocne, jeśli w istniejącym kodzie wykorzystywaliśmy już atrybuty. Właściwości wykonują kod automatycznie w momencie dostępu do atrybutów, jednak skoncentrowane są na ścisłe określonym ich zbiorze. Nie można ich wykorzystywać do przechwytywania wszystkich atrybutów w sposób uniwersalny.

By zrozumieć poniższy kod, kluczowe jest zwrócenie uwagi na to, że przypisania do atrybutów wewnętrz metody konstruktora `__init__` także wywołują metodę ustawiającą właściwości. Kiedy metoda ta przypisuje na przykład wartość do `self.name`, automatycznie wywołuje

metodę `setName`, przekształcającą wartość i przypisującą ją do atrybutu instancji o nazwie `_name`, tak by nie wchodził on w konflikt z nazwą właściwości.

Taka zmiana nazwy jest niezbędna, ponieważ właściwości wykorzystują wspólny stan instancji i nie mają własnego. Dane przechowane są w atrybutie o nazwie `_name`, a atrybut o nazwie `name` jest zawsze właściwością, nie danymi.

Podsumowując, poniższa klasa zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na żądanie. Na potrzeby porównania: kod przykładowu oparty na właściwościach składa się z 39 wierszy.

```
class CardHolder:
    acctlen = 8                               # Dane klasy
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Dane instancji
        self.name = name                       # Te także wywołują metody ustawiające właściwości
        self.age = age                          # Zmiana nazwy _X w celu uzyskania nazwy klasy
        self.addr = addr                        # Atrybut addr nie jest zarządzany
                                                # Atrybut remain nie ma danych

    def getName(self):
        return self._name
    def setName(self, value):
        value = value.lower().replace(' ', '_')
        self._name = value
    name = property(getName, setName)

    def getAge(self):
        return self._age
    def setAge(self, value):
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
        else:
            self._age = value
    age = property(getAge, setAge)

    def getAcct(self):
        return self._acct[:-3] + '***'
    def setAcct(self, value):
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('niepoprawny numer konta')
        else:
            self._acct = value
    acct = property(getAcct, setAcct)

    def remainGet(self):
        return self.retireage - self.age          # Mogliby być metodą, nie atrybutem
    remain = property(remainGet)                  # O ile niewykorzystywany jeszcze jako atrybut
```

Kod testu samosprawdzającego

Poniższy kod testuje naszą klasę. Należy go dodać na dole pliku lub umieścić klasę w module i najpierw ją zimportować. Dla wszystkich czterech wersji tego przykładu wykorzystamy ten sam kod sprawdzający. Po wykonaniu tworzymy dwie instancje klasy z zarządzanymi atrybutami, a także pobieramy i modyfikujemy jej różne atrybuty. Operacje, które mogą się nie powieść, opakowujemy w instrukcje `try`.

```

bob = CardHolder('1234-5678', 'Robert Zielony', 40, 'ul. Poziomkowa 15')
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')
bob.name = 'Robert A. Zielony'
bob.age = 50
bob.acct = '23-45-67-89'
print(bob.acct, bob.name, bob.age, bob.remain, bob.addr, sep=' / ')

anna = CardHolder('5678-12-34', 'Anna Czerwona', 35, 'ul. Poziomkowa 16')
print(anna.acct, anna.name, anna.age, anna.remain, anna.addr, sep=' / ')
try:
    anna.age = 200
except:
    print('Niepoprawny wiek dla Anny')

try:
    anna.remain = 5
except:
    print("Nie można ustawić anna.remain")

try:
    anna.acct = '1234567'
except:
    print('Niepoprawne konto dla Anny')

```

Poniżej znajdują się wyniki dla naszego kodu samosprawdzającego. I znów, będą one takie same dla wszystkich wersji tego przykładu. Warto prześledzić kod w celu przekonania się, w jaki sposób wywoływane są metody klasy. Konta wyświetlane są z ukryciem niektórych znaków, imiona i nazwiska przekształcane są na ustandaryzowany format, a czas pozostały do emerytury obliczany jest przy pobraniu za pomocą atrybutu klasy.

```

12345*** / robert_zielony / 40 / 19.5 / ul. Poziomkowa 15
23456*** / robert_a._zielony / 50 / 9.5 / ul. Poziomkowa 15
56781*** / anna_czerwona / 35 / 24.5 / ul. Poziomkowa 16
Niepoprawny wiek dla Anny
Nie można ustawić anna.remain
Niepoprawne konto dla Anny

```

Wykorzystywanie deskryptorów do sprawdzania poprawności

Napiszmy teraz nową wersję kodu naszego przykładu z wykorzystaniem deskryptorów zamiast właściwości. Jak widzieliśmy, deskryptory są bardzo podobne do właściwości, jeśli chodzi o funkcjonalność i pełnione role. Tak naprawdę właściwości są ograniczoną formą deskryptora. Tak jak właściwości, deskryptory zaprojektowano z myślą o obsłudze określonych atrybutów, a nie uniwersalnego dostępu do atrybutów. W przeciwieństwie do właściwości deskryptory mają własny stan i są rozwiązywaniem bardziej ogólnym.

By zrozumieć poniższy kod, istotne jest zauważenie, że przypisania atrybutów wewnętrz metody konstruktora `__init__` wywołują metody `__set__` deskryptora. Kiedy metoda konstruktora przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `Name.__set__(...)`, przekształcającą wartość i przypisującą ją do atrybutu deskryptora o nazwie `name`.

W przeciwieństwie jednak do wariantu opartego na właściwości w tym przypadku wartość `name` dołączana jest do obiektu *deskryptora*, a nie do instancji klasy klienta. Choć moglibyśmy przechować wartość w stanie albo instancji, albo deskryptora, to drugie rozwiązanie jest pozbaione konieczności zmiany nazw z użyciem znaków `_` w celu uniknięcia konfliktów. W klasie klienta `CardHolder` atrybut o nazwie `name` zawsze jest obiektem deskryptora, a nie danymi.

Wreszcie klasa ta implementuje te same atrybuty co wersja poprzednia — zarządza atrybutami o nazwach name, age oraz acct. Pozwala także na bezpośredni dostęp do atrybutu addr i udostępnia atrybut tylko do odczytu o nazwie remain, który jest w pełni wirtualny i obliczany na żądanie. Warto zwrócić uwagę na to, w jaki sposób musimy przechwytywać operacje przypisania do zmiennej remain w deskryptorze i zgłaszać wyjątek. Jak dowiedzieliśmy się wcześniej, gdybyśmy tego nie zrobili, przypisanie do tego atrybutu instancji po cichu utworzyłoby atrybut instancji ukrywający deskryptor atrybutu klasy. Na potrzeby porównania: kod oparty na deskryptorze składa się z 45 wierszy.

```

class CardHolder:
    acctlen = 8                               # Dane klasy
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                      # Dane instancji
        self.name = name                      # Te także wywołują metodę __set__
        self.age = age                        # Zmiana nazwy __X nie jest konieczna: w deskryptorze
        self.addr = addr                      # atrybut addr nie jest zarządzany
                                                # Atrybut remain nie ma danych

class Name:
    def __get__(self, instance, owner):      # Zmienne klasy: lokalne dla CardHolder
        return self.name
    def __set__(self, instance, value):
        value = value.lower().replace(' ', '_')
        self.name = value
name = Name()

class Age:
    def __get__(self, instance, owner):
        return self.age                      # Użycie danych deskryptora
    def __set__(self, instance, value):
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
        else:
            self.age = value
age = Age()

class Acct:
    def __get__(self, instance, owner):
        return self.acct[:-3] + '***'
    def __set__(self, instance, value):
        value = value.replace('-', '')
        if len(value) != instance.acctlen:      # Użycie danych instancji klasy
            raise TypeError('niepoprawny numer konta')
        else:
            self.acct = value
acct = Acct()

class Remain:
    def __get__(self, instance, owner):
        return instance.retireage - instance.age    # Wywołuje Age.__get__
    def __set__(self, instance, value):
        raise TypeError('nie można ustawić remain') # Inaczej zezwolilibyśmy na ustawienie
remain = Remain()

```

Wykorzystywanie metody `__getattr__` do sprawdzania poprawności

Jak widzieliśmy, metoda `__getattr__` przechwytuje wszystkie niezdefiniowane atrybuty, dzięki czemu może działać w sposób bardziej ogólny od właściwości czy deskryptorów. Na potrzeby przykładu sprawdzamy atrybut `name` w celu przekonania się, kiedy atrybut zarządzany jest pobierany. Pozostałe przechowywane są fizycznie w instancji, dzięki czemu nigdy nie docierają do metody `__getattr__`. Choć takie rozwiązanie jest bardziej uniwersalne od zastosowania właściwości czy deskryptorów, wymagana może być dodatkowa praca imitująca koncentrację innych narzędzi na określonych atrybutach. Musimy sprawdzać nazwy w czasie wykonywania, a także napisać kod metody `__setattr__` w celu przechwycenia oraz sprawdzenia poprawności operacji przypisania do atrybutów.

Tak jak w wersjach tego przykładu z właściwością i deskryptorem, kluczowe jest zauważenie, że przypisania atrybutów wewnętrz metody konstruktora `__init__` wywołują także metodę `__setattr__` klasy. Kiedy metoda konstruktora przypisuje na przykład wartość do `self.name`, automatycznie wywołuje metodę `__setattr__`, przekształcającą wartość i przypisującą ją do atrybutu instancji o nazwie `name`. Dzięki przechowaniu `name` w instancji upewniamy się, że przyszłe próby dostępu nie wywołają metody `__getattr__`. Atrybut `acct` jest z kolei przechowany jako `_acct`, tak by późniejsze próby dostępu do `acct` wywoływały metodę `__getattribute__`.

W rezultacie powyższa klasa, jak dwie poprzednie, zarządza atrybutami o nazwach `name`, `age` oraz `acct`. Pozwala także na bezpośredni dostęp do atrybutu `addr` i udostępnia atrybut tylko do odczytu o nazwie `remain`, który jest w pełni wirtualny i obliczany na żądanie.

Na potrzeby porównania: to rozwiązanie składa się z 32 wierszy kodu — 7 mniej od wersji opartej na właściwościach i 13 mniej od wersji wykorzystującej deskryptory. Jasność kodu ma oczywiście większe znaczenie od jego rozmiaru, jednak dodatkowy kod może czasami wiązać się z dodatkową pracą przy programowaniu oraz utrzymywaniu. Ważniejsze są tutaj chyba pełnione *role* — narzędzie uniwersalne, takie jak `__getattr__`, może się lepiej sprawdzać w ogólnej delegacji, natomiast właściwości i deskryptory są zaprojektowane z myślą o zarządzaniu określonymi atrybutami.

Warto również zauważać, że poniższy kod powoduje *dodatkowe wywołania* przy ustawianiu atrybutów niezarządzanych (na przykład `addr`), natomiast żadne dodatkowe wywołania nie występują dla pobierania atrybutów niezarządzanych, gdyż są one zdefiniowane. Choć dla większości programów będzie się to wiązało z niewielkim wzrostem nakładu pracy, właściwości i deskryptory powodują dodatkowe wywołanie jedynie przy dostępie do atrybutów zarządzanych.

Poniżej znajduje się wersja kodu wykorzystująca metodę `__getattr__`.

```
class CardHolder:  
    acctlen = 8                                # Dane klasy  
    retireage = 59.5  
  
    def __init__(self, acct, name, age, addr):  
        self.acct = acct                         # Dane instancji  
        self.name = name                          # Te także wywołują metodę __setattr__  
        self.age = age                            # Zmiana nazwy __acct nie jest konieczna: nazwa jest sprawdzana  
        self.addr = addr                          # Atrybut addr nie jest zarządzany  
                                                # Atrybut remain nie ma danych
```

```

def __getattr__(self, name):
    if name == 'acct':
        return self._acct[:-3] + '***'          # Przy pobraniu niezdefiniowanego atrybutu
                                                # Atrybuty name, age, addr są zdefiniowane
    elif name == 'remain':
        return self.retireage - self.age        # Nie wywołuje __getattr__
    else:
        raise AttributeError(name)

def __setattr__(self, name, value):
    if name == 'name':                      # Dla wszystkich przypisań do atrybutów
        value = value.lower().replace(' ', '_') # Atrybut addr przechowywany w sposób bezpośredni
    elif name == 'age':
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
    elif name == 'acct':
        name = '_acct'
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('niepoprawny numer konta')
    elif name == 'remain':
        raise TypeError('nie można ustawić remain')
    self.__dict__[name] = value              # Uniknięcie zapętlenia

```

Wykorzystywanie metody `__getattribute__` do sprawdzania poprawności

Nasz ostatni wariant wykorzystuje przechwytyującą wszystko metodę `__getattribute__` w celu przechwycenia operacji pobierania atrybutów i zarządzania nimi zgodnie z potrzebami. Przechwycone zostaje każde pobranie atrybutu, dlatego musimy sprawdzać nazwy atrybutów w celu wykrycia tych zarządzanych i przekierować wszystkie pozostałe do klasy nadzędnej, tak by pobranie zostało tam przetworzone w normalny sposób. Ta wersja wykorzystuje do przechwytywania operacji przypisania tę samą metodę `__setattr__` co wariant wcześniejszy.

Poniższy kod działa bardzo podobnie do wersji z metodą `__getattr__`, dlatego nie będę tutaj powtarzał pełnego opisu. Warto jednak zauważyć, że ponieważ każde pobranie atrybutu przekierowywane jest do metody `__getattribute__`, nie musimy zmieniać nazw w celu ich przechwycenia (atrybut `acct` przechowywany zostaje jako `acct`). Z drugiej strony, kod ten musi uwzględniać przekierowanie operacji pobrania niezarządzanych atrybutów do klasy nadzędnej w celu uniknięcia zapętlenia.

Warto również zauważyć, że poniższa wersja powoduje wykonanie dodatkowych wywołań zarówno dla ustawiania, jak i pobierania atrybutów niezarządzanych (na przykład `addr`). Jeśli szybkość działania kodu ma duże znaczenie, to rozwiązanie może być najwolniejsze ze wszystkich. Na potrzeby porównania: poniższa wersja składa się z 32 wierszy kodu, tak samo jak poprzednia.

```

class CardHolder:
    acctlen = 8                                # Dane klasy
    retireage = 59.5

    def __init__(self, acct, name, age, addr):
        self.acct = acct                         # Dane instancji
        self.name = name                          # Te także wywołują metodę __setattr__
        self.age = age                            # Zmiana nazwy __acct nie jest konieczna: nazwa jest sprawdzana
                                                # Atrybut addr nie jest zarządzany
        self.addr = addr                          # Atrybut remain nie ma danych

```

```

def __getattribute__(self, name):
    superget = object.__getattribute__          # Bez pętli — jeden poziom w góre
    if name == 'acct':                         # Dla wszystkich pobrą atrybutów
        return superget(self, 'acct')[:-3] + '***'
    elif name == 'remain':
        return superget(self, 'retireage') - superget(self, 'age')
    else:
        return superget(self, name)             # name, age, addr: przechowane

def __setattr__(self, name, value):
    if name == 'name':                         # Dla wszystkich przypisani atrybutów
        value = value.lower().replace(' ', '_') # Atrybut addr przechowany w sposób bezpośredni
    elif name == 'age':
        if value < 0 or value > 150:
            raise ValueError('niepoprawny wiek')
    elif name == 'acct':
        value = value.replace('-', '')
        if len(value) != self.acctlen:
            raise TypeError('niepoprawny numer konta')
    elif name == 'remain':
        raise TypeError('nie można ustawić remain')
    self.__dict__[name] = value                 # Unikanie pętli, oryginalne nazwy

```

By dowiedzieć się nieco więcej na temat technik zarządzania atrybutami, warto samodzielnie przestudiować i wykonać kody przykładów z tego rozdziału.

Podsumowanie rozdziału

Niniejszy rozdział omawiał różne techniki zarządzania dostępem do atrybutów w Pythonie, w tym metody przeciążania operatorów `__getattr__` oraz `__getattribute__`, właściwości klas oraz deskryptory atrybutów. Porównaliśmy te narzędzia, przedstawiliśmy różnice między nimi oraz zaprezentowaliśmy kilka przypadków użycia demonstrujących ich działanie.

W rozdziale 38. będziemy kontynuowali omawianie elementów służących do budowy narzędzi, przyglądając się *dekoratorom* — kodowi wykonywanemu automatycznie w czasie tworzenia funkcji oraz klas, a nie w momencie dostępu do atrybutów. Zanim jednak przejdziemy dalej, zapoznajmy się ze zbiorem pytań podsumowujących informacje zaprezentowane w tym rozdziale.

Sprawdź swoją wiedzę — quiz

1. Czym różnią się od siebie metody `__getattr__` oraz `__getattribute__`?
2. Czym różnią się od siebie właściwości oraz deskryptory?
3. W jaki sposób są ze sobą powiązane właściwości oraz dekoratory?
4. Jakie są główne różnice w funkcjonalności metod `__getattr__` oraz `__getattribute__` w porównaniu z funkcjonalnością właściwości oraz deskryptorów?
5. Czy całe to porównanie nie jest po prostu rodzajem kłótki?

Sprawdź swoją wiedzę — odpowiedzi

1. Metoda `__getattribute__` wykonywana jest jedynie dla operacji pobierania atrybutów *niezdefiniowanych* — to znaczy tych, które nie są obecne w instancji i nie są dziedziczone po żadnej z jej klas. Metoda `__getattribute__` jest z kolei wywoływana dla *każdej* operacji pobrania atrybutu, bez względu na to, czy atrybut ten jest zdefiniowany, czy też nie. Z tego powodu kod znajdujący się wewnątrz metody `__getattribute__` może swobodnie pobierać inne atrybuty, jeśli są one zdefiniowane, natomiast `__getattribute__` musi do pobrania takich atrybutów wykorzystywać specjalny kod umożliwiający uniknięcie zapętlenia (musi przekierować operacje pobrania do klasy nadrzędnej, tak by pominąć samą siebie).
2. Właściwości pełnią rolę szczegółową, natomiast deskryptory są bardziej ogólne. Właściwości definiują funkcje pobierania, ustawiania i usuwania dla określonego atrybutu. Deskryptory także udostępniają klasę z metodami przeznaczonymi dla tych operacji, jednak dają dodatkową elastyczność umożliwiającą obsługę bardziej dowolnych działań. Tak naprawdę właściwości są prostym sposobem tworzenia szczególnego rodzaju deskryptora — takiego, który wykonuje funkcje w momencie dostępu do atrybutu. Ich kod także się od siebie różni — właściwości tworzy się za pomocą funkcji wbudowanej, natomiast deskryptor za pomocą klasy. Tym samym deskryptory mogą korzystać ze wszystkich zwykłych opcji klas wynikających z programowania zorientowanego obiektywnego, takich jak dziedziczenie. Co więcej, poza informacjami o stanie instancji deskryptory mają własny lokalny stan, dzięki czemu są w stanie unikać konfliktów między nazwami w instancji.
3. Właściwości można tworzyć za pomocą składni dekoratorów. Ponieważ funkcja wbudowana `property` przyjmuje pojedynczy argument funkcji, można ją wykorzystać bezpośrednio jako dekorator funkcji w celu zdefiniowania właściwości dostępu przez pobranie. Z uwagi na ponowne dowiązywanie nazw dekoratorów nazwa udekorowanej funkcji przypisywana jest do właściwości, której akcesor pobierania ustawiany jest na udekorowaną oryginalną funkcję (`name = property(name)`). Atrybuty `setter` oraz `deleter` właściwości pozwalają nam na dodanie akcesorów ustawiania oraz usuwania za pomocą składni dekoratorów — ustawiają one akcesor na udekorowaną funkcję i zwracają rozszerzoną właściwość.
4. Metody `__getattr__` oraz `__getattribute__` są bardziej ogólne — można je wykorzystać do przechwycenia dowolnie wielu atrybutów. Z kolei każda właściwość lub deskryptor umożliwiają przechwytywanie dostępu jedynie dla określonego atrybutu — nie jesteśmy w stanie przechwycić operacji pobrania każdego atrybutu za pomocą pojedynczej właściwości czy deskryptora. Z drugiej strony, właściwości oraz deskryptory z założenia obsługują nie tylko pobieranie atrybutów, ale także *przypisanie*. Metody `__setattr__` oraz `__getattribute__` obsługują jedynie pobieranie. By przechwytywać także przypisania, niezbędne jest utworzenie metody `__setattribute__`. Ich implementacja również się różni — `__getattribute__` i `__getattribute__` są metodami przeciążania operatorów, podczas gdy właściwości i deskryptory są obiektami ręcznie przypisany do atrybutów klas.
5. Wcale nie. Cytując *Latający cyrk Monty Pythona*:
Kłótka to powiązana sekwencja stwierdzeń prowadząca do konkretnej propozycji.
Wcale nie.
Ależ tak! A nie samo zaprzeczanie.

Jeśli się z panem kłóczę, muszę zająć przeciwnie stanowisko.

Ale nie może pan tylko powtarzać: „Wcale nie”.

Ależ tak.

Wcale nie!

Ależ tak!

Wcale nie! Kłótnia to proces intelektualny. Zaprzeczanie to automatyczne przecistawianie się temu, co powie druga osoba.

(krótka przerwa)

Wcale nie.

Ależ tak.

Otoż nie.

Właśnie, że tak...

Dekoratory

W rozdziale poświęconym zaawansowanym aspektom klas (rozdział 31.) poznaliśmy metody statyczne oraz metody klas, a także krótko przyjrzaliśmy się składni dekoratora @, którą Python oferuje w celu zadeklarowania metod tego rodzaju. Z dekoratorami funkcji spotkaliśmy się z kolei w poprzednim rozdziale (rozdział 37.), przy okazji omawiania możliwości wykorzystania funkcji wbudowanej property w roli dekoratora, a także w rozdziale 28., przy omawianiu pojęcia abstrakcyjnych klas nadzędnych.

Niniejszy rozdział stanowi kontynuację zagadnień, na których poprzednio zakończyliśmy omawianie dekoratorów. Tym razem załączymy się w mechanizmy wewnętrzne dekoratorów i omówimy bardziej zaawansowane sposoby tworzenia własnych, nowych dekoratorów. Jak zobaczymy, wiele z koncepcji omawianych we wcześniejszych rozdziałach, takich jak przechowywanie stanu, regularnie pojawia się w połączeniu z dekoratorami.

Zagadnienie to jest stosunkowo zaawansowane, a budowanie dekoratorów zazwyczaj jest bardziej interesujące dla twórców narzędzi niż dla programistów aplikacji. Mimo to, biorąc pod uwagę, że dekoratory są coraz częściej spotykane w popularnych platformach opartych na Pythonie, ich podstawowe zrozumienie może pomóc lepiej pojąć ich rolę, nawet z punktu widzenia użytkownika.

Poza omówieniem szczegółów konstruowania dekoratorów niniejszy rozdział pełni również rolę bardziej realistycznego *studium przypadku* działania Pythona. Ponieważ przykłady są tu nieco bardziej rozbudowane od innych, spotykanych dotychczas w książce, w lepszy sposób pozwalają zilustrować łączenie kodu w bardziej rozbudowane systemy oraz narzędzia. Dodatkową zaletą jest to, że większość kodu pisanego w tym rozdziale będzie można wykorzystać jako narzędzia ogólnego przeznaczenia w programach tworzonych na co dzień.

Czym jest dekorator?

Dekoracja jest sposobem określania kodu zarządzającego dla funkcji oraz klas. Same dekoratory przybierają postać obiektów wywoływalnych (to znaczy funkcji), które przetwarzają inne obiekty wywoływalne. Jak widzieliśmy wcześniej w książce, dekoratory Pythona mają dwie, powiązane ze sobą odmiany:

- *Dekoratory funkcji* wykonują ponowne dowiązania nazw w momencie definicji funkcji, udostępniając warstwę logiki, która jest w stanie zarządzać funkcjami oraz metodami bądź ich późniejszymi wywołaniami.

- Dekoratory klas wykonują ponowne dowiązania nazw w momencie definicji klasy, udostępniając warstwę logiki, która jest w stanie zarządzać klasami bądź instancjami utworzonymi za pomocą ich późniejszego wywołania.

Mówiąc w skrócie, dekoratory udostępniają sposób wstawiania *automatycznie wykonywanego kodu* na końcu instrukcji definicji funkcji oraz klas — na końcu instrukcji `def` w przypadku dekoratorów funkcji oraz na końcu instrukcji `class` w przypadku dekoratorów klas. Taki kod może pełnić wiele różnych ról, opisanych w kolejnych podrozdziałach.

Zarządzanie wywołaniami oraz instancjami

Przykładowo w typowym zastosowaniu taki wykonywany automatycznie kod można wykorzystać do rozszerzenia wywołań funkcji oraz klas. Dzieje się tak dzięki zainstalowaniu *obiektów opakowujących*, które zostaną wywołane później:

- Dekoratory funkcji instalują obiekty opakowujące, przechwytyjące późniejsze *wywołania funkcji* i odpowiednio je przetwarzające.
- Dekoratory klas instalują obiekty opakowujące, przechwytyjące późniejsze *wywołania tworzące instancje* i odpowiednio je przetwarzające.

Dekoratory uzyskują taki efekt dzięki automatycznemu ponownemu dowiązaniu nazw funkcji oraz klas do innych obiektów wywoływalnych na końcu instrukcji `def` oraz `class`. W momencie późniejszego wywołania te obiekty mogą wykonywać zadania, takie jak śledzenie i pomiar czasu dla wywoływanego funkcji czy zarządzanie dostępem do atrybutów instancji klas.

Zarządzanie funkcjami oraz klasami

Choć większość przykładów w niniejszym rozdziale poświęcona jest obiektom opakowującym przechwytyującym późniejsze wywołania funkcji oraz klas, nie jest to jedyna możliwość wykorzystania dekoratorów:

- *Dekoratory funkcji* można także wykorzystać do zarządzania obiektami *funkcji* zamiast ich późniejszymi wywołaniami — na przykład w celu zarejestrowania funkcji w API. W naszym przypadku nacisk położony zostanie na częściej wykorzystywane zastosowanie w opakowywaniu wywołań.
- *Dekoratory klas* można również wykorzystać do bezpośredniego zarządzania obiektami *klas* zamiast wywołaniami tworzącymi instancje — na przykład w celu rozszerzenia klasy za pomocą nowych metod. Ponieważ ta rola w dużej mierze pokrywa się z zastosowaniem *metaklas* (i faktycznie obie metody wykonywane są na końcu procesu tworzenia klasy), dodatkowe przypadki użycia omówimy jeszcze w kolejnym rozdziale.

Innymi słowy, dekoratory funkcji można wykorzystać do zarządzania zarówno wywołaniami funkcji, jak i obiektami funkcji, natomiast dekoratory klas — do zarządzania zarówno instancjami klas, jak i samymi klasami. Dzięki zwracaniu samego udekorowanego obiektu zamiast obiektu opakowującego dekoratory stają się prostym krokiem wykonywanym po utworzeniu funkcji oraz klas.

Bez względu na pełnioną rolę dekoratory udostępniają wygodny i jawnego sposób tworzenia narzędzi przydatnych zarówno w trakcie tworzenia programu, jak i w działających systemach produkcyjnych.

Wykorzystywanie i definiowanie dekoratorów

W zależności od wykonywanych przez nas zadań z dekoratorami możemy spotkać się jako ich użytkownik bądź jako osoba je udostępniająca. Jak widzieliśmy, sam Python zawiera wbudowane dekoratory o wyspecjalizowanych rolach — deklaracje metod statycznych, tworzenie właściwości i wiele innych. Dodatkowo wiele popularnych zestawów narzędzi Pythona zawiera dekoratory wykonujące takie zadania, jak zarządzanie bazą danych czy logiką interfejsu użytkownika. W takich przypadkach możemy sobie poradzić bez wiedzy o tym, w jaki sposób tworzy się kod dekoratora.

W przypadku zadań bardziej ogólnych programiści mogą samodzielnie pisać dowolny kod własnych dekoratorów. Przykładowo dekoratory funkcji można wykorzystać do rozszerzania funkcji za pomocą kodu dodającego śledzenie wywołań, testującego poprawność argumentów w trakcie debugowania, automatycznie tworzącego i zwalniającego blokady wątków czy mierzącego czas wywołań funkcji w celu optymalizacji. Wszystkie działania, jakich dodanie do wywołania funkcji możemy sobie wyobrazić, są kandydatami na własne dekoratory funkcji.

Z drugiej strony, dekoratory funkcji zaprojektowano w taki sposób, by rozszerzały one jedynie określone *wywołanie* funkcji bądź metody, a nie cały *interfejs* obiektu. Lepiej spełniają tę rolę dekoratory klas — ponieważ mogą one przechwytywać wywołania tworzące instancje, można je wykorzystać do implementowania wszelkich zadań związanych z rozszerzaniem interfejsu obiektów czy zarządzaniem. Przykładowo własne dekoratory klas mogą śledzić lub sprawdzać wszystkie referencje do atrybutów wykonywane dla obiektu. Można je także wykorzystywać do tworzenia obiektów pośredniczących (proxy), klas singletona, a także innych popularnych wzorców projektowych. Tak naprawdę niedługo zobaczymy, że wiele z dekoratorów klas jest bardzo podobnych do wzorca projektowego *delegacji*, z którym spotkaliśmy się w rozdziale 30.

Do czego służą dekoratory?

Jak większość zaawansowanych narzędzi Pythona, dekoratory nigdy nie są wymagane i niezbędne — ich funkcjonalność można często zaimplementować za pomocą prostych wywołań funkcji pomocniczych lub innych technik (a na podstawowym poziomie możemy zawsze napisać kod dowiązujący ponownie zmienne w ręczny sposób; dekoratory robią to w sposób automatyczny).

Dekoratory udostępniają jawną składnię przeznaczoną do zadań tego typu, dzięki czemu intencje programisty są jasne, możliwe jest ograniczenie powtarzalności kodu, a także zapewnienie poprawnego użycia API.

- Dekoratory mają bardzo oczywistą, *jawną* składnię, co ułatwia ich dostrzeżenie w porównaniu z wywołaniami funkcji pomocniczych, które mogą być dowolnie oddalone od podmiotowych funkcji bądź klas.
- Dekoratory stosowane są *raz*, przy definicji podmiotowej funkcji bądź klasy. Nie jest konieczne dodawanie dodatkowego kodu (który być może w przyszłości będzie musiał się zmienić) do każdego wywołania klasy bądź funkcji.
- Z uwagi na dwa wcześniejsze punkty dekoratory sprawiają, że mniej prawdopodobne jest to, iż użytkownik API zapomni rozszerzyć funkcję bądź klasę zgodnie z wymaganiami API.

Innymi słowy, poza samym modelem funkcyjonalnym dekoratory mają pewne zalety z punktu widzenia utrzymywania kodu oraz estetyki. Co więcej, jako narzędzia strukturyzujące kod w naturalny sposób powodują one jego *hermetyzację*, co ogranicza powtarzalność i ułatwia wszelkie późniejsze zmiany.

Dekoratory mają jednak również pewne potencjalne *wady* — kiedy wstawiają logikę opakowującą, mogą zmieniać typy dekorowanych obiektów oraz powodować dodatkowe wywołania. Z drugiej strony, te same zastrzeżenia można mieć do wszystkich technik dodających do obiektów logikę opakowującą.

W niniejszym rozdziale omówimy wszystkie te ograniczenia w kontekście prawdziwego kodu. Choć wybór dekoratorów jest nadal kwestią w dużej mierze subiektywną, ich zalety są na tyle kuszące, że w świecie Pythona stają się one dobrą praktyką. By pomóc podjąć decyzję w tej sprawie, przejdziemy teraz do szczegółów.

Podstawy

Zacznijmy od pierwszego przyjrzenia się działaniu dekoratorów z nieco symbolicznej perspektywy. Niedługo będziemy także pisali prawdziwy, działający kod, jednak ponieważ magia dekoratorów sprowadza się do operacji automatycznego ponownego dowiązywania, istotne jest, by najpierw zrozumieć ten typ odwzorowania.

Dekoratory funkcji

Dekoratory funkcji dostępne są w Pythonie od wersji 2.5. Jak widzieliśmy wcześniej w książce, są one w dużej mierze składniowym elementem wykonującym jedną funkcję w drugiej pod koniec instrukcji `def` i dowiązującym oryginalną nazwę funkcji do wyniku.

Zastosowanie

Dekorator funkcji jest rodzajem *deklaracji w czasie wykonywania* i dotyczy funkcji, której definicja pojawi się później. Dekorator zapisywany jest w kodzie w wierszu tuż przed instrukcją `def` definiującą funkcję bądź metodę i składa się z symbolu @, po którym następuje referencja do *metafunkcji* — funkcji (bądź innego obiektu wywoływalnego) zarządzającej inną funkcją.

Z punktu widzenia kodu dekoratory funkcji automatycznie odwzorowują poniższą składnię:

```
@decorator  
def F(arg):  
    ...  
    F(99) # Wywołanie funkcji
```

na jej poniższy odpowiednik, w którym `decorator` jest jednoargumentowym obiektem wywoływalnym zwracającym obiekt wywoływalny o tej samej liczbie argumentów co `F`:

```
def F(arg):  
    ...  
    F = decorator(F) # Ponowne dowiązanie nazwy funkcji do wyniku dekoratora  
    F(99) # Wywołuje decorator(F)(99)
```

Takie automatyczne ponowne dowiązywanie nazw działa dla dowolnej instrukcji `def` — bez względu na to, czy jest to prosta funkcja, czy też metoda wewnętrz klasy. Kiedy później wywoływana jest funkcja `F`, tak naprawdę wywoływany jest obiekt *zwracany* przez dekorator — może to być albo inny obiekt implementujący wymaganą logikę opakowującą, albo oryginalna funkcja.

Innymi słowy, dekoracja odwzorowuje pierwszy z poniższych zapisów na drugi (choć dekorator wykonywany jest tak naprawdę raz, w czasie dekoracji):

```
func(6, 7)
decorator(func)(6, 7)
```

Takie automatyczne ponowne dowiązywanie nazw odpowiada składni metod statycznych oraz dekoracji właściwości, z którymi spotkaliśmy się wcześniej w książce:

```
class C:
    @staticmethod
    def meth(...): ...
                                         # meth = staticmethod(meth)

class C:
    @property
    def name(self): ...
                                         # name = property(name)
```

W obu przypadkach nazwa metody jest na końcu instrukcji `def` ponownie dowiązywana do wyniku wbudowanego dekoratora funkcji. Późniejsze wywołanie oryginalnej nazwy wywołuje obiekt zwrócony przez dekorację.

Implementacja

Sam dekorator jest *obiektem wywoływalnym zwracającym obiekt wywoływalny*. Oznacza to, że zwraca on obiekt, który będzie wywoływany później, kiedy udekorowana funkcja wywoływana jest za pomocą oryginalnej nazwy. Zwracany obiekt jest albo obiektem opakowującym przechwytyującym późniejsze wywołania, albo w jakiś sposób rozszerzoną oryginalną funkcją. Tak naprawdę dekoratory mogą być dowolnymi typami obiektów wywoływalnych i mogą zwracać dowolny typ obiektu wywoływalnego — można wykorzystać dowolną kombinację funkcji oraz klas, choć niektóre z nich lepiej sprawdzają się w pewnych kontekstach.

Przykładowo w celu wejścia do protokołu dekoracji i zarządzania funkcją tuż po jej utworzeniu możemy napisać kod dekoratora o poniższej formie:

```
def decorator(F):
    # Przetworzenie funkcji F
    return F

@decorator
def func(): ...
                                         # func = decorator(func)
```

Ponieważ oryginalna udekorowana funkcja przypisywana jest z powrotem do swojej nazwy, powyższy kod dodaje po prostu krok po utworzeniu do definicji funkcji. Taką strukturę można na przykład wykorzystać do zarejestrowania funkcji w API czy przypisania atrybutów funkcji.

W bardziej typowym zastosowaniu w celu wstawienia logiki przechwytyjącej późniejsze wywołania funkcji możemy napisać kod dekoratora zwracającego obiekt inny od oryginalnej funkcji:

```
def decorator(F):
    # Zapisanie lub użycie funkcji F
    # Zwrócenie innego obiektu wywoływalnego: zagnieżdżonej instrukcji def, klasy z __call__ i tak dalej

@decorator
def func(): ...
                                         # func = decorator(func)
```

Powyższy dekorator wywoływany jest w czasie dekoracji, a zwracany przez niego obiekt wywoływany jest przy późniejszym wywołaniu oryginalnej nazwy funkcji. Sam dekorator otrzymuje udekorowaną funkcję — zwracany obiekt wywoływalny otrzymuje wszelkie argumenty przekazane później do nazwy udekorowanej funkcji. Tak samo działa to w przypadku *metod klas* — domniemany obiekt instancji po prostu pokazuje się w pierwszym argumencie zwracanego obiektu wywoływalnego.

Oto popularny wzorzec kodu ujmujący tę kwestię z punktu widzenia szkieletu. Dekorator zwraca obiekt opakowujący, zachowując oryginalną funkcję w zakresie zawierającym:

```
def decorator(F):
    def wrapper(*args):
        # Użycie F oraz args
        # F(*args) wywołuje oryginalną funkcję
        return wrapper

@decorator
def func(x, y):
    ...
    func(6, 7)                                # 6, 7 przekazywane do *args obiektu opakowującego
```

Kiedy później wywołana zostaje nazwa `func`, tak naprawdę wywoływana jest funkcja opakowująca `wrapper` zwracana przez dekorator `decorator`. Funkcja `wrapper` może wtedy wykonać oryginalną funkcję `func`, ponieważ jest ona nadal dostępna w zakresie zawierającym. W kodzie tego typu każda udekorowana funkcja tworzy nowy zakres zachowujący stan.

By uzyskać to samo z użyciem *klas*, możemy przeciążyć operację wywoływania i skorzystać z atrybutów instancji w miejscu zakresów zawierających.

```
class decorator:
    def __init__(self, func):
        self.func = func
    def __call__(self, *args):
        # Użycie self.func oraz args
        # self.func(*args) wywołuje oryginalną funkcję

@decorator
def func(x, y):
    ...
    func(6, 7)                                # 6, 7 przekazywane do *args dla __call__
```

Gdy w tym przypadku wywołamy później nazwę `func`, tak naprawdę wywołamy metodę przeciążania operatora `__call__` instancji utworzonej za pomocą dekoratora `decorator`. Metoda `__call__` może następnie wywołać oryginalną funkcję `func`, ponieważ jest ona nadal dostępna w *atrybucie instancji*. W kodzie tego typu każda udekorowana funkcja tworzy nową instancję zachowującą stan.

Obsługa dekoracji metod

Jedną istotną kwestią dotyczącą powyższego kodu opartego na klasie jest to, że choć działa on dla przechwytywania prostych wywołań funkcji, nie do końca tak jest w przypadku zastosowania do funkcji *metod klas*.

```
class decorator:
    def __init__(self, func):                  # func to metoda bez instancji
        self.func = func
```

```

def __call__(self, *args):                      # self jest instancją dekoratora
    # self.func(*args) nie działa!
    # Instancja C nie znajduje się w args!

class C:
    @decorator
    def method(self, x, y):                      # method = decorator(method)
        ...                                         # Ponowne dowiązanie do instancji dekoratora

```

W przypadku powyższego kodu udekorowana metoda zostaje ponownie dowiązana do instancji klasy dekoratora zamiast do prostej funkcji.

Problem polega na tym, że `self` w metodzie `__call__` dekoratora otrzymuje przy późniejszym wywołaniu metody instancję dekoratora klasy, natomiast instancja klasy `C` nigdy nie znajdzie się w `*args`. Sprawia to, iż niemożliwe staje się przekazanie wywołania do oryginalnej metody — obiekt dekoratora zachowuje oryginalną funkcję metody, jednak nie ma instancji, która może do niej przekazać.

By obsługiwać zarówno funkcje, jak i metody, lepiej sprawdzi się alternatywa z funkcją zagnieżdzoną:

```

def decorator(F):
    def wrapper(*args):                         # F jest funkcją lub metodą bez instancji
        # F(*args) wykonuje funkcję lub metodę
        return wrapper

    @decorator
    def func(x, y):                           # func = decorator(func)
        ...
        func(6, 7)                            # Tak naprawdę wywołuje wrapper(6, 7)

    class C:
        @decorator
        def method(self, x, y):                 # method = decorator(method)
            ...                                 # Ponowne dowiązanie do prostej funkcji

    X = C()
    X.method(6, 7)                            # Tak naprawdę wywołuje wrapper(X, 6, 7)

```

W przypadku takiego kodu obiekt opakowujący `wrapper` otrzymuje instancję klasy `C` w pierwszym argumentem, dzięki czemu może przekazać go do oryginalnej metody i uzyskać dostęp do informacji o stanie.

Powyższa alternatywa z funkcją zagnieżdzoną działa, ponieważ Python tworzy dowiązany obiekt metody i tym samym przekazuje instancję podmiotowej klasy do argumentu `self` jedynie wtedy, gdy atrybut metody odwołuje się do prostej funkcji. Gdy odwołuje się on natomiast do instancji klasy wywoływalnej, instancja tej klasy przekazywana jest do `self` w celu udostępnienia klasie wywoływalnej jej własnych informacji o stanie. W dalszej części rozdziału zobaczymy, jakie znaczenie ma ta subtelna różnica w praktyce.

Warto również zauważyc, że funkcje zagnieżdzone są chyba najprostszym sposobem obsługi dekoracji zarówno dla funkcji, jak i metod — choć nie są jedynym dostępnym rozwiązaniem. *Deskryptory* z poprzedniego rozdziału otrzymują na przykład przy wywołaniu zarówno deskryptory, jak i instancję podmiotowej klasy. Choć jest to bardziej skomplikowane, w dalszej części rozdziału zobaczymy, w jaki sposób możemy wykorzystać to narzędzie także w tym kontekście.

Dekoratory klas

Dekoratory klas okazały się tak przydatne, że model ten został w Pythonie 2.6 oraz 3.0 rozszerzony, tak by pozwolić na dekorację klas. Dekoratory klas są silnie powiązane z dekoratorami funkcji. Tak naprawdę wykorzystują one tę samą składnię i podobne wzorce kodu. Zamiast opakowywać poszczególne funkcje bądź metody, dekoratory klas są sposobami zarządzania klasami lub opakowywania wywołań tworzących instancje w dodatkową logikę, która zarządza instancjami utworzonymi z klasy bądź je rozszerza.

Zastosowanie

Z punktu widzenia składni dekoratory klas pojawiają się tuż przed instrukcjami `class` (w podobny sposób jak dekoratory funkcji pojawiają się tuż przed definicjami funkcji). Zakładając, że dekorator jest funkcją jednoargumentową zwracającą obiekt wywoływalny, poniższa składnia dekoratora klasy:

```
@decorator          # Udekorowanie klasy
class C:
    ...
x = C(99)           # Utworzenie instancji
```

jest odpowiednikiem poniższego kodu. Klasa jest automatycznie przekazywana do funkcji dekoratora, natomiast wynik dekoratora przypisywany jest z powrotem do nazwy klasy:

```
class C:
    ...
C = decorator(C)      # Ponowne dawianie nazwy klasy do wyniku dekoratora
x = C(99)             # Tak naprawdę wywołuje decorator(C)(99)
```

W rezultacie późniejsze wywołanie nazwy klasy w celu utworzenia instancji wywołuje obiekt zwrócony przez dekorator, a nie oryginalną klasę.

Implementacja

Nowe dekoratory klas tworzy się za pomocą tych samych technik co w przypadku dekoratorów funkcji. Ponieważ dekorator klasy jest także obiektem wywoływalnym zwracającym obiekt wywoływalny, większość kombinacji funkcji i klas będzie w zupełności wystarczająca.

Bez względu na sposób zapisu w kodzie wynikiem dekoratora jest to, co zostaje wykonane przy późniejszym utworzeniu instancji. Przykładowo by po prostu zarządzać klasą tuż po jej utworzeniu, należy zwrócić oryginalną klasę.

```
def decorator(C):
    # Przetworzenie klasy C
    return C

@decorator
class C: ...           # C = decorator(C)
```

By zamiast tego wstawić warstwę opakowującą przechwytyującą późniejsze wywołania tworzące instancje, należy zwrócić inny obiekt wywoływalny.

```
def decorator(C):
    # Zapisanie lub użycie klasy C
    # Zwrócenie innego obiektu wywoływalnego: zagnieżdzonej instrukcji def, klasy z __call__ i tak dalej
```

```
@decorator  
class C: ...
```

C = decorator(C)

Obiekt wywoływalny zwracany przez tego typu dekorator klasy zazwyczaj tworzy i zwraca nową instancję oryginalnej klasy, rozszerzoną w jakiś sposób umożliwiający zarządzanie jej interfejsem. Przykładowo poniższy kod wstawia obiekt przechwytyujący niezdefiniowane atrybuty instancji klasy.

```
def decorator(cls):  
    class Wrapper:  
        def __init__(self, *args):  
            self.wrapped = cls(*args)  
        def __getattr__(self, name):  
            return getattr(self.wrapped, name)  
    return Wrapper  
  
@decorator  
class C:  
    def __init__(self, x, y):  
        self.attr = 'spam'  
  
x = C(6, 7)  
print(x.attr)
```

W momencie dekoracji @

W momencie tworzenia instancji

W momencie pobrania atrybutu

C = decorator(C)

Wykonywane przez Wrapper.__init__

Tak naprawdę wywołuje Wrapper(6, 7)

Wykonuje Wrapper.__getattr__ wyświetla "spam"

W powyższym przykładzie dekorator dowiązuje ponownie nazwę klasy do innej klasy, zachowując oryginalną klasę w zakresie zawierającym i tworząc oraz osadzając instancję oryginalnej klasy przy wywołaniu. Kiedy atrybut jest później pobierany z instancji, jest on przechwytywany przez metodę `__getattr__` obiektu opakowującego i delegowany do osadzonej instancji oryginalnej klasy. Co więcej, każda udekorowana klasa tworzy nowy zakres pamiętający oryginalną klasę. W dalszej części rozdziału przekształcimy ten przykład w bardziej przydatny kod.

Podobnie jak dekoratory funkcji, dekoratory klas są często zapisywane w kodzie jako funkcje fabryczne, tworzące i zwracające obiekty wywoływalne, lub jako klasy wykorzystujące metody `__init__` albo `__call__` do przechwytywania operacji wywoływanego — bądź dowolna kombinacja obu rozwiązań. Funkcje fabryczne zazwyczaj zachowują stan w referencjach do zakresu zawierającego, natomiast klasy — w atrybutach.

Obsługa większej liczby instancji

Tak jak w przypadku dekoratorów funkcji, w dekoratorach klas pewne typy kombinacji działają lepiej od innych. Rozważmy poniższą niepoprawną alternatywę dla dekoratora klasy z poprzedniego przykładu.

```
class Decorator:  
    def __init__(self, C):  
        self.C = C  
    def __call__(self, *args):  
        self.wrapped = self.C(*args)  
        return self  
    def __getattr__(self, attrname):  
        return getattr(self.wrapped, attrname)  
  
@Decorator  
class C: ...  
  
x = C()  
y = C()
```

W momencie dekoracji @

W momencie tworzenia instancji

W momencie pobrania atrybutu

C = Decorator(C)

Nadpisuje x!

Powyższy kod obsługuje kilka udekorowanych klas (każda z nich tworzy nową instancję klasy `Decorator`) i przechwytuje wywołania tworzące instancje (każda wykonuje metodę `__call__`). W przeciwnieństwie do poprzedniej wersji ta nie jest w stanie obsłużyć *więcej liczb instancji* danej klasy — każde wywołanie tworzące instancję nadpisuje poprzednio zapisaną instancję. Wersja oryginalna obsługuje większą liczbę instancji, ponieważ każde wywołanie tworzące instancję tworzy nowy, niezależny obiekt opakowujący. Mówiąc bardziej ogólnie, każdy z poniższych wzorców obsługuje większą liczbę opakowanych instancji:

```
def decorator(C):
    class Wrapper:
        def __init__(self, *args):
            self.wrapped = C(*args)
    return Wrapper

class Wrapper: ...
def decorator(C):
    def onCall(*args):
        return Wrapper(C(*args))
    return onCall
```

Zjawisko to omówimy w bardziej realistycznym kontekście w dalszej części rozdziału. W praktyce musimy uważać na poprawne łączenie typów wywoływalnych, tak by obsługiwały one wyznaczone przez nas zadania.

Zagnieżdżanie dekoratorów

Czasami jeden dekorator nie wystarczy. By obsługiwać kilka kroków rozszerzenia, składnia dekoratorów pozwala na dodawanie większej liczby warstw logiki opakowującej do udekorowanej funkcji lub metody. Przy skorzystaniu z tej możliwości każdy dekorator musi się pojawiać w osobnym wierszu. Następująca postać składni dekoratorów:

```
@A
@B
@C
def f(...):
    ...
```

wykonuje to samo co poniższy kod:

```
def f(...):
    ...
f = A(B(C(f)))
```

W kodzie oryginalna funkcja przekazywana jest przez trzy różne dekoratory, a wynikowy obiekt wywoływalny przypisywany jest z powrotem do oryginalnej nazwy. Każdy dekorator przetwarza wynik poprzedniego, którym może być oryginalna funkcja lub wstawiony obiekt opakowujący.

Jeśli każdy z dekoratorów wstawia obiekty opakowujące, rezultat będzie taki, że przy wywołaniu nazwy oryginalnej funkcji wywołane zostaną trzy różne warstwy logiki obiektów opakowujących, pozwalające na rozszerzenie oryginalnej funkcji na trzy różne sposoby. Ostatni wymieniony dekorator zostanie zastosowany jako pierwszy — jako zagnieżdżony najgłębiej.

Tak samo jak w przypadku funkcji, większa liczba dekoratorów klas daje większą liczbę wywołań zagnieżdżonych funkcji i być może większą liczbę poziomów logiki opakowującej wokół wywołań tworzących instancje. Przykładowo poniższy kod:

```
@spam  
@eggs  
class C:  
    ...  
  
    X = C()
```

jest odpowiednikiem następującego:

```
class C:  
    ...  
    C = spam(eggs(C))  
  
    X = C()
```

I znów, każdy z dekoratorów może zwracać albo oryginalną klasę, albo wstawiony obiekt opakowujący. W przypadku obiektów opakowujących, gdy zostanie zażądana instancja oryginalnej klasy C, wywołanie przekierowywane jest do obiektów warstw opakowujących udostępnianych przez dekoratory spam oraz eggs, które mogą pełnić zupełnie różne role.

Przykładowo poniższe nic nierobiące dekoratory po prostu zwracają udekorowaną funkcję.

```
def d1(F): return F  
def d2(F): return F  
def d3(F): return F  
  
@d1  
@d2  
@d3  
def func():                      # func = d1(d2(d3(func)))  
    print('mielonka')  
  
func()                           # Wyświetla "mielonka"
```

Ta sama składnia działa w przypadku klas, podobnie jak te same nic nierobiące dekoratory.

Kiedy dekoratory wstawiają obiekty funkcji opakowujących, mogą one jednak rozszerzać oryginalne funkcje przy wywołaniu. Poniższy kod dokonuje konkatenacji wyniku w warstwach dekoratorów, w miarę przechodzenia warstw od najbardziej wewnętrznej do zewnętrznej.

```
def d1(F): return lambda: 'X' + F()  
def d2(F): return lambda: 'Y' + F()  
def d3(F): return lambda: 'Z' + F()  
  
@d1  
@d2  
@d3  
def func():                      # func = d1(d2(d3(func)))  
    return 'mielonka'  
  
print(func())                     # Wyświetla "XYZmielonka"
```

Funkcje lambda wykorzystane zostają tutaj do zaimplementowania warstw opakowujących (każda zachowuje opakowaną funkcję w zakresie zawierającym). W praktyce warstwy opakowujące mogą przybierać postać na przykład funkcji czy klas wywoływalnych. Przy dobrym zaprojektowaniu zagieźdzanie pozwala łączyć ze sobą poszczególne etapy rozszerzania na wiele sposobów.

Argumenty dekoratorów

Dekoratory zarówno funkcji, jak i klas zdają się móc przyjmować *argumenty*, choć tak naprawdę argumenty te są przekazywane do obiektu wywoływalnego *zwracającego* z kolei dekorator, który natomiast zwraca obiekt wywoływalny. Poniższy kod:

```
@decorator(A, B)
def F(arg):
    ...
F(99)
```

automatycznie odwzorowywany jest na następującą postać równoważną, gdzie decorator jest obiektem wywoływalnym zwracającym sam dekorator. Zwrócony dekorator zwraca z kolei obiekt wywoływalny wykonywany później dla wywołań nazwy oryginalnej funkcji.

```
def F(arg):
    ...
F = decorator(A, B)(F)      # Ponowne dowiązanie F do wyniku wartości zwracanej przez dekorator
F(99)                      # Tak naprawdę wywołuje decorator(A, B)(F)(99)
```

Argumenty dekoratorów są ustalane przed wystąpieniem dekoracji i zazwyczaj wykorzystywane są do przechowania informacji o stanie do użycia w późniejszych wywołaniach. Funkcja dekoratora z poniższego przykładu może na przykład przybrać następującą postać:

```
def decorator(A, B):
    # Zapisanie lub użycie A, B
    def actualDecorator(F):
        # Zapisanie lub użycie funkcji F
        # Zwrócenie obiektu wywoływalnego: zagnieżdżonej instrukcji def, klasy z __call__ i tak dalej
        return callable
    return actualDecorator
```

W tej strukturze funkcja zewnętrzna zapisuje argumenty dekoratora w postaci informacji o stanie, by były one dostępne do użycia w samym dekoratorze, zwracanym przez niego obiekcie wywoływalnym lub w obu tych miejscach. Powyższy fragment kodu zachowuje argument informacji o stanie w referencjach do zakresu funkcji zawierającej, jednak często wykorzystywane są również atrybuty klas.

Innymi słowy, argumenty dekoratora często wymuszają *trzy poziomy obiektów wywoływalnych*: obiekt przyjmujący argumenty dekoratora zwracający obiekt służący jako dekorator, który z kolei zwraca obiekt obsługujący wywołania do oryginalnej funkcji bądź klasy. Każdy z tych trzech poziomów może być funkcją lub klasą i może zachowywać stan w postaci zakresów lub atrybutów klas. Konkretnie przykłady użycia argumentów dekoratorów zobaczymy w dalszej części rozdziału.

Dekoratory zarządzają także funkcjami oraz klasami

Choć większa część reszty niniejszego rozdziału skupia się na opakowywaniu późniejszych wywołań funkcji oraz klas, powinienem podkreślić, że mechanizm dekoratorów jest o wiele bardziej uniwersalny — jest to protokół służący do przekazywania funkcji oraz klas za pośrednictwem obiektów wywoływalnych natychmiast po ich utworzeniu. Tym samym można go również wykorzystać do wywołania dowolnego przetwarzania po utworzeniu.

```

def decorate(O):
    # Zapisanie lub rozszerzenie funkcji bądź klasy O
    return O

@decorator
def F(): ...                                # F = decorator(F)

@decorator
class C: ...                                # C = decorator(C)

```

Dopóki będziemy zwracać w ten sposób oryginalny udekorowany obiekt, a nie obiekt opakowujący, możemy zarządzać samymi funkcjami i klasami, a nie tylko późniejszymi ich wywołaniami. Bardziej realistyczne przykłady takiego działania zobaczymy w dalszej części rozdziału, gdzie technika ta posłuży do rejestrowania obiektów wywoływalnych w API za pomocą dekoracji oraz przypisywania atrybutów do funkcji w czasie ich tworzenia.

Kod dekoratorów funkcji

Przejdźmy do kodu. W dalszej części niniejszego rozdziału będziemy omawiały działające przykłady demonstrujące zaprezentowane właśnie kwestie związane z dekoratorami. W niniejszym podrozdziale zaprezentujemy kilka przykładów działania dekoratorów funkcji, natomiast w kolejnym — działanie dekoratorów klas. Na koniec zamkniniemy rozdział kilkoma większymi studiami przypadku użycia dekoratorów klas oraz funkcji.

Śledzenie wywołań

Na początek wróćmy do przykładu śledzenia wywołań z rozdziału 31. Poniższy kod definiuje oraz stosuje dekorator funkcji zliczający liczbę wywołań wykonywanych do udekorowanej funkcji, a także wyświetlający komunikat śledzenia dla każdego wywołania.

```

class tracer:
    def __init__(self, func):          # W momencie dekoracji @: zapisanie oryginalnej funkcji
        self.calls = 0
        self.func = func
    def __call__(self, *args):         # W momencie późniejszego wywołania: wykonanie oryginalnej funkcji
        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        self.func(*args)

@tracer
def spam(a, b, c):                  # spam = tracer(spam)
    print(a + b + c)                # Opakowuje spam w obiekt dekoratora

```

Warto przyjrzeć się temu, jak każda funkcja udekorowana za pomocą tej klasy będzie tworzyła nową instancję, z własnym zapisanym obiektem funkcji oraz licznikiem wywołań. Na uwagę zasługuje także to, w jaki sposób składnia argumentów `*args` wykorzystywana jest do spakowania i rozpakowania dowolnej liczby przekazanych argumentów. Taka uniwersalność pozwala na wykorzystanie dekoratora do opakowania dowolnej funkcji z dowolną liczbą argumentów (powyższa wersja nie działa jeszcze na metodach klas, ale w dalszej części podrozdziału to poprawimy).

Jeśli teraz zaimportujemy funkcję z tego modułu i przetestujemy ją w sesji interaktywnej, otrzymamy następujące działanie. Każde wywołanie generuje początkowo komunikat śledzenia,

ponieważ przechwytywane jest przez klasę dekoratora. Poniższy kod może zostać wykonany w Pythonie 2.6 oraz 3.0, podobnie jak pozostałe przykłady z rozdziału, o ile nie zostało zaznaczone inaczej.

```
>>> from decorator1 import spam

>>> spam(1, 2, 3)                                # Tak naprawdę wywołuje opakowującą obiekt śledzenia
wywołanie 1 spam
6

>>> spam('a', 'b', 'c')                         # Wywołuje metodę __call__ w klasie
wywołanie 2 spam
abc

>>> spam.calls                                 # Zliczenie wywołań w informacjach o stanie obiektu opakowującego
2

>>> spam
<decorator1.tracer object at 0x02D9A730>
```

Po wykonaniu klasa tracer zapisuje udekorowaną funkcję i przechwytuje jej późniejsze wywołania w celu dodania warstwy logiki zliczającej i wyświetlającej każde wywołanie. Warto zwrócić uwagę na pokazywanie się całkowitej liczby wywołań jako atrybutu udekorowanej funkcji — spam jest tak naprawdę instancją klasy tracer po udekorowaniu (to odkrycie może rodzić pewne konsekwencje w programach sprawdzających typy, jednak jest stosunkowo nieszkodliwe).

W przypadku wywołań funkcji składnia dekoracji ze znakiem @ może być wygodniejsza od modyfikowania każdego wywołania w celu uzyskania dodatkowego poziomu logiki, gdyż pozwala uniknąć przypadkowego bezpośredniego wywołania oryginalnej funkcji. Rozważmy odpowiednik tego kodu niezawierający dekoratora, jak poniższy:

```
calls = 0
def tracer(func, *args):
    global calls
    calls += 1
    print('wywołanie %s %s' % (calls, func.__name__))
    func(*args)

def spam(a, b, c):
    print(a, b, c)

>>> spam(1, 2, 3)                                # Normalne, nieśledzone wywołanie: przypadkowe?
1 2 3

>>> tracer(spam, 1, 2, 3)                        # Specjalne śledzone wywołanie bez dekoratorów
wywołanie 1 spam
1 2 3
```

Powyższą alternatywę można wykorzystać w dowolnej funkcji bez specjalnej składni ze znakiem @, jednak w przeciwieństwie do wersji z dekoratorem wymaga ona dodania dodatkowej składni w każdym miejscu, w którym w kodzie wywoływana jest funkcja. Co więcej, jej cel nie jest tak oczywisty i brak jest także zapewnienia, że dodatkowa warstwa zostanie wywołana dla normalnych wywołań. Choć dekoratory nigdy nie są *wymagane* (nazwy zawsze możemy dowieźć ponownie ręcznie), często są najwygodniejszą opcją.

Możliwości w zakresie zachowania informacji o stanie

Ostatni przykład zwraca uwagę na pewną istotną kwestię. Dekoratory funkcji mają kilka możliwości w zakresie zachowywania informacji o stanie udostępnianych w czasie dekoracji i wykorzystywanych w czasie samego wywołania funkcji. Zazwyczaj muszą one obsługiwać większą liczbę udekorowanych obiektów oraz wywołań, jednak istnieje kilka sposobów implementacji tych celów — do zachowania stanu można wykorzystać atrybuty instancji, zmienne globalne, zmienne nielokalne, a także atrybuty funkcji.

Atrybuty instancji klasy

Przykładowo poniżej znajduje się rozszerzona wersja poprzedniego przykładu, dodająca obsługę argumentów ze *światami kluczowymi* i *zwracającą* wynik opakowanej funkcji, tak by możliwa była obsługa większej liczby zastosowań.

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)

@tracer
def spam(a, b, c):
    print(a + b + c)
    # To samo co: spam = tracer(spam)
    # Uruchamia tracer.__init__

@tracer
def eggs(x, y):
    print(x ** y)
    # To samo co: eggs = tracer(eggs)
    # Opakowuje eggs w obiekt tracer

spam(1, 2, 3)
spam(a=4, b=5, c=6)
    # Tak naprawdę wywołujeinstancję tracer: wykonuje tracer.__call__
    # spam jest atrybutem instancji

eggs(2, 16)
eggs(4, y=4)
    # Tak naprawdę wywołujeinstancję tracer, self.func to eggs
    # self.calls zliczane tutaj per funkcję (potrzebna instrukcja nonlocal z 3.0)
```

Tak jak wersja oryginalna, powyższy kod wykorzystuje *atrybuty instancji klasy* do zapisania stanu w sposób jawnym. Zarówno opakowana funkcja, jak i licznik wywołań są informacjami *per instancja* — każda dekoracja otrzymuje ich własną kopię. Po wykonaniu skryptu w Pythonie 2.6 lub 3.0 wynik powyższej wersji kodu będzie następujący. Warto zwrócić uwagę na to, że funkcje spam oraz eggs mają własne liczniki wywołań, ponieważ każda dekoracja tworzy nową instancję klasy.

```
wywołanie 1 spam
6
wywołanie 2 spam
15
wywołanie 1 eggs
65536
wywołanie 2 eggs
256
```

Choć przydaje się przy dekoracji funkcji, taki schemat kodu sprawia pewne problemy po zastosowaniu do metod (więcej informacji na ten temat nieco później).

Zakresy zawierające oraz zmienne globalne

Ten sam efekt mogą dawać również referencje do zakresów zawierających instrukcji `def` oraz zagnieżdżone instrukcje `def`, zwłaszcza w przypadku danych statycznych, takich jak udekorowana oryginalna funkcja. W poniższym przykładzie potrzebny był jednak był również licznik w zakresie zawierającym, który zmienia się z każdym wywołaniem, co nie jest możliwe w Pythonie 2.6. W wersji 2.6 możemy albo użyć klas i atrybutów, jak wcześniej, albo przenieść zmienną stanu do *zakresu globalnego* za pomocą deklaracji `global`.

```
calls = 0
def tracer(func):
    def wrapper(*args, **kwargs):
        global calls
        calls += 1
        print('wywołanie %s %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):
    print(a + b + c)

@tracer
def eggs(x, y):
    print(x ** y)

spam(1, 2, 3)                                # Tak naprawdę wywołuje wrapper, dowiązany do func
spam(a=4, b=5, c=6)                            # wrapper wywołuje spam

eggs(2, 16)                                   # Tak naprawdę wywołuje wrapper, dowiązany do eggs
eggs(4, y=4)                                  # Zmienna globalna calls nie jest tutaj liczona per funkcja!
```

Niestety, przeniesienie licznika do wspólnego zakresu globalnego, tak by mógł on być modyfikowany w ten sposób, oznacza również, że będzie on *współdzielony* przez każdą opakowaną funkcję. W przeciwnieństwie do atrybutów instancji klas liczniki globalne są wspólne dla programu, a nie per funkcja — licznik inkrementowany jest z *każdym* wywołaniem śledzonej funkcji. Różnica staje się zauważalna, jeśli porównamy wynik tej wersji z wynikiem poprzedniej — jeden wspólnie, globalny licznik wywołań jest niepoprawnie uaktualniany przez wywołania każdej udekorowanej funkcji.

```
wywołanie 1 spam
6
wywołanie 2 spam
15
wywołanie 3 eggs
65536
wywołanie 4 eggs
256
```

Zakresy funkcji zawierających oraz zmienne nielokalne

Współdzielony stan globalny może w pewnych przypadkach być tym, czego chcemy. Jeśli jednak potrzebny jest nam licznik *per funkcja*, możemy albo użyć klas (jak wcześniej), albo skorzystać z nowej instrukcji Pythona 3.0 o nazwie `nonlocal`, opisanej w rozdziale 17. Ponieważ ta nowa instrukcja pozwala na modyfikowanie zmiennych z zakresu funkcji zawierającej, może służyć jako zmienne dane per dekoracja.

```

def tracer(func):
    calls = 0
    def wrapper(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('wywołanie %s %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return wrapper

@tracer
def spam(a, b, c):
    print(a + b + c)
    # To samo co: spam = tracer(spam)

@tracer
def eggs(x, y):
    print(x ** y)
    # To samo co: eggs = tracer(eggs)

spam(1, 2, 3)
spam(a=4, b=5, c=6)
    # Tak naprawdę wywołuje wrapper, dowiązany do func
    # wrapper wywołuje spam

eggs(2, 16)
eggs(4, y=4)
    # Tak naprawdę wywołuje wrapper, dowiązany do eggs
    # Zmienna nielokalna calls nie jest tutaj per funkcja

```

Teraz, ponieważ zmienne z zakresu funkcji zawierającej nie są globalne dla całego programu, każda opakowana funkcja otrzymuje znowu własny licznik, tak jak to było w przypadku klas i atrybutów. Oto wynik wykonania powyższego kodu w Pythonie 3.0:

```

wywołanie 1 spam
6
wywołanie 2 spam
15
wywołanie 1 eggs
65536
wywołanie 2 eggs
256

```

Atrybuty funkcji

Wreszcie, jeśli nie używamy Pythona 3.X i nie mamy dostępu do instrukcji `nonlocal`, nadal możemy ominąć zmienne globalne oraz klasy, na potrzeby zmiennego stanu wykorzystując *atrybuty funkcji*. W nowszych wersjach Pythona możemy za pomocą zapisu `funkcja.atrybut` przypisywać dowolne atrybuty do funkcji w celu ich dołączania. W naszym przykładzie możemy po prostu na potrzeby zapisania stanu wykorzystać kod `wrapper.calls`. Poniższy kod działa tak samo jak poprzednia wersja ze zmiennymi nielokalnymi, ponieważ licznik znów działa per udekorowana funkcja. Jedyną różnicą jest to, że rozwiązanie to działa również w Pythonie 2.6.

```

def tracer(func):                      # Stan w zakresie funkcji zawierającej i atrybutu funkcji
    def wrapper(*args, **kwargs):       # Zmienna calls jest per funkcja, a nie globalna
        wrapper.calls += 1
        print('wywołanie %s %s' % (wrapper.calls, func.__name__))
        return func(*args, **kwargs)
    wrapper.calls = 0
    return wrapper

```

Warto zwrócić uwagę na to, że rozwiązanie to działa tylko dzięki temu, że nazwa `wrapper` zachowywana jest w zakresie funkcji zawierającej `tracer`. Gdy później inkrementujemy zmienną `wrapper.calls`, nie zmieniamy samej zmiennej `wrapper`, dlatego deklaracja `nonlocal` nie jest wymagana.

Powysze rozwiązanie niemalże trafiło do przypisu, ponieważ jest mniej oczywiste niż deklaracja `nonlocal` z Pythona 3.0 i lepiej jest zostawić je na potrzeby przypadków, w których inne rozwiązania zawodzą. Wykorzystamy je jednak w odpowiedzi do jednego z pytań kończących rozdział, gdzie będziemy musieli uzyskać dostęp do zapisanych informacji o stanie *spoza* kodu dekoratora. Zmienne nielokalne są widoczne jedynie wewnątrz samej funkcji zagnieżdzonej, natomiast atrybuty funkcji mają bardziej rozszerzoną widoczność.

Ponieważ dekoratory często wymuszają kilka poziomów obiektów wywoływalnych, możemy połączyć funkcje z zakresami zawierającymi i klasami z atrybutami w celu uzyskania różnych rodzajów struktur kodu. Jak jednak zobaczymy nieco później, czasami różnice mogą być bardziej subtelne, niż moglibyśmy się tego spodziewać — każda udekorowana funkcja powinna mieć własny stan, a każda udekorowana klasa może wymagać stanu zarówno dla siebie samej, jak i dla każdej wygenerowanej instancji.

Tak naprawdę, jak zobaczymy w kolejnym podrozdziale, jeśli chcemy zastosować dekoratory funkcji także do metod klas, musimy również uważać na rozróżnienie, które Python robi pomiędzy dekoratorami zapisanymi w postaci wywoływalnych obiektów instancji klas a dekoratorami zapisanymi w postaci funkcji.

Uwagi na temat klas I — dekorowanie metod klas

Kiedy napisałem pierwszy dekorator funkcji `tracer` zamieszczony powyżej, naiwnie założyłem, że można go również zastosować do dowolnej *metody* — udekorowane metody powinny działać tak samo, jednak automatyczny argument instancji `self` miałby zostać dołączony na początku `*args`. Niestety, nie miałem racji — po zastosowaniu do metody klasy pierwsza wersja dekoratora `tracer` nie działa, ponieważ `self` jest instancją klasy dekoratora, a instancja udekorowanej klasy podmiotowej nie zostaje dołączona do `*args`. Takie zachowanie występuje zarówno w Pythonie 3.0, jak i 2.6.

Zjawisko to omówiłem pokrótce we wcześniejszej części rozdziału. Teraz jednak zobaczymy je w kontekście realistycznego, działającego kodu. Gdy mamy dekorator śledzący oparty na klasie:

```
class tracer:
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
```

dekoracja prostych funkcji działa zgodnie z wcześniejszym opisem:

```
@tracer
def spam(a, b, c):
    print(a + b + c)
spam(1, 2, 3)
spam(a=4, b=5, c=6)
```

Dekoracja metody klasy nadal jednak nie działa (bardziej czujni Czytelnicy rozpoznają w tym przykładzie klasę `Person` wziętą z omówienia programowania zorientowanego obiektywnego w rozdziale 27.).

```

class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):                      # giveRaise = tracer(giveRaise)
        self.pay *= (1.0 + percent)

    @tracer
    def lastName(self):                             # lastName = tracer(lastName)
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)               # tracer pamięta funkcje metod
bob.giveRaise(.25)                                 # Wykonuje tracer.__call__(???, .25)
print(bob.lastName())                            # Wykonuje tracer.__call__(???)
```

Sedno problemu leży w argumencie `self` metody `__call__` klasy `tracer` — czy jest to instancja klasy `tracer`, czy może instancja klasy `Person`? W obecnej postaci kodu potrzebujemy *obu* — instancji `tracer` dla stanu dekoratora, natomiast instancji `Person` dla przekierowania do oryginalnej metody. Tak naprawdę `self` *musi* jednak być obiektem `tracer`, tak byśmy mogli uzyskać dostęp do informacji o stanie klasy `tracer`. Będzie to prawdziwe bez względu na to, czy dekrujemy prostą funkcję, czy też metodę.

Niestety, gdy nazwa naszej udekorowanej metody zostaje ponownie dowiązana do obiektu instancji klasy za pomocą metody `__call__`, Python przekazuje do `self` jedynie *instancję* `tracer`. Nie przekazuje w liście argumentów elementu `Person`. Co więcej, ponieważ klasa `tracer` nie wie nic o instancji `Person`, którą próbujemy przetwarzającą pomocą wywołań metod, nie ma możliwości utworzenia metody dowiązanej do instancji i tym samym nie da się w sposób poprawny wykonać wywołania.

Tak naprawdę powyższa wersja kodu przekazuje zbyt małą liczbę argumentów do udekorowanej metody i kończy się zwróceniem błędu. Można to zweryfikować, dodając do metody `__call__` dekoratora wiersz wyświetlający wszystkie argumenty. Jak widać, `self` to instancja klasy `tracer`, natomiast instancji klasy `Person` w ogóle nie ma.

```

<__main__.tracer object at 0x02D6AD90> (0.25,) {}
wywołanie 1 giveRaise
Traceback (most recent call last):
  File "C:/misc/tracer.py", line 56, in <module>
    bob.giveRaise(.25)
  File "C:/misc/tracer.py", line 9, in __call__
    return self.func(*args, **kwargs)
TypeError: giveRaise() takes exactly 2 positional arguments (1 given)
```

Jak wspomniano wcześniej, ma to miejsce, gdyż Python przekazuje domniemaną instancję podmiotową do `self`, gdy nazwa metody dowiązywana jest jedynie do prostej funkcji. Kiedy jest to instancja klasy wywoływalnej, zamiast tego przekazywana jest instancja tej klasy. Python tworzy obiekt dowiązanej metody zawierający podmiotową instancję, kiedy metoda jest prostą funkcją.

Wykorzystywanie zagnieżdzonych funkcji do dekoracji metod

Jeśli chcemy, by dekoratory funkcji działały *zarówno* na prostych funkcjach, *jak i* na metodach klas, najprostszym rozwiązaniem jest wykorzystanie jednego z opisanych wcześniej rozwiązań dla zachowywania stanu. Dekorator funkcji należy zapisać w kodzie w postaci zagnieżdzonych

instrukcji `def`, tak byśmy nie musieli polegać na pojedynczym argumencie instancji `self`, który ma być zarówno instancją klasy opakowującej, jak i instancją klasy podmiotowej.

Poniższe rozwiązywanie alternatywne naprawia nasz błąd, wykorzystując do tego zmienne nielokalne z Pythona 3.0. Ponieważ udekorowane metody są ponownie dowiązywane do prostych funkcji zamiast do obiektów instancji, Python w poprawny sposób przekazuje obiekt klasy Person jako pierwszy argument, a dekorator przekazuje go w pierwszym argumencie listy `*args` do argumentu `self` prawdziwych, udekorowanych metod.

Dekorator przeznaczony zarówno dla funkcji, jak i dla metod

```
def tracer(func):                      # Użycie funkcji, a nie klasy z __call__  
    calls = 0                          # Inaczej "self" będzie jedynie instancją dekoratora!  
    def onCall(*args, **kwargs):  
        nonlocal calls  
        calls += 1  
        print('wywołanie %s %s' % (calls, func.__name__))  
        return func(*args, **kwargs)  
    return onCall
```

Ma zastosowanie do prostych funkcji

```
@tracer  
def spam(a, b, c):                  # spam = tracer(spam)  
    print(a + b + c)                 # onCall pamięta spam  
  
spam(1, 2, 3)                      # Wykonuje onCall(1, 2, 3)  
spam(a=4, b=5, c=6)
```

Ma zastosowanie również do funkcji metod klas!

```
class Person:  
    def __init__(self, name, pay):  
        self.name = name  
        self.pay = pay  
  
    @tracer  
    def giveRaise(self, percent):      # giveRaise = tracer(giveRaise)  
        self.pay *= (1.0 + percent)     # onCall pamięta giveRaise  
  
    @tracer  
    def lastName(self):              # lastName = tracer(lastName)  
        return self.name.split()[-1]  
  
print('metody...')  
bob = Person('Robert Zielony', 50000)  
anna = Person('Anna Czerwona', 100000)  
print(bob.name, anna.name)  
anna.giveRaise(.10)                  # Wykonuje onCall(anna, .10)  
print(anna.pay)                    # Wykonuje onCall(bob), lastName w zakresach  
print(bob.lastName(), anna.lastName())
```

Powyższa wersja działa w ten sam sposób zarówno na funkcjach, jak i na metodach.

```
wywołanie 1 spam  
6  
wywołanie 2 spam  
15  
metody...  
Robert Zielony Anna Czerwona  
wywołanie 1 giveRaise  
110000.0
```

```
wywołanie 1 lastName  
wywołanie 2 lastName  
Zielony Czerwona
```

Wykorzystywanie deskryptorów do dekorowania metod

Choć rozwiążanie z zagnieżdżonymi funkcjami przedstawione wyżej jest najprostszym sposobem obsługiwanego dekoratorów mających zastosowanie zarówno do funkcji, jak i do metod klas, możliwe są również inne rozwiązania. Mogą nam tutaj pomóc na przykład deskryptory omówione w poprzednim rozdziale.

Przypomnijmy, że zgodnie z informacjami z poprzedniego rozdziału deskryptor może być atrybutem klasy przypisywanym do obiektów za pomocą metody `__get__` wykonywanej automatycznie wtedy, gdy następuje odwołanie się do atrybutu oraz jego pobranie (w Pythonie 2.6 niezbędne jest pochodzenie od `object`, które nie jest wymagane w wersji 3.0).

```
class Descriptor(object):  
    def __get__(self, instance, owner): ...  
  
class Subject:  
    attr = Descriptor()  
  
X = Subject()  
X.attr # Wykonuje w przybliżeniu Descriptor.__get__(Subject.attr, X, Subject)
```

Deskryptory mogą również zawierać metody dostępu `__set__` oraz `__del__`, których tutaj jednak nie potrzebujemy. Ponieważ metoda `__get__` deskryptora otrzymuje po wywołaniu instancję klasy deskryptora, jak i klasy podmiotowej, dobrze nadaje się do dekorowania metod, kiedy przy wywołaniach potrzebne są nam zarówno stan dekoratora, jak i instancja oryginalnej klasy. Rozważmy następującą alternatywę dla dekoratora śledzącego, będącą deskryptorem:

```
class tracer(object):  
    def __init__(self, func): # W momencie dekoracji @  
        self.calls = 0 # Zapisanie func na potrzeby późniejszego wywołania  
        self.func = func  
    def __call__(self, *args, **kwargs): # W momencie wywołania oryginalnej funkcji  
        self.calls += 1  
        print('wywołanie %s %s' % (self.calls, self.func.__name__))  
        return self.func(*args, **kwargs)  
    def __get__(self, instance, owner): # W momencie pobrania atrybutu metody  
        return wrapper(self, instance)  
  
class wrapper:  
    def __init__(self, desc, subj): # Zapisanie obu instancji  
        self.desc = desc # Przekierowanie wywołań z powrotem do dekoratora  
        self.subj = subj  
    def __call__(self, *args, **kwargs): # Wykonuje tracer.__call__  
        return self.desc(self.subj, *args, **kwargs)  
  
@tracer  
def spam(a, b, c): # spam = tracer(spam)  
    ...to samo co poprzednio... # Wykorzystuje jedynie __call__  
  
class Person:  
    @tracer  
    def giveRaise(self, percent): # giveRaise = tracer(giveRaise)  
        ...to samo co poprzednio... # Sprawia, że giveRaise staje się deskryptorem
```

Takie rozwiązanie działa tak samo jak poprzedni kod funkcji zagnieżdżonej. Udekorowane funkcje wywołują jedynie metodę `__call__`, natomiast udekorowane metody wywołują najpierw

metodę `__get__` w celu przeanalizowania pobrania nazwy metody (dla `instancja.metoda`). Obiekt zwracany przez metodę `__get__` zachowuje instancję klasy podmiotowej i jest następnie wywoływany w celu zakończenia wyrażenia wywołania, uruchamiając metodę `__call__` (dla `(args...)`). Przykładowo poniższe wywołanie kodu testu:

```
anna.giveRaise(.10) # Wykonuje __get__ a następnie __call__
```

wykonuje najpierw metodę `tracer.__get__`, ponieważ atrybut `giveRaise` klasy `Person` został ponownie dowiązany do deskryptora za pomocą dekoratora funkcji. Wyrażenie wywołania uruchamia następnie metodę `__call__` zwracanego obiektu opakowującego, która z kolei wywołuje metodę `tracer.__call__`.

Obiekt `wrapper` zachowuje zarówno instancję deskryptora, jak i instancję podmiotową, dlatego może przekazać sterowanie z powrotem do instancji oryginalnej klasy dekoratora (czy deskryptora). W rezultacie obiekt `wrapper` zapisuje instancję podmiotowej klasy dostępną w trakcie pobrania atrybutu metody i dodaje ją do listy argumentów późniejszego wywołania, przekazywanej do metody `__call__`. Przekierowanie wywołania w ten sposób z powrotem do instancji klasy deskryptora wymagane jest w tej aplikacji, by wszystkie wywołania opakowanej metody wykorzystywały te same informacje o stanie licznika `calls` w obiekcie instancji deskryptora.

Alternatywnie moglibyśmy wykorzystać funkcję zagnieżdzoną i referencje do zakresu funkcji zawierającej w celu uzyskania tego samego efektu. Poniższa wersja działa tak samo jak poprzednia, zamieniając jednak klasę i atrybuty obiektów na funkcję zagnieżdzoną i referencje do zakresu. Wymaga też wyraźnie mniejszej ilości kodu.

```
class tracer(object):
    def __init__(self, func):
        self.calls = 0
        self.func = func
    def __call__(self, *args, **kwargs):
        self.calls += 1
        print('wywołanie %s %s' % (self.calls, self.func.__name__))
        return self.func(*args, **kwargs)
    def __get__(self, instance, owner):
        def wrapper(*args, **kwargs):
            # Zachowanie obu instancji
            return self(instance, *args, **kwargs) # Wykonuje __call__
        return wrapper
```

By prześledzić dwuetapowy proces pobierania i wywoływania, można samodzielnie dodać do metod alternatywnych instrukcję `print` i wykonać je z tym samym kodem testowym co w przypadku pokazanej wcześniej wersji z funkcją zagnieżdzoną. W obu przypadkach rozwiązanie oparte na deskryptorze jest bardziej subtelne od opcji z funkcją zagnieżdzoną, dlatego też będzie raczej drugim wyborem. Ten wzorzec kodu może się jednak przydać w innych kontekstach.

W dalszej części rozdziału będziemy raczej wykorzystywać klasy lub funkcje w celu zapisania w kodzie dekoratorów funkcji, o ile będą one miały zastosowanie jedynie do funkcji. Niektóre dekoratory mogą nie wymagać instancji oryginalnej klasy i nadal będą działały zarówno na funkcjach, jak i metodach, jeśli zapiszemy je w postaci klas. Coś takiego jak własny dekorator `staticmethod` Pythona nie wymaga na przykład instancji klasy podmiotowej (i tak naprawdę jedynym jego celem jest usunięcie instancji z wywołania).

Moral płynący z tej historii jest jednak taki, że jeśli chcemy, by nasze dekoratory działały zarówno na prostych funkcjach, jak i na metodach klas, lepiej będzie wykorzystać przedstawiony tutaj wzorzec kodu oparty na funkcji zagnieżdzonej zamiast klasy z przechwytywaniem wywołań.

Mierzenie czasu wywołania

By w nieco szerszym zakresie wypróbować możliwości dekoratorów funkcji, przejdźmy do innego przypadku użycia. Nasz kolejny dekorator mierzy czas trwania wywołań udekorowanej funkcji — zarówno dla pojedynczego wywołania, jak i całkowity czas dla wszystkich wywołań. Dekorator zostanie zastosowany do dwóch funkcji w celu porównania czasu wymaganego do tworzenia listy składanej oraz wywołania funkcji wbudowanej `map` (dla porównania można również zajrzeć do rozdziału 20., w którym znajduje się kolejny, choć niezawierający dekoratora, przykład mierzący czas dla takich alternatyw iteracyjnych).

```
import time

class timer:
    def __init__(self, func):
        self.func = func
        self.alltime = 0
    def __call__(self, *args, **kargs):
        start = time.clock()
        result = self.func(*args, **kargs)
        elapsed = time.clock() - start
        self.alltime += elapsed
        print('%.2f: %.2f %s' % (self.func.__name__, elapsed, self.alltime))
        return result

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

@timer
def mapcall(N):
    return map(lambda x: x * 2, range(N))

result = listcomp(5)                                # Czas dla tego wywołania, wszystkich wywołań, zwracana wartość
listcomp(50000)
listcomp(500000)
listcomp(1000000)
print(result)
print('allTime = %s' % listcomp.alltime)          # Całkowity czas dla wszystkich wywołań listcomp

print('')
result = mapcall(5)
mapcall(50000)
mapcall(500000)
mapcall(1000000)
print(result)
print('allTime = %s' % mapcall.alltime)            # Całkowity czas dla wszystkich wywołań mapcall

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

W tym przypadku rozwiążanie bez dekoratora pozwalałoby na wykorzystanie funkcji podmiotowych z pomiarem czasu bądź bez niego, jednak skomplikowałoby także sygnaturę wywołania, gdyby pomiar czasu był pożądany (musielibyśmy dodawać kod z każdym wywołaniem zamiast raz w instrukcji `def`). Nie byłoby także bezpośredniego sposobu zagwarantowania, że wszystkie wywołania budujące listę w programie zostałyby przekierowane przez logikę pomiaru czasu — poza próbą odnalezienia wszystkich i potencjalnego zmodyfikowania ich.

Po wykonaniu w Pythonie 2.6 wynik kodu testu samosprawdzającego będzie następujący:

```
listcomp: 0.00002, 0.00002
listcomp: 0.00910, 0.00912
listcomp: 0.09105, 0.10017
listcomp: 0.17605, 0.27622
[0, 2, 4, 6, 8]
allTime = 0.276223304917

mapcall: 0.00003, 0.00003
mapcall: 0.01363, 0.01366
mapcall: 0.13579, 0.14945
mapcall: 0.27648, 0.42593
[0, 2, 4, 6, 8]
allTime = 0.425933533452
map/comp = 1.542
```

I tu pewna subtelność: nie wykonywałem tego kodu w Pythonie 3.0, ponieważ, zgodnie z informacjami z rozdziału 14., funkcja wbudowana `map` zwraca w tej wersji *iterator*, a nie samą listę, jak w wersji 2.6. Tym samym `map` nie da się bezpośrednio porównać z działaniem listy składanej (w obecnej postaci wywołanie funkcji `map` jest w Pythonie 3.0 natychmiastowe).

Jeśli chcemy wykonać ten kod również w Pythonie 3.0, musimy użyć `list(map())` w celu wymuszenia utworzenia listy w podobny sposób jak w liście składanej, gdyż inaczej nie będziemy w stanie porównać tych dwóch rozwiązań. Nie należy tego jednak robić w Pythonie 2.6 — w przeciwnym razie test funkcji `map` zostanie obciążony za utworzenie dwóch list, a nie jednej.

Poniższy kod powinien działać tak samo dla Pythona 2.6 i 3.0. Warto jednak zauważyc, że choć sprawia on, iż porównanie list składanych i funkcji `map` będzie zarówno w wersji 2.6, jak i 3.0 bardziej sprawiedliwe, to ponieważ `range` jest w Pythonie 3.0 także iteratorem, wyników dla obu wersji tego języka nie da się ze sobą bezpośrednio porównać.

```
...
import sys

@timer
def listcomp(N):
    return [x * 2 for x in range(N)]

if sys.version_info[0] == 2:
    @timer
    def mapcall(N):
        return map((lambda x: x * 2), range(N))
else:
    @timer
    def mapcall(N):
        return list(map((lambda x: x * 2), range(N)))
...
```

Wreszcie, jak wiemy z części książki poświęconej modułom, jeśli chcemy móc wykorzystać ten dekorator ponownie w innych modułach, powinniśmy dokonać indentacji kodu testu samosprawdzającego na końcu pliku pod testem `__name__ == '__main__'`, tak by wykonywany był on tylko przy wykonywaniu pliku, a nie jego importowaniu. Nie będziemy jednak tego robić, ponieważ za moment dodamy do naszego kodu dodatkowe opcje.

Dodawanie argumentów dekoratora

Dekorator mierzący czas z poprzedniego podrozdziału działa, ale byłoby miło, gdyby był bardziej konfigurowalny. Podanie etykiety danych wyjściowych czy na przykład włączanie i wyłączanie komunikatów śledzenia mogłyby się przydać w tego typu uniwersalnym narzędziu. W takiej sytuacji przydają się argumenty dekoratora — po poprawnym dodaniu ich do kodu możemy je wykorzystać w celu określenia opcji konfiguracyjnych, które mogą się różnić dla każdej udekorowanej funkcji. Etyketę można na przykład dodać w następujący sposób:

```
def timer(label=''):          # Argumenty przekazane do funkcji
    def decorator(func):      # Funkcja zachowana w zakresie zawierającym
        def onCall(*args):      # Etykietą zachowana w zakresie zawierającym
            ...                 # Zwraca sam dekorator
            print(label, ...)
            return onCall
        return decorator

@timer('==>')
def listcomp(N): ...          # Jak listcomp = timer('==>')(listcomp)
                                # Nazwa listcomp ponownie dowiązana do dekoratora

listcomp(...)                  # Tak naprawdę wywołuje dekorator
```

Powyższy kod dodaje zakres funkcji zawierającej w celu zachowania argumentu dekoratora do zastosowania w późniejszym wywołaniu. Kiedy definiowana jest funkcja `listcomp`, tak naprawdę wywołuje ona funkcję `decorator` (wynik funkcji `timer`, wykonanej zanim nastąpi sama dekoracja) z wartością etykiety `label` dostępną w zakresie funkcji zawierającej. Oznacza to, że funkcja `timer` zwraca dekorator pamiętający zarówno argument dekoratora, jak i oryginalną funkcję oraz zwracający obiekt wywoływalny uruchamiający oryginalną funkcję w przypadku późniejszych wywołań.

Taką strukturę możemy wykorzystać w naszej funkcji mierzącej czas, tak by możliwe było przekazywanie etykiety oraz opcji sterowania śledzeniem w czasie dekoracji. Poniżej znajduje się przykład wykonujący to działanie, zapisany w module `mytools.py`, tak by mógł on być importowany jako uniwersalne narzędzie:

```
import time

def timer(label='', trace=True):          # Dla argumentów dekoratora: zachowanie argumentów
    class Timer:                         # Dla @: zachowanie udekorowanej funkcji
        def __init__(self, func):
            self.func = func
            self.alltime = 0
        def __call__(self, *args, **kargs): # Dla wywołań: wywołanie oryginalnej funkcji
            start = time.clock()
            result = self.func(*args, **kargs)
            elapsed = time.clock() - start
            self.alltime += elapsed
            if trace:
                format = '%s %s: %.5f, %.5f'
                values = (label, self.func.__name__, elapsed, self.alltime)
                print(format % values)
            return result
    return Timer
```

Większość naszej pracy polegała tutaj na osadzeniu oryginalnej klasy `Timer` w funkcji zawierającej w celu utworzenia zakresu zachowującego argumenty dekoratora. Zewnętrzna funkcja `timer` wywoływana jest przed wystąpieniem dekoracji i po prostu zwraca klasę `Timer`, która

sługi jako faktyczny dekorator. W momencie dekoracji tworzona jest instancja klasy `Timer`, pamiętająca samą udekorowaną funkcję i mająca również dostęp do argumentów dekoratora z zakresu funkcji zawierającej.

Tym razem zamiast osadzać kod testu samosprawdzającego w pliku, wykonamy dekorator w innym pliku. Oto klient naszego dekoratora mierzącego czas — plik modułu `testseqs.py`, który ponownie zastosuje dekorator do alternatywnych rozwiązań w zakresie iteracji po sekwencjach.

```
from mytools import timer

@timer(label='[CCC]==>')
def listcomp(N):
    return [x * 2 for x in range(N)]          # Jak listcomp = timer(...)(listcomp)
                                                # listcomp(...) uruchamia Timer.__call__

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return map((lambda x: x * 2), range(N))

for func in (listcomp, mapcall):
    print('')
    result = func(5)                         # Czas dla tego wywołania, wszystkich wywołań, zwracana wartość
    func(50000)
    func(500000)
    func(1000000)
    print(result)
    print('allTime = %s' % func.alltime)     # Całkowity czas dla wszystkich wywołań

print('map/comp = %s' % round(mapcall.alltime / listcomp.alltime, 3))
```

I znów, jeśli chcemy w sprawiedliwy sposób wykonać test w Pythonie 3.0, musimy opakować funkcję `map` w wywołanie `list`. Po wykonaniu w obecnej postaci w Pythonie 2.6 powyższy plik wyświetla następujące wyniki. Każda udekorowana funkcja ma teraz własną etykietę, zdefiniowaną za pomocą argumentów dekoratora.

```
[CCC]==> listcomp: 0.00003, 0.00003
[CCC]==> listcomp: 0.00640, 0.00643
[CCC]==> listcomp: 0.08687, 0.09330
[CCC]==> listcomp: 0.17911, 0.27241
[0, 2, 4, 6, 8]
allTime = 0.272407666337

[MMM]==> mapcall: 0.00004, 0.00004
[MMM]==> mapcall: 0.01340, 0.01343
[MMM]==> mapcall: 0.13907, 0.15250
[MMM]==> mapcall: 0.27907, 0.43157
[0, 2, 4, 6, 8]
allTime = 0.431572169089
map/comp = 1.584
```

Jak zawsze, możemy także przetestować to w sesji interaktywnej w celu przekonania się, w jaki sposób działają argumenty konfiguracyjne.

```
>>> from mytools import timer
>>> @timer(trace=False)                      # Brak śledzenia, zebranie całkowitego czasu
...   def listcomp(N):
...       return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
>>> x = listcomp(5000)
>>> x = listcomp(5000)
```

```

>>> listcomp
<mytools.Timer instance at 0x025C77B0>
>>> listcomp.alltime
0.0051938863738243413

>>> @timer(trace=True, label='\t=>')                      # Załączenie śledzenia
... def listcomp(N):
...     return [x * 2 for x in range(N)]
...
>>> x = listcomp(5000)
=> listcomp: 0.00155, 0.00155
>>> x = listcomp(5000)
=> listcomp: 0.00156, 0.00311
>>> x = listcomp(5000)
=> listcomp: 0.00174, 0.00486
>>> listcomp.alltime
0.0048562736325408196

```

Powyższy dekorator funkcji mierzący czas można wykorzystać dla dowolnej funkcji, zarówno w modułach, jak i w sesji interaktywnej. Innymi słowy, automatycznie można go zakwalifikować jako *narzędzie ogólnego przeznaczenia* służące do pomiaru czasu wykonywania kodu w skryptach. Następny przykład argumentów dekoratora znajduje się w podrozdziale „Implementacja atrybutów prywatnych”, natomiast jeszcze kolejny — w podrozdziale „Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych”.



Metody pomiaru czasu: Dekorator `timer` z niniejszego podrozdziału działa na wszystkich funkcjach, jednak w celu zastosowania go również do *metod klas* wymagana jest niewielka modyfikacja. Mówiąc w skrócie, jak wspomnieliśmy we wcześniejszym podrozdziale zatytułowanym „Uwagi na temat klas I — dekorowanie metod klas”, należy unikać wykorzystywania zagnieżdżonej klasy. Ponieważ jednak taka zmiana będzie tematem pytań kończących rozdział, całkowicie pominę tutaj podanie pełnego rozwiązania.

Kod dekoratorów klas

Dotychczas pisaliśmy kod dekoratorów funkcji zarządzających wywołaniami funkcji, jednak, jak widzieliśmy, w Pythonie 2.6 oraz 3.0 dekoratory zostały rozszerzone w taki sposób, by działać również na klasach. Zgodnie z wcześniejszym opisem, choć są one podobne do dekoratorów funkcji, dekoratory klas stosowane są zamiast tego do klas — można je wykorzystać albo do zarządzania samymi klasami, albo do przechwytywania wywołań tworzących instancje w celu zarządzania instancjami. Podobnie do dekoratorów funkcji, dekoratory klas są tak naprawdę składnią opcjonalną, choć wiele osób uważa, że pozwalają one uczynić intencje programisty bardziej oczywistymi, a także minimalizują liczbę błędnych wywołań.

Klasy singletona

Ponieważ dekoratory klas mogą przechwytywać wywołania tworzące instancje, można je wykorzystać albo do zarządzania wszystkimi instancjami klasy, albo do rozszerzenia interfejsów tych instancji. By to zademonstrować, poniżej znajduje się przykład dekoratora klasy, który wykonuje to pierwsze zadanie — zarządzania instancjami klasy. Kod ten implementuje klasyczny wzorzec projektowy *singletona*, w którym istnieje maksymalnie jedna instancja klasy.

Funkcja singleton definiuje i zwraca funkcję zarządzającą instancjami, natomiast składnia ze znakiem @ automatycznie opakowuje w tę funkcję klasę podmiotową.

```
instances = {}
def getInstance(aClass, *args):
    if aClass not in instances:
        instances[aClass] = aClass(*args)
    return instances[aClass]

def singleton(aClass):
    def onCall(*args):
        return getInstance(aClass, *args)
    return onCall
```

Zarządzanie globalną tabelą
Dodanie **args dla słów kluczowych
Jeden wpis słownika na klasę

W momencie dekoracji @
W momencie tworzenia instancji

By wykorzystać powyższy kod, należy udekorować klasy, dla których chcemy wymusić model z pojedynczą instancją.

```
@singleton
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

@singleton
class Spam:
    def __init__(self, val):
        self.attr = val

bob = Person('Robert', 40, 10)
print(bob.name, bob.pay())                                     # Tak naprawdę wywołuje onCall

anna = Person('Anna', 50, 20)
print(anna.name, anna.pay())                                    # Ten sam pojedynczy obiekt

X = Spam(42)
Y = Spam(99)
print(X.attr, Y.attr)                                         # Jeden obiekt Person, jeden obiekt Spam
```

Person = singleton(Person)
Ponownie dowiązuje Person do onCall
onCall pamięta Person

Spam = singleton(Spam)
Ponownie dowiązuje Spam do onCall
onCall pamięta Spam

Tak naprawdę wywołuje onCall

Ten sam pojedynczy obiekt

Jeden obiekt Person, jeden obiekt Spam

Gdy klasa Person bądź Spam zostaje później wykorzystana do utworzenia instancji, warstwa logiki opakowującej udostępniana przez dekorator przekierowuje wywołania tworzące instancję do funkcji onCall, która z kolei wywołuje funkcję getInstance zarządzającą pojedynczą instancją na klasę i ją współdzielącą, bez względu na to, ile wywołań tworzących zostanie wykonanych. Oto wynik powyższego kodu:

```
Robert 400
Robert 400
42 42
```

Co ciekawe, możemy także wykorzystać rozwiązanie alternatywne, jeśli jesteśmy w stanie użyć instrukcji nonlocal (dostępnej w Pythonie 3.0 oraz wersjach nowszych) do modyfikacji zmiennych z zakresu zawierającego, zgodnie z wcześniejszym opisem. Poniższa alternatywa daje identyczny wynik, wykorzystując tylko jeden zakres funkcji zawierającej na klasę zamiast jednego wpisu do globalnej tabeli na klasę.

```
def singleton(aClass):
    instance = None
    def onCall(*args):
        nonlocal instance
        if instance == None:
```

W momencie dekoracji @
W momencie tworzenia instancji
Instrukcja nonlocal z wersji 3.0 i nowszych

```

    instance = aClass(*args)      # Jeden zakres na klasę
    return instance
    return onCall

```

Powyzsza wersja działa tak samo, jednak nie jest uzależniona od nazw z zakresu globalnego poza dekoratorem. W Pythonie 2.6 oraz 3.0 możemy także napisać takie samodzielne rozwiązanie za pomocą klasy. Poniższa propozycja wykorzystuje jedną *instancję* na klasę zamiast zakresu funkcji zawierającej czy tabeli globalnej i działa tak samo jak dwie pozostałe wersje. Tak naprawdę opiera się ona na tym samym wzorcu projektowym, który później zobaczymy w często spotykanym rozwiązaniu związanym z dekoratorem klasy. Tutaj *potrzebna* jest nam tylko jedna instancja, jednak nie zawsze tak jest.

```

class singleton:
    def __init__(self, aClass):
        self.aClass = aClass
        self.instance = None
    def __call__(self, *args):
        if self.instance == None:
            self.instance = self.aClass(*args)      # Jedna instancja na klasę
        return self.instance

```

By uczynić z tego dekoratora prawdziwe narzędzie ogólnego przeznaczenia, należy przechować go w pliku modułu, który można importować, a także dokonać indentacji kodu testu samo-sprawdzającego pod sprawdzeniem `__name__` i dodać obsługę argumentów ze słowami kluczowymi w wywołaniach tworzących instancje za pomocą składni `**kwargs` (pozostawię to jako sugerowane ćwiczenie).

Śledzenie interfejsów obiektów

Przykład z singletonem z poprzedniego podrozdziału ilustrował użycie dekoratorów klas do zarządzania *wszystkimi* instancjami klasy. Kolejny często stosowany przypadek użycia dekoratorów klas rozszerza interfejs *każdej* z wygenerowanych instancji. Dekoratory klas mogą właściwie instalować na instancjach warstwę logiki opakowującej zarządzającą w jakiś sposób dostępem do ich interfejsów.

Przykładowo w rozdziale 30. metoda przeciążania operatora `__getattr__` zaprezentowana jest jako sposób opakowania całych interfejsów obiektów osadzonych instancji w celu zaimplementowania wzorca projektowego *delegacji*. Podobne przykłady widzieliśmy również w omówieniu zarządzanych atrybutów w poprzednim rozdziale. Warto przypomnieć, że metoda `__getattr__` jest wykonywana przy pobraniu niezdefiniowanej nazwy atrybutu. Ten punkt zaczepienia możemy wykorzystać do przechwytywania wywołań metod w klasie kontrolera i przekazywania ich do osadzonego obiektu.

Jako punkt odniesienia poniżej znajduje się oryginalny przykład delegacji niezawierający dekoratora, działający na dwóch obiektach typów wbudowanych.

```

class Wrapper:
    def __init__(self, object):
        self.wrapped = object
    def __getattr__(self, attrname):
        print('Śledzenie:', attrname)
        return getattr(self.wrapped, attrname)

```

```

>>> x = Wrapper([1,2,3])          # Opakowanie listy
>>> x.append(4)                 # Wydelegowanie do metody listy
Śledzenie: append

```

```

>>> x.wrapped          # Wyświetlenie mojej składowej
[1, 2, 3, 4]

>>> x = Wrapper({"a": 1, "b": 2})      # Opakowanie słownika
>>> list(x.keys())                   # Wydelegowanie do metody słownika
Śledzenie: keys                      # W Pythonie 3.0 należy użyć list()
['a', 'b']

```

W powyższym kodzie klasa `Wrapper` przechwytuje próby dostępu do każdego z atrybutów opakowanego obiektu, wyświetla komunikat śledzenia i wykorzystuje funkcję wbudowaną `getattr` do przekazania żądania do opakowanego obiektu. W szczególności śledzi ona próby dostępu do atrybutów wykonywane poza klasą opakowanego obiektu. Próby dostępu wewnętrz metod opakowanego obiektu nie są przechwytywane i są normalnie wykonywane. Cały ten model interfejsu różni się od zachowania dekoratorów funkcji, które opakowują tylko jedną, określona metodę.

Dekoratory klas udostępniają alternatywny i wygodny sposób zapisu w kodzie techniki `__getattribute__` służącej do opakowania całego interfejsu. W Pythonie 2.6 oraz 3.0 poprzedni przykład z klasą można zapisać w postaci dekoratora klasy uruchamiającego tworzenie opakowanej instancji, zamiast przekazywać przygotowaną wcześniej instancję do konstruktora obiektu opakowującego (rozszerzonego tutaj również w celu obsługiwanego argumentów ze słowami kluczowymi w `**kwargs`, a także zliczania liczby wykonanych prób dostępu).

```

def Tracer(aClass):                                # W momencie dekoracji @
    class Wrapper:
        def __init__(self, *args, **kwargs):       # W momencie tworzenia instancji
            self.fetches = 0
            self.wrapped = aClass(*args, **kwargs)   # Użycie nazwy z zakresu funkcji zawierającej
        def __getattribute__(self, attrname):        # Przechwytuje wszystko oprócz własnych atrybutów
            print('Śledzenie: ' + attrname)
            self.fetches += 1
            return getattr(self.wrapped, attrname)  # Delegacja do opakowanego obiektu
    return Wrapper

@Tracer
class Spam:
    def display(self):
        print('Mielonka!' * 8)

    # Spam = Tracer(Spam)
    # Klasa Spam ponownie dowiązana do Wrapper

@Tracer
class Person:
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

    # Person = Tracer(Person)
    # Wrapper pamięta Person

food = Spam()                                     # Wywołuje Wrapper()
food.display()                                    # Wywołuje __getattribute__
print([food.fetches])                            # Wywołuje __getattribute__

bob = Person('Robert', 40, 50)                   # Obiekt bob jest tak naprawdę instancją Wrapper
print(bob.name)                                  # Wrapper osadza instancję Person
print(bob.pay())

print('')
anna = Person('Anna', rate=100, hours=60)      # Obiekt anna jest inną instancją Wrapper
print(anna.name)                                 # z inną klasą Person

```

```

print(anna.pay())
print(bob.name)                                # Obiekt bob ma inny stan
print(bob.pay())
print([bob.fetches, anna.fetches])               # Atrybuty klasy Wrapper nie są śledzone

```

Istotne jest, by zwrócić uwagę na to, jak bardzo kod ten różni się od dekoratora śledzącego, z którym spotkaliśmy się wcześniej. W podrozdziale zatytułowanym „Kod dekoratorów funkcji” przygląдаliśmy się dekoratorom pozwalającym na śledzenie i pomiar czasu wywołań określonej funkcji lub metody. W przeciwnieństwie do nich dzięki przechwytywaniu wywołań tworzących instancje dekorator klasy pozwala nam śledzić pełny interfejs obiektu — czyli próby dostępu do dowolnego z jego atrybutów.

Poniżej znajdują się dane wyjściowe zwracane przez powyższy kod w wersjach 2.6 oraz 3.0. Próby pobrania atrybutów instancji zarówno klasy Spam, jak i Person wywołują logikę metody `__getattribute__` klasy Wrapper, ponieważ obiekty food oraz bob są tak naprawdęinstancjami klasy Wrapper dzięki przekierowaniu wywołań tworzących instancje przez dekorator.

```

Śledzenie: display
Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!Mielonka!
[1]
Śledzenie: name
Robert
Śledzenie: pay
2000

Śledzenie: name
Anna
Śledzenie: pay
6000
Śledzenie: name
Robert
Śledzenie: pay
2000
[4, 2]

```

Warto zauważyć, że poprzedni kod dekoruje klasę zdefiniowaną przez użytkownika. Tak jak w oryginalnym przykładzie z rozdziału 30., możemy także wykorzystać dekorator do opakowania typu wbudowanego, takiego jak lista, o ile albo wykorzystamy klasę podrzędną w celu umożliwienia składni dekoracji, albo dekorację wykonamy ręcznie — składnia dekoratora wymaga instrukcji `class` dla wiersza ze znakiem @.

W poniższym kodzie zmienna `x` jest tak naprawdę znowu instancją klasy `Wrapper` z powodu pośredniego działania dekoracji (przeniosłem klasę dekoratora do pliku modułu `tracer.py` w celu jego ponownego wykorzystania w ten sposób).

```

>>> from tracer import Tracer          # Dekorator przeniesiony do pliku modułu
>>> @Tracer
... class MyList(list): pass           # myList = Tracer(MyList)
>>> x = MyList([1, 2, 3])              # Wywołuje Wrapper()
>>> x.append(4)                      # Wywołuje __getattribute__, append
Śledzenie: append
>>> x.wrapped
[1, 2, 3, 4]

>>> WrapList = Tracer(list)           # Lub ręczne wykonanie dekoracji
>>> x = WrapList([4, 5, 6])           # Inaczej wymagana instrukcja klasy podrzędnnej

```

```
>>> x.append(7)
Śledzenie: append
>>> x.wrapped
[4, 5, 6, 7]
```

Rozwiążanie z dekoratorem pozwala nam na przeniesienie tworzenia instancji do samego dekoratora zamiast wymagania przekazania utworzonego wcześniej obiektu. Choć różnica wydaje się nieznaczna, pozwala nam to na zachowanie normalnej składni tworzenia instancji i ogólnie na skorzystanie ze wszystkich zalet dekoratorów. Zamiast wymagać, by wszystkie wywołania tworzące instancję ręcznie przekierowywały obiekty do obiektu pośredniczącego, wystarczy, że rozszerzymy klasy za pomocą składni dekoratorów.

```
@Tracer
class Person: ...
bob = Person('Robert', 40, 50)
anna = Person('Anna', rate=100, hours=60)

class Person: ...
# Rozwiążanie bez dekoratora
bob = Wrapper(Person('Robert', 40, 50))
anna = Wrapper(Person('Anna', rate=100, hours=60))
```

Zakładając, że będziemy tworzyć więcej niż jedną instancję klasy, dekoratory zazwyczaj będą lepszym rozwiązaniem z punktu widzenia zarówno wielkości kodu, jak i jego późniejszego utrzymywania.



Uwaga na temat wersji: Jak wiemy z rozdziału 37., metoda `__getattr__` przechwytuje próby dostępu do metod przeciążających operatory, takich jak `__str__` czy `__repr__`, w Pythonie 2.6, ale nie w wersji 3.0.

W Pythonie 3.0 instancje klas dziedziczą wartości domyślne dla części (jednak nie wszystkich) z tych nazw po klasie (a tak naprawdę po automatycznej klasie nadzędnej `object`), ponieważ wszystkie klasy są „w nowym stylu”. Co więcej, w wersji 3.0 atrybuty wywoływane w sposób niejawny dla operacji wbudowanych, takich jak wyświetlanie czy `+`, nie są przekierowywane za pośrednictwem metody `__getattr__` (bądź jej kremnego, `__getattribute__`). Klasy w nowym stylu wyszukują takie metody w `klasach` i całkowicie pomijają normalne wyszukiwanie w instancjach.

Tutaj oznacza to, że obiekt opakowujący oparty na metodzie `__getattr__` automatycznie będzie śledził i przekazywał wywołania przeciążające operatory w Pythonie 2.6, ale w wersji 3.0 już nie. By się o tym przekonać, wystarczy wyświetlić zmienną `x` w sposób bezpośredni na końcu poprzedniej sesji interaktywnej. W Pythonie 2.6 metoda `__repr__` atrybutu jest śledzona i lista wyświetlana jest w oczekiwany sposób, natomiast w wersji 3.0 śledzenie nie występuje, a lista wyświetlana jest z wykorzystaniem domyślnego sposobu wyświetlania dla klasy `Wrapper`.

```
>>> x
Śledzenie: __repr__
[4, 5, 6, 7]
>>> x
# Python 3.0
<tracer.Wrapper object at 0x026C07D0>
```

By kod działał w ten sam sposób w Pythonie 3.0, należy ponownie zdefiniować metody przeciążania operatorów w klasie opakowującej — ręcznie, za pomocą narzędzi lub definicji w klasie nadzędnej. Jedynie proste nazwane atrybuty będą działały tak samo w obu wersjach. Z tą różnicą między wersjami spotkamy się ponownie w dekoratorze `Private` w dalszej części rozdziału.

Uwagi na temat klas II — zachowanie większej liczby instancji

Co ciekawe, funkcję dekoratora z tego przykładu można by *prawie* zapisać w kodzie jako klasę, a nie funkcję, z odpowiednim protokołem przeciążania operatora. Poniższa, nieco uproszczona alternatywa działa w podobny sposób, ponieważ jej metoda `__init__` wywoływana jest, kiedy dekorator @ zostanie zastosowany do klasy, natomiast jej metoda `__call__` uruchamiana jest, kiedy tworzona jest instancja klasy podmiotowej. Nasze obiekty są tym razem tak naprawdę instancjami klasy `Tracer` i w gruncie rzeczy wymieniamy tutaj referencję do zakresu funkcji zawierającej na atrybut `instancji`.

```
class Tracer:
    def __init__(self, aClass):
        self.aClass = aClass
    def __call__(self, *args):
        self.wrapped = self.aClass(*args)
        return self
    def __getattr__(self, attrname):
        print('Śledzenie: ' + attrname)
        return getattr(self.wrapped, attrname)

@Tracer
class Spam:
    def display(self):
        print('Mielonka!' * 8)

...
food = Spam()
food.display()                                # Wywołuje __call__
                                                # Wywołuje __getattr__
```

Jak jednak widzieliśmy wcześniej, alternatywa z klasą obsługuje większą liczbę klas, podobnie jak poprzednie rozwiązanie, jednak w zasadzie nie działa dla *większej liczby instancji* określonej klasy — każde wywołanie tworzące instancję wywołuje metodę `__call__`, która nadpisuje poprzednią instancję. W rezultacie klasa `Tracer` zapisuje tylko jedną instancję — ostatnią utworzoną. Warto samodzielnie poeksperymentować, by się o tym przekonać, jednak poniżej znajduje się przykład tego problemu.

```
@Tracer
class Person:
    def __init__(self, name):
        self.name = name

bob = Person('Robert')
print(bob.name)                                # Obiekt bob jest tak naprawdę instancją Wrapper
                                                # Wrapper osadza Person
anna = Person('Anna')
print(anna.name)                                # Obiekt anna nadpisuje obiekt bob
print(bob.name)                                # UPS: teraz obiekt bob nazywa się 'Anna'!
```

Wynik powyższego kodu będzie następujący. Ponieważ ta klasa śledząca ma tylko jedną wspólną instancję, druga z nich nadpisuje pierwszą.

```
Śledzenie: name
Robert
Śledzenie: name
Anna
Śledzenie: name
Anna
```

Problemem jest tutaj złe zachowanie stanu. Tworzymy jedną instancję dekoratora na klasę, ale nie na instancję klasy, przez co jedynie ostatnia instancja zostanie zachowana. Rozwiązaniem, jak w poprzednich rozważaniach na temat klas w przypadku dekoracji metod, jest porzucenie dekoratorów opartych na klasach.

Wersja klasy `Tracer` oparta na funkcji `działa` dla większej liczby instancji, ponieważ każde wywołanie tworzące instancję tworzy nową instancję klasy `Wrapper`, zamiast nadpisywać stan pojedynczej, współdzielonej instancji `Tracer`. Oryginalna wersja bez dekoratora również poprawnie obsługuje większą liczbę instancji — z tych samych przyczyn. Dekoratory są nie tylko nieco magiczne, ale także dość subtelne w swoim działaniu!

Dekoratory a funkcje zarządzające

Bez względu na subtelności tego typu przykład dekoratora klasy `Tracer` nadal opiera się na metodzie `_getattr_` przechwytyjącej pobrania opakowanego i osadzonego obiektu instancji. Jak widzieliśmy wcześniej, jedyne, co udało nam się osiągnąć, to przeniesienie wywołania tworzącego instancję wewnątrz klasy zamiast przekazywania instancji do funkcji zarządzającej. W przypadku oryginalnego przykładu śledzenia bez dekoratora po prostu w inny sposób zapisaliśmy kod tworzenia instancji.

```
class Spam:
    ...
    food = Wrapper(Spam())
# Wersja bez dekoratora
# Wystarczy jakakolwiek klasa
# Specjalna składnia tworzenia instancji

@Tracer
class Spam:
    ...
    food = Spam()
# Wersja z dekoratorem
# Wymaga składni z @ w klasie
# Normalna składnia tworzenia instancji
```

Dekoratory klasy przesuwają wymagania w zakresie specjalnej składni z wywołania tworzącego instancję do samej instrukcji `class`. Tak samo było w przypadku przykładu singletona umieszczonego wcześniej — zamiast dekorowania klasy i użycia normalnych wywołań tworzących instancję po prostu przekazywaliśmy klasę oraz jej argumenty konstrukcyjne do funkcji zarządzającej.

```
instances = {}
def getInstance(aClass, *args):
    if aClass not in instances:
        instances[aClass] = aClass(*args)
    return instances[aClass]

bob = getInstance(Person, 'Robert', 40, 10)           # Nie: bob = Person('Robert', 40, 10)
```

Alternatywnie moglibyśmy wykorzystać możliwości Pythona w zakresie introspekcji do pobrania klasy z utworzonej już instancji (zakładając, że tworzenie początkowej instancji jest dopuszczalne).

```
instances = {}
def getInstance(object):
    aClass = object.__class__
    if aClass not in instances:
        instances[aClass] = object
    return instances[aClass]

bob = getInstance(Person('Robert', 40, 10))           # Nie: bob = Person('Robert', 40, 10)
```

Tak samo jest w przypadku *dekoratorów funkcji*, takich jak napisana wcześniej funkcja śledząca. Zamiast dekorować funkcję logiką przechwytyującą późniejsze wywołania, moglibyśmy po prostu przekazać funkcję oraz jej argumenty do funkcji zarządzającej przesyłającej wywołanie.

```
def func(x, y):                                # Wersja bez dekoratora
    ...                                            # def tracer(func, args): ... func(*args)
    result = tracer(func, (1, 2))                # Specjalna składnia wywołania

@tracer
def func(x, y):                                # Wersja z dekoratorem
    ...                                            # Ponownie dowiązuje nazwę: func = tracer(func)
    result = func(1, 2)                          # Normalna składnia wywołania
```

Rozwiązania z funkcjami zarządzającymi, jak powyższe, przenoszą ciężar użycia specjalnej składni na *wywołania*, zamiast oczekiwania składni dekoracji w definicjach funkcji oraz klas.

Do czego służą dekoratory? (raz jeszcze)

Dlaczego zatem zaprezentowałem właśnie sposoby *nieużywania* dekoratorów w implementacji klas typu singleton? Jak wspomniałem na początku niniejszego rozdziału, dekoratory często oznaczają pewne kompromisy. Choć składnia ma znaczenie, zbyt często zapominamy zadać pytanie „po co?”, kiedy spotykamy się z nowymi narzędziami. Skoro już widzieliśmy, jak działają dekoratory, spróbujmy poświecić chwilę na spojrzenie na nie z pewnej perspektywy.

Jak większość funkcji języka, dekoratory mają zarówno wady, jak i zalety. Przykładowo wśród wad możemy wymienić dwa potencjalne niedociągnięcia *dekoratorów klas*:

Zmiany typu

Jak widzieliśmy, kiedy wstawiane są obiekty opakowujące, udekorowana funkcja bądź klasa nie zachowuje *oryginalnego typu*. Nazwa dowiązywana jest ponownie do obiektu opakowującego, co może mieć znaczenie w programach wykorzystujących nazwy obiektów lub sprawdzających ich typy. W przykładzie z singletonem zarówno rozwiązanie z dekoratorem, jak i funkcją zarządzającą zachowuje oryginalny typ klasy dla instancji. W kodzie śledzącym żadne z rozwiązań tego nie robi, ponieważ wymagane są obiekty opakowujące.

Dodatkowe wywołania

Warstwa opakowująca dodana przez dekorację powoduje dodatkowe obciążenie w zakresie wydajności spowodowane przez *dodatkowe wywołania* wykonywane za każdym razem, gdy wywoływany jest udekorowany obiekt. Wywołania są operacjami pochłaniającymi stosunkowo dużo czasu, przez co dekorujące obiekty opakowujące mogą spowolnić program. W kodzie śledzącym oba rozwiązania wymagają, by każdy atrybut był przekierowywany za pośrednictwem warstwy opakowującej. Przykład signletona unika dodatkowych wywołań, zachowując oryginalny typ klasy.

Podobne uwagi można mieć do *dekoratorów funkcji* — zarówno dekoracja, jak i funkcje zarządzające wymuszają dodatkowe wywołania, a zmiana typu zazwyczaj występuje w trakcie dekoracji (jednak nie w tym drugim przypadku).

Żadna z poruszonych kwestii nie jest jednak zbyt poważna. W przypadku większości programów różnica typu raczej nie będzie miała znaczenia, a utrata szybkości związana z dodatkowymi wywołaniami będzie nieznaczna. Co więcej, ta ostatnia występuje jedynie wtedy, gdy wykorzystywane są obiekty opakowujące, i często można sobie z nią poradzić, usuwając dekorator,

gdy wymagana jest optymalna wydajność. Utrata szybkości występuje również w rozwiązańach bez dekoratora, które dodają logikę opakowującą (w tym w *metaklasach*, o czym przekonamy się w rozdziale 39.).

I odważnie, jak widzieliśmy na początku rozdziału, dekoratory mają trzy istotne zalety. W porównaniu z rozwiązaniami z funkcją zarządzającą (inaczej: pomocniczą) z poprzedniego podrozdziału dekoratory to także:

Jasna, jawną składnia

Dekoratory sprawiają, że rozszerzenie jest jasne i oczywiste. Ich składnia ze znakiem @ jest łatwiejsza do rozpoznania od specjalnego kodu wywołań, który może się pojawić w dowolnym miejscu pliku źródłowego. W naszych przykładach singletona i śledzenia wiersze z dekoratorem wydają się łatwiejsze do zauważenia niż dodatkowy kod w wywołaniach. Co więcej, dekoratory pozwalają na wykorzystywanie normalnej, znanej wszystkim programistom Pythona składni funkcji oraz wywołań tworzących instancje.

Utrzymywanie kodu

Dekoratory pozwalają uniknąć powtarzania kodu rozszerzającego w każdej funkcji czy wywołaniu klasy. Ponieważ pojawiają się tylko raz, w samej definicji klasy lub funkcji, ograniczają powtarzalność kodu i upraszczają utrzymywanie go w przyszłości. W przypadku naszego kodu singletona oraz śledzenia musielibyśmy zastosować specjalny kod w każdym wywołaniu, by skorzystać z rozwiązań z funkcją zarządzającą. Dodatkowa praca wymagana była zatem nie tylko na początku, ale także w przypadku wszelkich modyfikacji, które będą musiały być wykonane w przyszłości.

Spójność

Dekoratory sprawiają, że mniej prawdopodobne będzie to, iż programista zapomni skorzystać z wymaganej logiki opakowującej. Wynika to w dużej mierze z dwóch poprzednich zalet. Ponieważ dekoracja jest jasna i pojawia się tylko raz, w samych udekorowanych obiektach, dekoratory promują bardziej spójne i jednolite wykorzystywanie API w stosunku do specjalnego kodu, który należy umieszczać w każdym wywołaniu. W przykładzie z singletonem łatwo byłoby zapomnieć o przekierowaniu wszystkich wywołań tworzących klasę do specjalnego kodu, co całkowicie zaprzeczyłoby idei zarządzania singletonem.

Dekoratory promują także *hermetyzację* kodu w celu ograniczenia jego powtarzalności i zminimalizowania wysiłku wymaganego przy jego utrzymywaniu w przyszłości. Choć inne narzędzia strukturyzujące kod także to robią, w przypadku dekoratorów jest to naturalne dla wszystkich zadań związanych z rozszerzaniem.

Żadna z wymienionych zalet nie nakazuje jednak bezwzględnie używać składni dekoratorów, a samo ich wykorzystanie jest wyborem stylistycznym. Jednak większość programistów uznaje dekoratory za rozwiązań korzystniejsze, zwłaszcza w roli narzędzi umożliwiających poprawne korzystanie z bibliotek oraz API.

Pamiętam podobną argumentację za i przeciw funkcjom konstruktora w klasach. Przed wprowadzeniem metod `__init__` ten sam efekt można było często osiągnąć po ręcznym wywołaniu metody na instancji przy tworzeniu jej (na przykład `X=Class().init()`). Z czasem jednak, choć był to wybór w dużej mierze stylistyczny, składnia `__init__` stała się preferowana, gdyż była bardziej oczywista, jawną, spójną i łatwiejszą w utrzymaniu. Choć każdy powinien decydować za siebie, dekoratory zdają się kłaść na szali te same zalety.

Bezpośrednie zarządzanie funkcjami oraz klasami

Większość z naszych przykładów z niniejszego rozdziału została zaprojektowana pod kątem przechwytywania wywołań funkcji oraz wywołań tworzących instancje. Choć jest to typowe zastosowanie dekoratorów, nie są one ograniczone do tej roli. Ponieważ dekoratory działają, przekazując nowe funkcje i klasy do kodu dekoratora, można je także wykorzystać do zarządzania samymi obiektami funkcji oraz klas, a nie tylko wykonywanymi do nich wywołaniami.

Wyobraźmy sobie na przykład, że metody bądź klasy wykorzystywane przez aplikację muszą być rejestrowane w API w celu późniejszego przetworzenia (być może API wywoła te obiekty później, w odpowiedzi na zdarzenia). Choć można udostępnić funkcję rejestrującą, którą wywołamy ręcznie po zdefiniowaniu obiektów, dekoratory uczynią nasze intencje bardziej oczywistymi.

Poniższa prosta implementacja tego pomysłu definiuje dekorator, który można zastosować zarówno do funkcji, jak i klas w celu dodania obiektu do rejestru opartego na słowniku. Ponieważ zwraca ona sam obiekt zamiast obiektu opakowującego, nie przechwytuje późniejszych wywołań.

```
# Rejestrowanie udekorowanych obiektów w API

registry = {}
def register(obj):
    registry[obj.__name__] = obj
    return obj

@register
def spam(x):
    return(x ** 2)                                # Dekorator zarówno klasy, jak i funkcji
                                                    # Dodanie do rejestru
                                                    # Zwrócenie samego obiektu, a nie obiektu opakowującego

@register
def ham(x):
    return(x ** 3)

@register
class Eggs:                                     # Eggs = register(Eggs)
    def __init__(self, x):
        self.data = x ** 4
    def __str__(self):
        return str(self.data)

print('Rejestr:')
for name in registry:
    print(name, '=>', registry[name], type(registry[name]))

print('\nWywołania ręczne:')
print(spam(2))                                    # Ręczne wywołanie obiektów
print(ham(2))                                    # Późniejsze wywołania nie są przechwytywane
X = Eggs(2)
print(X)

print('\nWywołania z rejestru:')
for name in registry:
    print(name, '=>', registry[name](3))          # Wywołanie z rejestru
```

Po wykonaniu powyższego kodu udekorowane obiekty dodawane są do rejestru po nazwach, jednak przy późniejszym wywołaniu nadal działają w taki sposób, jak je zaprojektowano, bez przekierowania do warstwy opakowującej. Tak naprawdę nasze obiekty można wykonać zarówno ręcznie, jak i z wewnątrz tabeli rejestru.

```
Rejestr:  
Eggs => <class '__main__.Eggs'> <class 'type'>  
ham => <function ham at 0x02CFB738> <class 'function'>  
spam => <function spam at 0x02CFB6F0> <class 'function'>
```

```
Wywołania ręczne:  
4  
8  
16
```

```
Wywołania z rejestru:  
Eggs => 81  
ham => 27  
spam => 9
```

Technikę tę mogliby wykorzystywać na przykład interfejs użytkownika do rejestrowania programów obsługi wywołań zwrotnych dla działań użytkownika. Programy obsługi mogą być rejestrowane za pomocą nazwy funkcji bądź klasy, jak powyżej; można także wykorzystać argumenty dekoratora do określenia podmiotowego zdarzenia. Dodatkowa instrukcja `def` zawierająca dekorator mogłaby zostać użyta do zachowania takich argumentów w celu zastosowania ich w dekoracji.

Powыższy przykład jest dość sztuczny, ale technika ta jest bardzo uniwersalna. Dekoratory funkcji można na przykład wykorzystać także do przetwarzania atrybutów funkcji, natomiast dekoratory klas mogą w sposób dynamiczny wstawiać nowe atrybuty klas czy nawet nowe metody. Rozważmy poniższe dekoratory funkcji. Przypisują one atrybuty funkcji do informacji rekorдов w celu późniejszego wykorzystania ich przez API, jednak nie wstawiają warstwy opakowującej, przechwytyjącej późniejsze wywołania.

```
# Bezpośrednie rozszerzenie udekorowanych obiektów  
  
>>> def decorate(func):  
...     func.marked = True  
...     return func  
...  
>>> @decorate  
... def spam(a, b):  
...     return a + b  
...  
>>> spam.marked  
True  
  
>>> def annotate(text):  
...     def decorate(func):  
...         func.label = text  
...         return func  
...     return decorate  
...  
>>> @annotate('mielonka dane')  
... def spam(a, b):  
...     return a + b  
...  
>>> spam(1, 2), spam.label  
(3, 'mielonka dane')
```

Dekoratory tego typu rozszerzają funkcje oraz klasy w sposób bezpośredni, bez przechwytywania ich późniejszych wywołań. Więcej przykładów dekoracji klas zarządzającej bezpośrednio klasami zobaczymy w kolejnym rozdziale, ponieważ kwestia ta okazuje się pokrywać z dziedziną *metaklas*. W dalszej części niniejszego rozdziału zajmiemy się działaniem dwóch większych przypadków użycia dekoratorów.

Przykład — atrybuty „prywatne” i „publiczne”

Ostatnie dwie części niniejszego rozdziału prezentują większe przykłady zastosowania dekoratorów. Oba zawierają skróconą część opisową — po części dlatego, że rozdział ten już przekroczył przewidziany dla niego limit długości, ale również z powodu tego, że Czytelnik powinien już na tyle dobrze rozumieć podstawy dekoratorów, by potrafić przestudiować przykłady samodzielnie. Są to narzędzia ogólnego zastosowania i dają nam możliwość przekonania się, w jaki sposób koncepcje związane z dekoratorami łączą się ze sobą w bardziej użytecznym kodzie.

Implementacja atrybutów prywatnych

Poniższy dekorator *klasy* implementuje deklarację atrybutów instancji klasy jako prywatnych. Oznacza to, że atrybuty są przechowywane w instancji lub dziedziczone po jednej z jej klas. Nie pozwala on na pobieranie i modyfikację takich atrybutów spoza udekorowanej klasy, jednak nadal pozwala samej klasie na swobodny dostęp do tych zmiennych wewnętrz jej metod. Kod ten nie do końca jest tym samym co mechanizmy języków C++ czy Java, ale umożliwia podobną kontrolę dostępu jako element opcjonalny Pythona.

W rozdziale 29. widzieliśmy już niepełną, wstępna implementację prywatności atrybutów instancji w przypadku modyfikacji. Wersja z tego rozdziału rozszerza tę koncepcję w taki sposób, by sprawdzać również próby pobierania atrybutu. Do implementacji tego modelu wykorzystuje także delegację zamiast dziedziczenia. Tak naprawdę w pewnym sensie jest to tylko rozszerzenie dekoratora klasy śledzącej atrybuty, z którym spotkaliśmy się wcześniej.

Choć przykład ten wykorzystuje do implementacji prywatności atrybutów nowość składniową, jaką są dekoratory klas, przechwytywanie atrybutów jest w nim w dużej mierze nadal oparte na metodach przeciążania operatorów `__getattr__` oraz `__setattr__`, z którymi spotkaliśmy się w poprzednich rozdziałach. Kiedy wykryta zostaje próba dostępu do atrybutu prywatnego, ta wersja kodu wykorzystuje instrukcję `raise` do zgłoszenia wyjątku wraz z komunikatem o błędzie. Wyjątek ten można przechwycić w instrukcji `try` lub pozwolić mu na zakończenie skryptu.

Poniżej znajduje się kod przykładu wraz z testem samosprawdzającym zamieszczonym na dole pliku. Działa on zarówno w Pythonie 2.6, jak i 3.0, ponieważ wykorzystuje składnię `print` i `raise` z wersji 3.0, jednak przechwytuje atrybuty metod przeciążania operatorów jedynie w wersji 2.6 (więcej na ten temat za chwilę).

```
"""
Prywatność dla atrybutów pobranych z instancji klas.
Przykłady użycia można znaleźć w kodzie testu samosprawdzającego na dole pliku.
Dekorator jest tym samym co: Doubler = Private('data', 'size')(Doubler).
Private zwraca onDecorator, onDecorator zwraca onInstance, a każda instancja onInstance osadza instancję Doubler.
"""

traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def Private(*privates):
    def onDecorator(aClass):
        class OnInstance:
            def __init__(self, *args, **kargs):
                # privates w zakresie funkcji zawierającej
                # aClass w zakresie funkcji zawierającej
                # Opakowane w atrybut instancji
                self.wrapped = aClass(*args, **kargs)
    return onDecorator
```

```

def __getattribute__(self, attr):           # Moje atrybuty nie wywołuję metody getattribute
    trace('pobranie:', attr)               # Inne zakładane w obiekcie wrapped
    if attr in privates:
        raise TypeError('pobranie atrybutu prywatnego: ' + attr)
    else:
        return getattr(self.wrapped, attr)
def __setattr__(self, attr, value):         # Próby dostępu z zewnątrz
    trace('ustawienie:', attr, value)      # Pozostale działają normalnie
    if attr == 'wrapped':                  # Pozwolenie na własne atrybuty
        self.__dict__[attr] = value        # Uniknięcie pętli
    elif attr in privates:
        raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
    else:
        setattr(self.wrapped, attr, value) # Opakowane atrybuty obiektu
                                                # Lub użycie __dict__
return onInstance                           # Lub użycie __dict__
return onDecorator

if __name__ == '__main__':
    traceMe = True

@Private('data', 'size')                   # Doubler = Private(...)(Doubler)
class Doubler:
    def __init__(self, label, start):
        self.label = label
        self.data = start
    def size(self):
        return len(self.data)
    def double(self):
        for i in range(self.size()):
            self.data[i] = self.data[i] * 2
    def display(self):
        print('%s => %s' % (self.label, self.data))

X = Doubler('X to', [1, 2, 3])
Y = Doubler('Y to', [-10, -20, -30])

# Poniższe testy kończą się powodzeniem
print(X.label)                            # Próby dostępu we wnętrzu podmiotowej klasy
X.display(); X.double(); X.display()       # Nie zostaje przechwycony: wykonany normalnie
print(Y.label)                            # Metody działają bez sprawdzania
Y.display(); Y.double()                   # Ponieważ prywatność nie jest dziedziczona
Y.label = 'Mielonka'
Y.display()

# Poniższe testy wszystkie kończą się niepowodzeniem (zgodnie z zamierzeniami)
"""
print(X.size())                          # Wyświetla "TypeError: pobranie atrybutu prywatnego: size"
print(X.data)
X.data = [1, 1, 1]
X.size = lambda S: 0
print(Y.data)
print(Y.size())
"""


```

Kiedy traceMe zwraca True, kod testu samosprawdzającego pliku modułu zwraca następujące dane wyjściowe. Warto zwrócić uwagę na to, w jaki sposób dekorator przechwytuje i sprawdza zarówno próby pobierania, jak i przypisania wykonywane poza opakowaną klasą — jednak nie przechwytuje prób dostępu wykonywanych we wnętrzu samej klasy.

```
[ustawienie: wrapped <__main__.Doubler object at 0x02B2AAFO>]
[ustawienie: wrapped <__main__.Doubler object at 0x02B2AE70>]
[pobranie: label]
X to
```

```
[pobranie: display]
X to => [1, 2, 3]
[pobranie: double]
[pobranie: display]
X to => [2, 4, 6]
[pobranie: label]
Y to
[pobranie: display]
Y to => [-10, -20, -30]
[pobranie: double]
[ustawienie: label Mielonka]
[pobranie: display]
Mielonka => [-20, -40, -60]
```

Szczegóły implementacji I

Powyższy kod jest nieco skomplikowany i najlepiej będzie prześledzić go samodzielnie w celu przekonania się, jak działa. By pomóc w tej analizie, poniżej znajduje się kilka elementów, na które warto zwrócić uwagę.

Dziedziczenie a delegacja

Pierwszy, wstępny przykład implementacji prywatności z rozdziału 29. wykorzystywał *dziedziczenie* do wzmieszania metod `__setattr__` do przechwytywania prób dostępu. Dziedziczenie mocno to jednak utrudnia, ponieważ rozróżnienie pomiędzy dostępem z wewnętrz i z zewnątrz klasy nie jest tak oczywiste (dostęp z wewnętrz powinien móc się odbywać normalnie, natomiast dostęp z zewnątrz powinien być ograniczony). By to obejść, przykład z rozdziału 29. wymagał, aby dziedziczące klasy wykorzystywały przypisania `__dict__` w celu ustawienia atrybutów — rozwiązanie to można w najlepszym razie nazwać niepełnym.

Wersja zaprezentowana powyżej wykorzystuje *delegację* (osadzenie jednego obiektu wewnętrz innego) zamiast dziedziczenia. Taki wzorzec kodu lepiej nadaje się do naszego zadania, gdyż bardzo ułatwia rozróżnienie pomiędzy próbami dostępu z wewnętrz oraz z zewnątrz podmiotowej klasy. Dostęp do atrybutów spoza klasy podmiotowej jest przechwytywany przez metody przeciążania operatorów warstwy opakowującej i delegowany do klasy, jeśli jest poprawny. Próby dostępu wewnętrz samej klasy (na przykład za pośrednictwem `self` wewnątrz kodu metod) nie są przechwytywane i mogą być wykonywane normalnie bez sprawdzania, ponieważ prywatność nie jest tutaj dziedziczona.

Argumenty dekoratora

Wykorzystany tutaj dekorator klasy przyjmuje dowolną liczbę argumentów nazywających atrybuty prywatne. Tak naprawdę argumenty przekazywane są do funkcji `Private`, natomiast `Private` zwraca funkcję dekoratora, która ma być zastosowana do podmiotowej klasy. Argumenty wykorzystywane są zatem przed wystąpieniem dekoracji. Funkcja `Private` zwraca dekorator, który z kolei „pamięta” listę atrybutów prywatnych w postaci referencji do zakresu funkcji zawierającej.

Zachowywanie stanu i zakresy funkcji zawierającej

A skoro już mowa o zakresach funkcji zawierającej, tak naprawdę w powyższym kodzie zachowywanie stanu działa na trzech poziomach:

- Argumenty funkcji `Private` wykorzystywane są, zanim nastąpi dekoracja, i są zachowywane w postaci referencji do zakresu funkcji zawierającej — do wykorzystania w `onDecorator` oraz `onInstance`.
- Argument klasy dla `onDecorator` wykorzystywany jest w czasie dekoracji i zachowywany w postaci referencji do zakresu funkcji zawierającej — do wykorzystania w czasie tworzenia instancji.
- Obiekt opakowanej instancji zachowywany jest w postaci atrybutu instancji w `onInstance` — do wykorzystania, gdy w późniejszym czasie ma miejsce próba dostępu do atrybutów spoza klasy.

Wszystko to działa w miarę naturalnie w oparciu o reguły Pythona dotyczące zakresów oraz przestrzeni nazw.

Wykorzystanie `_dict_` oraz `_slots_`

Metoda `__setattr__` z powyższego kodu, próbując ustawić własny opakowany atrybut `onInstance`, opiera się na słowniku `_dict_` przestrzeni nazw atrybutów obiektu instancji. Jak wiemy z poprzedniego rozdziału, nie może ona przypisać atrybutu w sposób bezpośredni bez wykonania pętli. Wykorzystuje ona jednak funkcję wbudowaną `setattr` w miejsce `_dict_` w celu ustawienia atrybutów w samym opakowanym obiekcie. Co więcej, do pobrania atrybutów w opakowanym obiekcie wykorzystana zostaje metoda `getattr`, ponieważ mogą one być przechowywane w samym obiekcie bądź odziedziczone przez niego.

Z tej przyczyny kod ten będzie działał dla większości klas. Niektóre osoby mogą pamiętać z rozdziału 31., że klasy w nowym stylu ze `_slots_` mogą nie przechowywać atrybutów w słowniku `_dict_`. Ponieważ jednak polegamy na `_dict_` jedynie na poziomie `onInstance`, a nie w opakowanej instancji, a także dlatego, że metody `setattr` oraz `getattr` mają zastosowanie do atrybutów opartych zarówno na `_dict_`, jak i na `_slots_`, nasz dekorator działa na klasach wykorzystujących dowolny z tych mechanizmów przechowywania.

Uogólnienie kodu pod kątem deklaracji atrybutów jako publicznych

Skoro już mamy implementację prywatności, dość łatwo jest uogólnić ten kod w celu dopuszczenia także deklaracji publicznych — są one właściwie odwrotnością deklaracji prywatnych, dlatego wystarczy, że będą negować wewnętrzny test. Przykład wymieniony w tym podrozdziale pozwala klasie wykorzystywać dekoratory do zdefiniowania zbioru publicznych lub prywatnych atrybutów instancji (atrributów przechowywanych w instancji lub odziedziczonych po jej klasach) za pomocą następującej semantyki:

- `Private` deklaruje atrybuty instancji klas, których nie można pobierać lub przypisywać, z wyjątkiem pochodzących z wewnętrz kodu metod klasy. Oznacza to, że nie jest możliwy dostęp z zewnątrz do żadnej zmiennej zadeklarowanej jako prywatna, natomiast wszystkie zmienne niezadeklarowane w ten sposób można swobodnie pobierać i przypisywać z zewnątrz.
- `Public` deklaruje atrybuty instancji klas, które można pobierać lub przypisywać zarówno z zewnątrz klasy, jak i z wewnętrz jej metod. Oznacza to, że dostęp do każdej zmiennej zadeklarowanej jako publiczna jest możliwy z dowolnego miejsca, natomiast dostęp z zewnątrz do zmiennych niezadeklarowanych w ten sposób nie jest możliwy.

Deklaracje Private oraz Public mają się wzajemnie wykluczać. Kiedy używamy Private, wszystkie niezadeklarowane zmienne uznawane są za publiczne, natomiast jeśli użyjemy Public, wszystkie niezadeklarowane zmienną uznawane są za prywatne. Są one swoimi przeciwieństwami, choć niezadeklarowane zmienne nieutworzone przez metody klas zachowują się nieco inaczej — mogą zostać przypisane i tym samym utworzone poza klasą pod Private (wszystkie niezadeklarowane zmienne są dostępne), jednak pod Public już nie (wszystkie niezadeklarowane zmienne są niedostępne).

Ponownie zachęcam do samodzielnego przestudiowania kodu w celu przekonania się, jak on działa. Warto zwrócić uwagę na to, że rozwiązywanie to na górze dodaje dodatkowy, *czwarty poziom zachowywania stanu*, poza poziomami opisanymi w poprzednim podrozdziale. Funkcje sprawdzające wykorzystywane przez lambda są zapisywane w dodatkowym zakresie zawierającym. Przykład ten napisany jest w taki sposób, by działał w Pythonie 2.6 lub 3.0, choć ma pewne ograniczenie w wersji 3.0 (wyjaśnione pokrótko w łańcuchu znaków dokumentacji pliku i nieco szerzej w opisie następującym po kodzie).

"""

Dekorator klasy z deklaracjami atrybutów jako prywatne oraz publiczne.

Kontroluje dostęp do atrybutów przechowywanych w instancji lub dziedziczonych przez nią po klasach. Private deklaruje nazwy atrybutów, których nie można pobrać lub przypisać z zewnątrz udekorowanej klasy. Public deklaruje nazwy atrybutów, które można pobrać lub przypisać z zewnątrz.

Uwaga: działa w Pythonie 3.0 jedynie dla atrybutów o normalnych nazwach. Metody przeciążania operatorów __X__ wykonywane w sposób niejawny dla operacji wbudowanych nie wywołują metod __getattr__ ani __getattribute__ w klasach w nowym stylu. Należy tutaj dodać metody __X__ w celu przechwytcenia i wydelegowania nazw wbudowanych.

"""

```
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped = aClass(*args, **kargs)
            def __getattr__(self, attr):
                trace('pobranie:', attr)
                if failIf(attr):
                    raise TypeError('pobranie atrybutu prywatnego: ' + attr)
                else:
                    return getattr(self.__wrapped, attr)
            def __setattr__(self, attr, value):
                trace('ustawienie:', attr, value)
                if attr == '_onInstance_wrapped':
                    self.__dict__[attr] = value
                elif failIf(attr):
                    raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
                else:
                    setattr(self.__wrapped, attr, value)
        return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(lambda attr: attr in attributes)

def Public(*attributes):
    return accessControl(lambda attr: attr not in attributes)
```

Przykłady użycia można znaleźć w kodzie testu samosprawdzającego poprzedniego przykładu. Oto szybkie spojrzenie na działanie tych dekoratorów klas w sesji interaktywnej (działają one tak samo w Pythonie 2.6 oraz 3.0). Zgodnie z tym, co zapowiadaliśmy, nazwy nie prywatne lub publiczne można pobierać i modyfikować spoza podmiotowej klasy, natomiast nie można tego robić w przypadku nazw prywatnych lub niepublicznych.

```
>>> from access import Private, Public

>>> @Private('age')
... class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
...>>> X = Person('Robert', 40)
>>> X.name                                     # Person = Private('age')(Person)
'Robert'                                         # Person = onInstance ze stanem
>>> X.name = 'Anna'                           # Próby dostępu z wewnętrz odbywają się normalnie
>>> X.name
'Anna'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tomasz'
TypeError: private attribute change: age

>>> @Public('name')
... class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
...>>> X = Person('robert', 40)                  # X jest instancją onInstance
>>> X.name                                     # onInstance osadza Person
'robert'
>>> X.name = 'Anna'                           # onInstance osadza Person
>>> X.name
'Anna'
>>> X.age
TypeError: private attribute fetch: age
>>> X.age = 'Tomasz'
TypeError: private attribute change: age
```

Szczegóły implementacji II

By pomóc w analizie kodu, poniżej znajduje się kilka finalnych uwag dotyczących tej wersji. Ponieważ jest ona jedynie uogólnieniem przykładu z poprzedniego podrozdziału, większość uwag, które go dotyczą, ma zastosowanie również tutaj.

Użycie nazw pseudoprzywatnych `_X`

Poza uogólnieniem wersja ta wykorzystuje również opcję nazw pseudoprzywatnych Pythona `_X` (z którą spotkaliśmy się w rozdziale 30.) do zlokalizowania atrybutu `wrapped` w klasie kontrolnej dzięki automatycznemu poprzedzeniu go nazwą klasy. Pozwala nam to uniknąć ryzyka konfliktów z atrybutem `wrapped`, który może być wykorzystywany przez prawdziwą, opakowaną klasę (co miało miejsce w poprzedniej wersji kodu) — może się to przydać w tak uniwersalnym narzędziu. Opcja ta nie zapewnia jednak prywatności, ponieważ taka nazwa

może być swobodnie wykorzystywana poza klasą. Warto również zauważyc, że w metodzie `__setattr__` będziemy musieli użyć łańcucha znaków pełnej, rozszerzonej nazwy ('`_on`
`→Instance``_wrapped`'), ponieważ na to zmienia ją Python.

Złamanie prywatności

Choć powyższy przykład implementuje kontrolę dostępu dla atrybutów instancji oraz jej klas, można tę kontrolę na różne sposoby odwrócić — na przykład w sposób jawnym przechodzącąc rozszerzoną wersję atrybutu `wrapped` (`bob.pay` może nie zadziałać, ale pełna nazwa `bob._onInstance_wrapped.pay` już może!). Jeśli jednak musimy to robić w sposób jawnym, poziom kontroli powinien być wystarczający na potrzeby normalnego, zamierzonego użycia. Oczywiście kontrolę prywatności można odwrócić w każdym języku programowania, o ile będziemy się wystarczająco mocno starać (w niektórych implementacjach C++ może także zadziałać `#define private public`). Choć kontrola dostępu może ograniczyć przypadkowe zmiany, w dowolnym języku programowania wiele zależy od programisty. W każdej sytuacji, w której kod źródłowy można zmienić, kontrola dostępu będzie mrzonką.

Kompromisy związane z dekoratorem

I znów, moglibyśmy uzyskać te same wyniki bez dekoratorów, wykorzystując funkcje zarządzające lub ręcznie pisząc kod ponownie dowiązujący nazwy dekoratorów. Składnia dekoratorów sprawia jednak, że całość jest nieco bardziej oczywista i spójna. Największą potencjalną wadą tego rozwiązania (i wszystkich innych opartych na obiektach opakowujących) jest to, że dostęp do atrybutów wymaga dodatkowego wywołania, a instancje udekorowanych klas nie są tak naprawdę instancjami oryginalnej dekorowanej klasy. Jeśli na przykład sprawdzimy ich typ za pomocą `X.__class__` czy `isinstance(X, C)`, okaże się, że są one instancjami klasy *opakowującej*. O ile jednak nie planujemy wykonywać introspekcji na typach obiektów, kwestia ta jest najprawdopodobniej bez znaczenia.

Znane problemy

W obecnej postaci przykład ten działa zgodnie z zamierzeniami w Pythonie 2.6 oraz 3.0 (o ile metody przeciążania operatorów, które mają być wydelegowane, zostaną zdefiniowane ponownie w obiekcie opakowującym). Jak to jednak ma miejsce w przypadku większości oprogramowania, zawsze coś można poprawić.

Ograniczenie: delegacja metod przeciążania operatorów kończy się niepowodzeniem w Pythonie 3.0

Tak jak wszystkie klasy oparte na delegacji, korzystające z metody `__getattribute__`, powyższy dekorator działa we wszystkich wersjach jedynie dla atrybutów o normalnych nazwach. Metody przeciążania operatorów, takie jak `__str__` oraz `__add__`, będą działały inaczej dla klas w nowym stylu i tym samym po wykonaniu kodu w Pythonie 3.0 nie dotrą do osadzonego obiektu, jeśli są tam zdefiniowane.

Jak wiemy z poprzedniego rozdziału, klasy tradycyjne normalnie wyszukują nazwy przeciążania operatorów w instancjach w czasie wykonywania, jednak klasy w nowym stylu tego nie robią — całkowicie pomijają instancję i wyszukują metody tego typu w klasach. Tym samym

metody przeciążania operatorów `_X_` wykonywane w sposób niejawny dla operacji wbudowanych *nie* wywołają ani metody `__getattr__`, ani `__getattribute__` dla klas w nowym stylu z Pythona 2.6 i wszystkich klas z wersji 3.0. Takie próby pobrania atrybutów pomijają całkowicie naszą metodę `onInstance.__getattr__`, dlatego nie da się ich sprawdzić ani wydelegować.

Nasza klasa dekoratora nie jest zapisana w kodzie jako klasa w nowym stylu (pochodząca od `object`), dlatego przechwyci metody przeciążania operatorów po wykonaniu w Pythonie 2.6. Ponieważ jednak w wersji 3.0 wszystkie klasy automatycznie są klasami w nowym stylu, metody tego typu *nie będą działały*, jeśli są one zapisane na osadzonym obiekcie. Najprostsze obejście tego ograniczenia w Pythonie 3.0 polega na ponownym, powtórzonym zdefiniowaniu w `onInstance` wszystkich metod przeciążania operatorów, które mogą być użyte w osadzonych obiektach. Takie dodatkowe metody można dodać ręcznie, za pomocą narzędzi po części automatyzujących to zadanie (na przykład dekoratorów klas lub metaklas omówionych w kolejnym rozdziale) lub za pomocą definicji w klasach nadzędnych.

By zobaczyć tę różnicę na własne oczy, można spróbować zastosować dekorator do klasy wykorzystującej metody przeciążania operatorów w Pythonie 2.6. Sprawdzanie działa jak poprzednio i zarówno metoda `__str__` wykorzystywana dla wyświetlania, jak i metoda `__add__` wykonywana dla operatora + wywołują metodę `__getattr__` dekoratora i tym samym zostaną poprawnie sprawdzone i wydelegowane do podmiotowego obiektu klasy `Person`.

```
C:\misc> c:\python26\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
...         self.age = 42
...     def __str__(self):
...         return 'Osoba: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()
>>> X.age
# Sprawdzanie poprawności nie powiedzie się (zgodnie z planem)
TypeError: pobranie atrybutu prywatnego: age
>>> print(X)
# __getattr__ => wykonuje Person.__str__
Osoba: 42
>>> X + 10
# __getattr__ => wykonuje Person.__add__
>>> print(X)
# __getattr__ => wykonuje Person.__str__
Osoba: 52
```

Gdy jednak ten sam kod wykonamy w Pythonie 3.0, wywoływane w niejawny sposób metody `__str__` oraz `__add__` pomijają metodę `__getattr__` dekoratora i szukają definicji w samej klasie dekoratora oraz ponad nią. Instrukcja `print` odnajduje domyślny sposób wyświetlania odziedziczony po typie klasy (a tak naprawdę po domniemanej klasie nadzędnej `object` w Pythonie 3.0), natomiast operator + generuje błąd, ponieważ nie odziedziczył żadnej wartości domyślnej.

```
C:\misc> c:\python30\python
>>> from access import Private
>>> @Private('age')
... class Person:
...     def __init__(self):
...         self.age = 42
...     def __str__(self):
```

```

...     return 'Osoba: ' + str(self.age)
...     def __add__(self, yrs):
...         self.age += yrs
...
>>> X = Person()                      # Sprawdzanie poprawności nazw nadal działa
>>> X.age                            # Jednak Python 3.0 nie deleguje nazw wbudowanych!
TypeError: pobranie atrybutu prywatnego: age
>>> print(X)
<access.onInstance object at 0x025E0790>
>>> X + 10
TypeError: unsupported operand type(s) for +: 'onInstance' and 'int'
>>> print(X)
<access.onInstance object at 0x025E0790>

```

Użycie alternatywnej metody `__getattribute__` nic tu nie pomoże. Choć jest ona zdefiniowana w taki sposób, by przechwytywać każdą referencję do atrybutu (a nie tylko dla nazw niezdefiniowanych), nie jest wykonywana przez operacje wbudowane. Opcja `property` Pythona (omówiona w rozdziale 37.) także nam nie pomoże. Warto przypomnieć, że właściwości to kod wykonywany automatycznie, powiązany z określonymi atrybutami, definiowany w czasie pisania klasy. Nie są one zaprojektowane do obsługi dowolnych atrybutów w opakowanych obiektach.

Jak wspomniano wcześniej, najprostszym rozwiązaniem w Pythonie 3.0 jest powtórne zdefiniowanie nazw przeciążających operatory, które mogą się pojawić w obiektach osadzonych w klasach opartych na delegacji, takich jak nasz dekorator. Nie jest to rozwiązanie idealne, ponieważ powoduje pewną powtarzalność kodu, zwłaszcza w porównaniu z rozwiązaniem przeznaczonym dla Pythona 2.6. Nie jest to jednak również zbyt duży wysiłek programistyczny, można go do pewnego stopnia zautomatyzować za pomocą narzędzi lub klas nadzrzednych, co wystarczy do umożliwienia działania dekoratora w Pythonie 3.0 i pozwoli na deklarowanie jako prywatne lub publiczne również nazw przeciążających operatory (przy założeniu, że każda metoda przeciążania operatora wewnętrznie wykonuje test `failIf`).

```

def accessControl(failIf):
    def onDecorator(aClass):
        class onInstance:
            def __init__(self, *args, **kargs):
                self.__wrapped__ = aClass(*args, **kargs)

            # Przechwycenie i wydelegowanie metod przeciążających operatory
            def __str__(self):
                return str(self.__wrapped__)
            def __add__(self, other):
                return self.__wrapped__ + other
            def __getitem__(self, index):
                return self.__wrapped__[index]           # W miarę potrzeb
            def __call__(self, *args, **kargs):
                return self.__wrapped__(*arg, *kargs)    # W miarę potrzeb
            ... i ewentualne inne potrzebne metody ...

            # Przechwycenie i wydelegowanie nazwanych atrybutów
            def __getattr__(self, attr):
                ...
            def __setattr__(self, attr, value):
                ...
        return onInstance
    return onDecorator

```

Po dodaniu powyższych metod przeciążania operatorów poprzedni przykład z metodami `__str__` oraz `__add__` działa tak samo w Pythonie 2.6 oraz 3.0, choć w wersji 3.0 niezbędne może być dodanie znacznej ilości dodatkowego kodu. Co do zasady *każda* metoda przeciążania

operatora, która nie jest wykonywana automatycznie, będzie musiała być ponownie zdefiniowana w Pythonie 3.0 dla narzędzi uniwersalnych tego typu (dlatego właśnie to rozszerzenie zostało pominięte w naszym kodzie). Ponieważ każda klasa jest w wersji 3.0 klasą w nowym stylu, kod oparty na delegacji jest tam nieco bardziej skomplikowany (choć nie niemożliwy).

Z drugiej strony, obiekty opakowujące i delegujące mogłyby po prostu dziedziczyć po wspólnej klasie nadrzędnej, która redefiniuje metody przeciążania operatorów raz, za pomocą standardowego kodu delegacji. Co więcej, narzędzi takie, jak dodatkowe dekoratory klas bądź metaklasy mogą zautomatyzować część pracy związanej z dodawaniem metod tego typu do klas delegacji (po szczegóły warto sięgnąć do przykładów rozszerzania z rozdziału 39.). Choć nadal nie jest to tak proste jak rozwiązywanie przeznaczone dla Pythona 2.6, takie techniki mogą nam pomóc uczynić klasy delegacji z wersji 3.0 bardziej uniwersalnymi.

Alternatywy implementacyjne: wstawianie `__getattribute__`, inspekcja stosu wywołań

Choć powtarzanie definicji metod przeciążania operatorów w obiektach opakowujących jest najprawdopodobniej najprostszym obejściem problemu z Pythonem 3.0 zarysowanego w tym podrozdziale, niekoniecznie musi to być jedynie rozwiązanie. Nie mamy tutaj miejsca na dalsze omawianie tej kwestii, dlatego zbadanie innych potencjalnych rozwiązań pozostawiam jako sugerowane ćwiczenie. Ponieważ jednak jedno z rozwiązań dobrze opisuje ogólne koncepcje związane z klasami, zasługuje na krótkie omówienie.

Jedną z wad tego przykładu jest to, że obiekty instancji nie są tak naprawdęinstancjami oryginalnej klasy — są zamiast tego instancjami klasy opakowującej. W niektórych programach polegających na sprawdzaniu typów może to mieć znaczenie. By obsługiwać tego typu przypadki, możemy spróbować osiągnąć ten sam efekt, *wstawiając* metodę `__getattribute__` do oryginalnej klasy w celu przechwycenia wszystkich referencji do atrybutów wykonywanych na instancjach. Wstawiona metoda przekazałaby poprawne żądania w górę do klasy nadrzędnej w celu uniknięcia pętli za pomocą technik omówionych w poprzednim rozdziale. Oto potencjalna zmiana, jaką należy wprowadzić do kodu naszego dekoratora klasy.

Obsługa śledzenia jak wcześniej

```
def accessControl(failIf):
    def onDecorator(aClass):
        def getattributes(self, attr):
            trace('pobranie:', attr)
            if failIf(attr):
                raise TypeError('pobranie atrybutu prywatnego: ' + attr)
            else:
                return object.__getattribute__(self, attr)
        aClass.__getattribute__ = getattributes
        return aClass
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))
```

Powyższa alternatywa rozwiązuje problem sprawdzania typów, jednak ma inne ograniczenia. Przykładowo obsługuje ona jedynie *pobieranie* atrybutów — w obecnej wersji pozwala na swobodne *przypisywanie* zmiennych prywatnych. Przechwytywanie prób przypisania nadal musiałoby korzystać z metody `__setattr__` i albo obiektu opakowującego instancję, albo innej

metody wstawiania klasy. Dodanie obiektu opakowującego instancję w celu przechwycenia przypisań ponownie zmieniłoby typ, natomiast wstawienie metod zawiedzie, jeśli oryginalna klasa wykorzystuje własną metodę `__setattr__` (lub `__getattribute__`, skoro o tym mowa!). Wstawiona metoda `__setattr__` musiałaby także pozwolić na użycie `__slots__` w klasie klienta.

Dodatkowo rozwiązanie to nie rozwiązuje problemu atrybutów operacji *wbudowanych*, omówionego w poprzednim podrozdziale, ponieważ metoda `__getattribute__` nie działa w tych kontekstach. W naszym przypadku, gdyby klasa `Person` miała metodę `__str__`, zostałaby ona wykonana przez operacje wyświetlanego, jednak tylko dlatego, że byłaby obecna w tej klasie. Tak jak wcześniej, atrybut `__str__` nie zostałby przekierowany do wstawionej metody `__getattribute__` w sposób uniwersalny — wyświetlanie całkowicie obeszłoby tę metodę i powodowałoby bezpośrednie wywołanie metody `__str__` klasy.

Choć i tak jest to najprawdopodobniej lepsze od całkowitego braku obsługi metod przeciążających operatory w opakowanym obiekcie (pomijając ich redefinicję), rozwiązanie to nadal nie jest w stanie przechwytywać i sprawdzać metod `__X__`, przez co żadna z nich nie może być prywatna. Choć większość metod przeciążania operatorów z założenia ma być publiczna, niektóre wcale nie muszą takie być.

Co gorsza, ponieważ rozwiązanie bez opakowywania działa dzięki dodaniu metody `__getattribute__` do udekorowanej klasy, przechwytuje również próby dostępu do atrybutów wykonywane przez samą klasę i sprawdza je w ten sam sposób jak próby dostępu wykonywane z zewnątrz — co oznacza, że metody klasy również nie będą w stanie wykorzystywać zmiennych prywatnych!

Tak naprawdę wstawienie metod w ten sposób jest funkcjonalnym odpowiednikiem ich *dziediczenia* i wiąże się z tymi samymi ograniczeniami co kod tworzący prywatność atrybutów z rozdziału 29. By dowiedzieć się, czy próba dostępu odbywa się z wewnętrz klasy, czy z zewnątrz, nasza metoda może być zmuszona sprawdzać obiekty ramek lub *stos wywołań* Pythona. Może to prowadzić do pewnego rozwiązania problemu (na przykład zastąpienia atrybutów prywatnych właściwościami lub deskryptorami sprawdzającymi *stos*), ale w dalszym ciągu spowalniałoby dostęp i jest zbyt zagmatwane, byśmy to tutaj omawiali.

Choć technika wstawiania metod jest interesująca i może mieć znaczenie w innych przypadkach użycia, nie sprostała jednak naszym celom. Nie będziemy dalej zajmować się tym wzorcem kodu, ponieważ w kolejnym rozdziale techniki rozszerzania klas omówimy w połączeniu z metaklasami. Jak zobaczymy, metaklasy nie są ściśle wymagane do tego typu modyfikacji klas, ponieważ tę samą rolę mogą często pełnić dekoratory klas.

W Pythonie nie chodzi o kontrolę

Skoro już podjąłem tak wielki wysiłek w celu dodania deklaracji atrybutów jako prywatnych i publicznych do kodu napisanego w Pythonie, muszę raz jeszcze przypomnieć, że dodawanie tego typu kontroli dostępu do klas nie jest w pełni *pythonowym* sposobem działania. Tak naprawdę większość programistów Pythona uzna najprawdopodobniej ten przykład za w dużej mierze (lub całkowicie) zbędny — poza pełnieniem roli demonstracji działania dekoratorów w praktyce. Większość dużych programów w Pythonie świetnie sobie radzi bez żadnej tego typu kontroli. Jeśli jednak chcemy regulować dostęp w celu wyeliminowania błędów programistycznych lub jesteśmy prawie byłym programistą C++ lub Javy, większość zadań można zrealizować za pomocą przeciążania operatorów i narzędzi do introspekcji Pythona.

Przykład: Sprawdzanie poprawności argumentów funkcji

W ostatnim przykładzie użyteczności dekoratorów w niniejszym podrozdziale napiszemy *dekorator funkcji*, który automatycznie sprawdza, czy argumenty przekazane do funkcji bądź metody mieszczą się w określonym przedziale liczbowym. Został on zaprojektowany w celu użycia na etapie programowania lub produkcji i można go wykorzystać jako szablon dla innych, podobnych zadań (na przykład sprawdzania typów argumentów, jeśli musimy je wykonywać). Ponieważ przekroczyliśmy już limit objętości dla tego rozdziału, kod z tego przykładu będzie w dużej mierze materiałem do samodzielnego przestudiowania, z ograniczoną częścią opisową. Jak zwykle więcej szczegółów można znaleźć, przeglądając sam kod.

Cel

W omówieniu programowania zorientowanego obiektowo w rozdziale 27. napisaliśmy klasę, która przyznawała podwyżkę obiektom reprezentującym ludzi w oparciu o przekazaną wartość procentową.

```
class Person:  
    ...  
    def giveRaise(self, percent):  
        self.pay = int(self.pay * (1 + percent))
```

Zauważaliśmy tam, że gdybyśmy chcieli, by kod był bardziej rozbudowany, niezłym pomysłem byłoby sprawdzenie, czy wartość procentowa nie jest zbyt duża lub zbyt mała. Moglibyśmy wprowadzić tego typu sprawdzanie za pomocą instrukcji `if` lub `assert wstawionych` do samej metody.

```
class Person:  
    def giveRaise(self, percent):          # Sprawdzenie w kodzie metody  
        if percent < 0.0 or percent > 1.0:  
            raise TypeError, 'niepoprawna wartość procentowa'  
        self.pay = int(self.pay * (1 + percent))  
  
class Person:                         # Sprawdzenie za pomocą instrukcji assert  
    def giveRaise(self, percent):  
        assert percent >= 0.0 and percent <= 1.0, 'niepoprawna wartość procentowa'  
        self.pay = int(self.pay * (1 + percent))
```

Takie rozwiązanie zanieczyszcza jednak metodę testami wewnętrznymi, które najprawdopodobniej przydadzą się jedynie w trakcie pisania programu. W bardziej skomplikowanych przypadkach może się to stać dość żmudne (wyobraźmy sobie próby wstawiania kodu potrzebnego do zaimplementowania prywatności atrybutów z dekoratora z poprzedniego podrozdziału). Co jednak gorsze, jeśli logika sprawdzająca będzie się musiała kiedykolwiek zmienić, potencjalnie będziemy musieli odnaleźć i uaktualnić dowolnie dużą liczbę wierszy.

Ciekawszą i bardziej przydatną alternatywą byłoby napisanie narzędzia ogólnego przeznaczenia, które jest w stanie automatycznie wykonywać dla nas testy przedziałów dla argumentów dowolnej funkcji bądź metody, jaką utworzymy teraz lub w przyszłości. Rozwiązanie z *dekoratorem* sprawia, że będzie się to odbywało w sposób spójny i jawnny.

```
class Person:  
    @rangetest(percent=(0.0, 1.0))          # Użycie dekoratora do sprawdzania  
    def giveRaise(self, percent):  
        self.pay = int(self.pay * (1 + percent))
```

Wyizolowanie logiki sprawdzającej poprawność w dekoratorze upraszcza zarówno kod klienta, jak i późniejsze utrzymywanie.

Warto zauważyć, że nasz cel jest tutaj inny od sytuacji ze sprawdzaniem poprawności atrybutów z ostatniego przykładu z poprzedniego rozdziału. Tutaj chcemy sprawdzić wartości *argumentów funkcji* przy przekazywaniu, a nie *wartości atrybutów* przy ustawianiu. Dekorator Pythona wraz z narzędziami do introspekcji pozwolą nam wdrożyć to nowe zadanie równie łatwo.

Prosty dekorator sprawdzający przedziały dla argumentów pozycyjnych

Zacznijmy od prostej implementacji testu przedziału. Dla uproszczenia zaczniemy od napisania kodu dekoratora, który działa jedynie dla argumentów pozycyjnych i zakłada, że zawsze, w każdym wywołaniu pojawiają się one na tej samej pozycji. Nie mogą być przekazane po nazwie słowa kluczowego i nie obsługujemy dodatkowych słów kluczowych `**args` w wywołaniach, ponieważ może to zmienić pozycje zadeklarowane w dekoratorze. Zapiszmy poniższy kod w pliku o nazwie `devtools.py`:

```
def rangetest(*argchecks):
    def onDecorator(func):
        if not __debug__:
            return func
        else:
            def onCall(*args):
                for (ix, low, high) in argchecks:
                    if args[ix] < low or args[ix] > high:
                        errmsg = 'Argument %s nie mieści się w przedziale %s..%s' %
                                (ix, low, high)
                        raise TypeError(errmsg)
                return func(*args)
            return onCall
    return onDecorator
```

Sprawdzenie przedziałów argumentów pozycyjnych
True, jeśli "python -O main.py args..."
Nie działa: bezpośrednie wywołanie oryginału
Inaczej opakowanie w czasie debugowania

W obecnej postaci powyższy kod jest właściwie powtórzeniem omówionych wcześniej wzorców kodu. Wykorzystujemy między innymi argumenty dekoratora i zagnieżdżone zakresy na potrzeby zachowywania stanu.

Używamy również zagnieżdżonych instrukcji `def`, by przykład działał zarówno dla prostych funkcji, jak i dla metod, zgodnie z tym, czego nauczyliśmy się wcześniej. Po wykorzystaniu dla metody klasy `onCall` otrzymujemy instancję klasy podmiotowej w pierwszym elemencie `*args` i przekazuje ją dalej do `self` w oryginalnej funkcji metody. Liczby dla argumentów w testach przedziałów zaczynają się w tym przypadku od 1, a nie od 0.

Warto również zauważyć, że kod ten wykorzystuje wbudowaną zmienną `__debug__`. Python ustawia ją na `True`, o ile nie wykonujemy kodu z ustawioną opcją wiersza poleceń `-O` oznaczającą optymalizację (na przykład `python -O main.py`). Kiedy zmienna `__debug__` zwraca `False`, dekorator zwraca oryginalną funkcję bez zmian w celu uniknięcia dodatkowych wywołań i związanego z tym negatywnego wpływu na wydajność.

Pierwsza wersja naszego rozwiązania będzie wykorzystywana w następujący sposób:

```
# Plik devtools_test.py

from devtools import rangetest
print(__debug__)
```

False, jeśli "python -O main.py"

```

@rangetest((1, 0, 120))                                # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):                               # age musi się mieścić w przedziale 0..120
    print('%s ma %s lat' % (name, age))

@rangetest([0, 1, 31], [1, 1, 12], [2, 0, 2009])
def birthday(D, M, Y):
    print('Data urodzenia = {0}/{1}/{2}'.format(D, M, Y))

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay

    @rangetest([1, 0.0, 1.0])                         # giveRaise = rangetest(...)(giveRaise)
    def giveRaise(self, percent):                      # Argument 0 to tutaj instancja self
        self.pay = int(self.pay * (1 + percent))

# Wiersze z komentarzem zgłaszą błąd TypeError, o ile w wierszu polecień powłoki nie wykorzystano "python -O"

persinfo('Robert Zielony', 45)                        # Tak naprawdę wykonuje onCall(...) ze stanem
#persinfo('Robert Zielony', 200)                      # Lub person, jeśli ustalono argument wiersza polecień -O

birthday(31, 5, 1963)
#birthday(32, 5, 1963)

anna = Person('Anna Czerwona', 'programista', 100000)
anna.giveRaise(.10)                                     # Tak naprawdę wykonuje onCall(self, .10)
print(anna.pay)                                       # Lub giveRaise(self, .10), jeśli z -O
#anna.giveRaise(1.10)
#print(anna.pay)

```

Po wykonaniu poprawne wywołania w powyższym kodzie zwracają następujące wyniki. Cały kod z tego podrozdziału działa tak samo w Pythonie 2.6 oraz 3.0, ponieważ dekoratory funkcji obsługiwane są w obu wersjach, nie korzystamy z delegacji atrybutów i używamy wywołań `print` oraz składni wyjątków z wersji 3.0.

```
C:\misc> C:\python30\python devtools_test.py
True
Robert Zielony ma 45 lat
Data urodzenia = 31/5/1963
110000
```

Usunięcie komentarzy z któregoś z niepoprawnych wywołań powoduje zgłoszenie błędu `TypeError` przez dekorator. Oto wynik po pozwoleniu na wykonanie dwóch ostatnich wierszy (jak zwykle pominąłem część tekstu komunikatu o błędzie w celu zaoszczędzenia miejsca).

```
C:\misc> C:\python30\python devtools_test.py
True
Robert Zielony ma 45 lat
Data urodzenia = 31/5/1963
110000
TypeError: Argument 1 nie mieści się w przedziale 0.0..1.0
```

Uruchomienie Pythona z opcją wiersza polecień `-O` wyłączy sprawdzanie przedziałów, jednak pozwoli także uniknąć pogorszenia wydajności związanego z warstwą opakowującą. W rezultacie wywołujemy oryginalną, nieudekorowaną funkcję w sposób bezpośredni. Zakładając, że narzędzie to służy nam tylko do debugowania, możemy wykorzystać tę opcję do zoptymalizowania programu przed użyciem go w środowisku produkcyjnym.

```
C:\misc> C:\python30\python -O devtools_test.py
False
Robert Zielony ma 45 lat
```

```
Data urodzenia = 31/5/1963
110000
231000
```

Uogólnienie kodu pod kątem słów kluczowych i wartości domyślnych

Poprzednia wersja kodu ilustruje podstawy, z których musimy skorzystać, jednak jest stosunkowo ograniczona. Obsługuje jedynie sprawdzanie poprawności argumentów przekazanych po pozycji i nie sprawdza argumentów ze słowami kluczowymi. Tak naprawdę zakłada, że żadne słowa kluczowe nie będą przekazywane w sposób zakłócający pozycję argumentów. Dodatkowo nie robi nic z argumentami z wartościami domyślnymi, które można pominąć w wywołaniu. Takie rozwiązanie będzie w porządku, jeśli wszystkie argumenty przekazywane są po pozycji i nigdy nie mają wartości domyślnych, ale jest to dalekie od ideału w przypadku narędzia ogólnego przeznaczenia. Python obsługuje o wiele bardziej elastyczne tryby przekazywania argumentów, które nasz kod na razie pomija.

Odmiana naszego przykładu zaprezentowana poniżej radzi sobie nieco lepiej. Dopasowując oczekiwane argumenty opakowanej funkcji do prawdziwych argumentów przekazanych w wywołaniu, pozwala obsługiwać sprawdzanie poprawności przedziałów dla argumentów przekazywanych albo po pozycji, albo za pomocą słowa kluczowego. Pomijane jest testowanie argumentów domyślnych pominiętych w wywołaniu. Mówiąc w skrócie, argumenty do sprawdzenia są określane przez słowa kluczowe dla dekoratora, który później przechodzi zarówno krótką pozycyjną `*pargs`, jak i słownikków słów kluczowych `**kargs` w celu sprawdzenia poprawności.

```
"""
```

Plik `devtools.py`

Dekorator funkcji wykonujący sprawdzanie poprawności przedziałów dla przekazanych argumentów. Argumenty dla dekoratora określane są po słowach kluczowych. W samym wywołaniu argumenty mogą być przekazywane po pozycji lub za pomocą słowa kluczowego. Wartości domyślne mogą być pomijane.

Przykładowe przypadki użycia znajdują się w pliku `devtools_test.py`.

```
"""
```

```
trace = True

def rangetest(**argchecks):
    def onDecorator(func):
        if not __debug__:
            return func
        else:
            import sys
            code = func.__code__
            allargs = code.co_varnames[:code.co_argcount]
            funcname = func.__name__

    def onCall(*pargs, **kargs):
        # Wszystkie argumenty pozycyjne pargs dopasowują pierwsze N oczekiwanych argumentów po pozycji
        # Reszta musi być w kargs lub jest pomijanymi wartościami domyślnymi
        positionals = list(allargs)
        positionals = positionals[:len(pargs)]

        for argname, (low, high) in argchecks.items():
            # Dla wszystkich argumentów, które mają być sprawdzone
            if argname in kargs:
                # Przekazane po nazwie
```

```

if kargs[argname] < low or kargs[argname] > high:
    errmsg = '{0} argument "{1}" nie mieści się w przedziale {2}..{3}'
    errmsg = errmsg.format(funcname, argname, low, high)
    raise TypeError(errmsg)

elif argname in positionals:
    # Przekazane po pozycji
    position = positionals.index(argname)
    if pargs[position] < low or pargs[position] > high:
        errmsg = '{0} argument "{1}" nie mieści się w przedziale {2}..{3}'
        errmsg = errmsg.format(funcname, argname, low, high)
        raise TypeError(errmsg)

else:
    # Zakładamy, że nie został przekazany: wartość domyślna
    if trace:
        print('Argument "{0}" ma wartość domyślną'.format(argname))
    return func(*pargs, **kargs) # OK: wykonanie oryginalnego wywołania
return onCall
return onDecorator

```

Poniższy skrypt testowy pokazuje, w jaki sposób wykorzystywany jest dekorator. Argumenty do sprawdzenia przekazywane są jako argumenty dekoratora ze słowami kluczowymi, natomiast w samym wywołaniu możemy przekazać je albo po nazwie, albo po pozycji i pominąć argumenty z wartościami domyślnymi, nawet jeśli mają one być sprawdzane w innym sposobie.

```

# Plik devtools_test.py
# Wiersze z komentarzami zgłaszają błąd TypeError, o ile nie ustawiono "python -O" w wierszu poleceń powłoki.
from devtools import rangetest

```

Testowanie funkcji, pozycyjne i po słowach kluczowych

```

@rangetest(age=(0, 120)) # persinfo = rangetest(...)(persinfo)
def persinfo(name, age):
    print('%s ma %s lat' % (name, age))

@rangetest(D=(1, 31), M=(1, 12), Y=(0, 2009))
def birthday(D, M, Y):
    print('birthday = {0}/{1}/{2}'.format(D, M, Y))

persinfo('Robert', 40)
persinfo(age=40, name='Robert')
birthday(1, M=5, Y=1963)
#persinfo('Robert', 150)
#persinfo(age=150, name='Robert')
#birthday(40, M=5, Y=1963)

```

Testowanie metod, pozycyjne i po słowach kluczowych

```

class Person:
    def __init__(self, name, job, pay):
        self.job = job
        self.pay = pay # giveRaise = rangetest(...)(giveRaise)
    @rangetest(percent=(0.0, 1.0)) # Wartość percent przekazana po nazwie lub pozycji
    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

bob = Person('Robert Zielony', 'programista', 100000)
anna = Person('Anna Czerwona', 'programista', 100000)
bob.giveRaise(.10)
anna.giveRaise(percent=.20)
print(bob.pay, anna.pay)

```

```

#bob.giveRaise(1.10)
#bob.giveRaise(percent=1.20)

# Testowanie pominiętych wartości domyślnych: pominięte

@rangetest(a=(1, 10), b=(1, 10), c=(1, 10), d=(1, 10))
def omitargs(a, b=7, c=8, d=9):
    print(a, b, c, d)

omitargs(1, 2, 3, 4)                                # Niepoprawne d
omitargs(1, 2, 3)                                   # Niepoprawne c
omitargs(1, 2, 3, d=4)                             # Niepoprawne d
omitargs(1, d=4)                                    # Niepoprawne a
omitargs(d=4, a=1)                                 # Niepoprawne a
omitargs(1, b=2, d=4)                             # Niepoprawne b
omitargs(d=8, c=7, a=1)                           # Niepoprawne a

```

Po wykonaniu powyższego skryptu argumenty niemieszczące się w przedziałach tak jak wcześniej zgłaszą wyjątki, jednak możemy je przekazywać po nazwie bądź pozycji, a pominięte wartości domyślne nie są sprawdzane. Kod działa w Pythonie 2.6 oraz 3.0, jednak w wersji 2.6 wyświetlane są dodatkowe nawiasy krotek. Warto prześledzić dane wyjściowe i kontynuować testy na własną rękę w celach eksperymentalnych. Kod działa jak wcześniej, jednak jego zakres został rozszerzony.

```

C:\misc> C:\python30\python devtools_test.py
Robert ma 40 lat
Robert ma 40 lat
Data urodzenia = 1/5/1963
110000 120000
1 2 3 4
Argument "d" ma wartość domyślną
1 2 3 9
1 2 3 4
Argument "c" ma wartość domyślną
Argument "b" ma wartość domyślną
1 7 8 4
Argument "c" ma wartość domyślną
Argument "b" ma wartość domyślną
1 7 8 4
Argument "c" ma wartość domyślną
1 2 8 4
Argument "b" ma wartość domyślną
1 7 7 8

```

W przypadku błędów w sprawdzaniu otrzymujemy, jak poprzednio, wyjątek (o ile do Pythona nie przekazano argumentu wiersza poleceń -O), jeśli jeden z wierszy testowania metod zostanie wyjęty z komentarza.

```
TypeError: giveRaise argument "percent" nie mieści się w przedziale 0.0..1.0
```

Szczegóły implementacji

Kod tego dekoratora opiera się zarówno na API introspekcji, jak i subtelnych ograniczeniach przekazywania argumentów. By go w pełni uogólnić, powinniśmy spróbować naśladować pełną logikę dopasowywania argumentów Pythona w celu sprawdzenia, które nazwy zostaną przekazane w których trybach, jednak jest to zbyt wysoki stopień skomplikowania jak na nasze narzędzie. Lepiej byłoby, gdybyśmy mogli jakoś argumenty przekazane po nazwie dopasować do zbioru wszystkich oczekiwanych nazw argumentów w celu ustalenia, na jakiej pozycji pojawiają się one w określonym wywołaniu.

Dalsza introspekcja

Okazuje się, że API do introspekcji dostępne dla obiektów funkcji oraz powiązanych z nimi obiektów kodu ma do dyspozycji narzędzie, którego potrzebujemy. API to zostało krótko wprowadzone w rozdziale 19., jednak teraz możemy je wykorzystać. Zbiór oczekiwanych nazw argumentów to po prostu pierwszych N nazw zmiennych dołączonych do obiektu kodu funkcji.

```
# W Pythonie 3.0 (oraz 2.6 dla zgodności):
>>> def func(a, b, c, d):
...     x = 1
...     y = 2
...
...>>> code = func.__code__                                # Obiekt kodu obiektu funkcji
>>> code.co_nlocals
6
>>> code.co_varnames                                  # Nazwy wszystkich zmiennych lokalnych
('a', 'b', 'c', 'd', 'x', 'y')
>>> code.co_varnames[:code.co_argcount]               # Pierwsze N zmiennych lokalnych to oczekiwane argumenty
('a', 'b', 'c', 'd')

>>> import sys                                         # Dla zgodności z poprzednimi wersjami
>>> sys.version_info                                  # [0] to numer dużego wydania
(3, 0, 0, 'final', 0)
>>> code = func.__code__ if sys.version_info[0] == 3 else func.func_code
```

To samo API dostępne jest również w starszych wersjach Pythona, jednak atrybut `func.__code__` zapisywany jest jako `func.func_code` w wersji 2.5 oraz wcześniejszych (nowszy atrybut `__code__` jest również dostępny w wersji 2.6 z uwagi na przenośność). Więcej informacji można otrzymać po wykonaniu wywołania funkcji `dir` na obiektach funkcji oraz kodu.

Założenia dotyczące argumentów

Gdy mamy zbiór oczekiwanych nazw argumentów, rozwiązanie opiera się na dwóch ograniczeniach w zakresie kolejności ich przekazywania, narzuconych przez Pythona (w wersjach 2.6 oraz 3.0 nadal są one obowiązujące):

- W wywołaniu wszystkie argumenty pozycyjne pojawiają się przed wszystkimi argumentami ze słowami kluczowymi.
- W instrukcji `def` wszystkie argumenty bez wartości domyślnej pojawiają się przed wszystkimi argumentami z wartościami domyślnymi.

Oznacza to, że argument niebędący słowem kluczowym nie może w wywołaniu pojawić się po argumencie ze słowem kluczowym, natomiast argument bez wartości domyślnej nie może się pojawić po argumencie z wartością domyślną w definicji funkcji. Cała składnia `nazwa=wartość` musi się w obu miejscach pojawiać po wszystkich zwykłych argumentach typu `nazwa`.

W celu uproszczenia sobie pracy możemy także założyć, że wywołanie jest ogólnie poprawne, to znaczy wszystkie argumenty albo otrzymają wartości (po nazwie bądź pozycji), albo zostaną celowo pominięte w celu pobrania wartości domyślnych. Takie założenie niekoniecznie będzie prawdziwe, ponieważ funkcja nie zostaje jeszcze wywołana, zanim logika opakowująca sprawdzi poprawność. Wywołanie może nadal się nie powieść później, po uruchomieniu za pośrednictwem warstwy opakowującej, z uwagi na niepoprawne przekazanie argumentów. Dopóki jednak nie będzie to powodowało jeszcze większego błędu w warstwie opakowującej, będziemy mogli pracować nad szczegółami poprawności wywołania. Jest to dla nas pomocne, ponieważ sprawdzanie poprawności wywołań przed ich wykonaniem wymagałoby pełnej emulacji algorytmu dopasowywania argumentów Pythona — co jest zbyt skomplikowaną procedurą jak na nasze narzędzie.

Algorytm dopasowywania

Po przyjęciu powyższych założeń i poznaniu ograniczeń możemy teraz w naszym algorytmie zarówno pozwolić na użycie argumentów ze słowami kluczowymi, jak i pomijanie argumentów z wartościami domyślnymi w wywołaniu. Kiedy wywołanie zostaje przechwycone, możemy poczynić następujące założenia:

- Wszystkie N przekazanych argumentów pozycyjnych z `*args` muszą odpowiadać pierwszym N oczekiwanych argumentów pozyskanych z obiektu kodu funkcji. Jest to prawdziwe zgodnie z przedstawionymi wcześniej regułami kolejności wywołań Pythona, ponieważ wszystkie argumenty pozycyjne pojawiają się przed wszystkimi argumentami ze słowami kluczowymi.
- W celu uzyskania nazw argumentów przekazanych po pozycji musimy wykonać wycinek z listy wszystkich oczekiwanych argumentów aż do długości N krótki pozycyjnej `*args`.
- Wszystkie argumenty po pierwszych N oczekiwanych argumentach zostały albo przekazane po słowie kluczowym, albo pominięte w czasie wywołania i przyjmują wartości domyślne.
- Dla każdej nazwy argumentu do sprawdzenia, jeśli znajduje się ona w `**kwargs`, argument został przekazany po nazwie, natomiast jeśli znajduje się ona w pierwszych N oczekiwanych argumentach, został przekazany po pozycji (w którym to przypadku jego względna pozycja na liście oczekiwanych argumentów określa jego względową pozycję w krotce `*args`).

W przeciwnym razie możemy założyć, że argument został pominięty w wywołaniu, ma wartość domyślną i nie musi być sprawdzany. Innymi słowy, możemy pominąć testy dla argumentów, które zostały pominięte w wywołaniu, zakładając, że pierwszych N faktycznie przekazanych argumentów pozycyjnych z `*args` musi odpowiadać pierwszym N nazw argumentów z listy wszystkich oczekiwanych argumentów, a wszystkie pozostałe albo musiały być przekazane po słowach kluczowych (i tym samym znajdować się w `**kwargs`), albo przybierają wartości domyślne. W tym rozwiązaniu dekorator po prostu pominie wszelkie argumenty do sprawdzenia pominięte pomiędzy znajdująącym się najbardziej na prawo argumentem pozycyjnym a znajdująjącym się najbardziej na lewo argumentem ze słowem kluczowym, pomiędzy argumentami ze słowami kluczowymi lub ogólnie po argumencie pozycyjnym znajdującym się najbardziej na prawo. By przekonać się, w jaki sposób odbywa się to w kodzie, warto prześledzić kod dekoratora oraz jego skryptu testowego.

Znane problemy

Choć nasze narzędzie sprawdzające przedziały działa zgodnie z planem, pozostają dwie kwestie. Po pierwsze, jak wspomniano wcześniej, wywołania oryginalnej funkcji, które nie są poprawne, nadal nie powiodą się w ostatecznej wersji dekoratora. Przykładowo oba poniższe wywołania zwracają wyjątki:

```
omitargs()  
omitargs(d=8, c=7, b=6)
```

Nie powiodą się one jednak tylko tam, gdzie próbujemy wywołać oryginalną funkcję — na końcu obiektu opakowującego. Choć moglibyśmy próbować imitować mechanizm dopasowywania argumentów Pythona w celu uniknięcia tej sytuacji, nie za bardzo jest powód, by to robić — ponieważ wywołanie na tym etapie i tak by się nie powiodło, możemy również dobrze pozwolić własnej logice dopasowywania argumentów Pythona na wykrycie tego problemu za nas.

Na koniec, choć nasza ostateczna wersja kodu obsługuje argumenty pozycyjne, argumenty ze słowami kluczowymi oraz pominięte wartości domyślne, nadal nie robi nic z `*args` oraz `**args`, które mogą zostać użyte w udekorowanej funkcji przyjmującej dowolną liczbę argumentów. Najprawdopodobniej nie będziemy się tym jednak musieli przejmować w naszej sytuacji:

- Jeśli dodatkowy argument ze słowem *kluczowym* zostanie przekazany, jego nazwa pokaże się w słowniku `**kwargs` i zostanie on normalnie przetestowany, o ile zostanie wspomniany w dekoratorze.
- Jeśli dodatkowy argument ze słowem *kluczowym* *nie* zostanie przekazany, jego nazwy nie znajdziemy ani w słowniku `**kwargs`, ani w wycinku oczekiwanych argumentów pozycyjnych, dlatego nie zostanie on sprawdzony. Zostanie potraktowany tak, jakby miał mieć wartość domyślną, nawet jeśli tak naprawdę jest to dodatkowy argument opcjonalny.
- Jeśli przekazany zostanie dodatkowy argument *pozycyjny*, i tak nie można się do niego w żaden sposób odwołać w dekoratorze — jego nazwy nie będzie ani w słowniku `**kwargs`, ani w wycinku listy oczekiwanych argumentów, dlatego zostanie po prostu pominięty. Ponieważ takie argumenty nie są wymienione w definicji funkcji, nie można odwzorować nazwy podanej do dekoratora z powrotem na oczekiwanaą względową pozycję.

Innymi słowy, w obecnej postaci kod obsługuje sprawdzanie dowolnych argumentów ze słowami kluczowymi po nazwie, ale już nie dowolnych argumentów pozycyjnych, które pozostały bez nazwy i tym samym nie mają ustalonej pozycji w sygnaturze argumentu funkcji.

Co do zasady moglibyśmy rozszerzyć interfejs dekoratora, tak by obsługiwał on `*args` w udekorowanej funkcji na potrzeby rzadkich przypadków, w których może się to przydać (na przykład specjalna nazwa argumentu z testem do zastosowania do wszystkich argumentów w krotce `*pargs` obiektu opakowującego wykraczających poza długość listy oczekiwanych argumentów). Ponieważ jednak wyczerpaliśmy już miejsce przeznaczone na ten przykład, osoby, które mają ochotę zająć się takimi ulepszeniami, oficjalnie kierujemy do krainy ćwiczeń sugerowanych.

Argumenty dekoratora a adnotacje funkcji

Co ciekawe, opcja adnotacji funkcji wprowadzona w Pythonie 3.0 mogłaby stanowić alternatywę dla argumentów dekoratora wykorzystywanych w naszym przykładzie do określania testów przedziałów. Jak wiemy z rozdziału 19., adnotacje pozwalają nam wiązać wyrażenia z argumentami i zwracanymi wartościami poprzez zapisanie ich w kodzie w samym wierszu nagłówka instrukcji `def`. Python zbiera adnotacje w słownik i dołącza je do opisanej w ten sposób funkcji.

Moglibyśmy wykorzystać to w naszym przykładzie do zapisania granic przedziałów w wierszu nagłówka zamiast w argumentach dekoratora. Nadal potrzebowalibyśmy dekoratora funkcji do opakowania funkcji w celu przechwytcenia póżniejszych wywołań, jednak zamienilibyśmy następującą składnię argumentów dekoratora:

```
@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                      # func = rangetest(...)(func)
    print(a + b + c)
```

na poniższą składnię adnotacji:

```
@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):
    print(a + b + c)
```

Oznacza to, że granice przedziałów zostałyby przesunięte do samej funkcji, a nie pozostałyby w kodzie poza nią. Poniższy skrypt ilustruje strukturę wynikowych dekoratorów w obu rozwiązaniach, w niekompletnym szkielecie kodu. Wzorzec z argumentami dekoratorów pochodzi z zaprezentowanego wcześniej pełnego rozwiązania. Alternatywa z adnotacjami wymaga o jeden poziom zagnieżdżenia mniej, ponieważ nie musi zachowywać argumentów dekoratorów.

Wykorzystanie argumentów dekoratora

```
def rangetest(**argchecks):
    def onDecorator(func):
        def onCall(*pargs, **kargs):
            print(argchecks)
            for check in argchecks: pass
            return func(*pargs, **kargs)
        return onCall
    return onDecorator

@rangetest(a=(1, 5), c=(0.0, 1.0))
def func(a, b, c):                      # func = rangetest(...)(func)
    print(a + b + c)

func(1, 2, c=3)                         # Wykonuje onCall, argchecks w zakresie
```

Wykorzystanie adnotacji funkcji

```
def rangetest(func):
    def onCall(*pargs, **kargs):
        argchecks = func.__annotations__
        print(argchecks)
        for check in argchecks: pass
        return func(*pargs, **kargs)
    return onCall

@rangetest
def func(a:(1, 5), b, c:(0.0, 1.0)):    # func = rangetest(func)
    print(a + b + c)

func(1, 2, c=3)                         # Wykonuje onCall, adnotacje w funkcji
```

Po wykonaniu oba rozwiązania mają dostęp do tych samych informacji testu sprawdzającego, jednak w innych formach. Informacje z wersji z argumentami dekoratora zachowywane są w argumencie w zakresie zawierającym, natomiast informacje z wersji z adnotacją zachowywane są w atrybutie samej funkcji.

```
{'a': (1, 5), 'c': (0.0, 1.0)}  
6  
{'a': (1, 5), 'c': (0.0, 1.0)}  
6
```

Utworzenie reszty wersji opartej na adnotacji pozostawiam jako sugerowane ćwiczenie; jej kod byłby identyczny z kodem zaprezentowanego wcześniej pełnego rozwiązania, ponieważ informacje z testów przedziałów znajdują się po prostu w funkcji zamiast w zakresie zawierającym. Tak naprawdę rozwiązanie to daje nam inny interfejs użytkownika dla naszego narzędzia — nadal, jak wcześniej, będzie ono musiało dopasowywać nazwy argumentów do nazw oczekiwanych argumentów w celu uzyskania ich względnych pozycji.

Właściwie użycie adnotacji w miejsce argumentów dekoratora w tym przykładzie tak naprawdę ogranicza jego przydatność. Po pierwsze, adnotacje działają jedynie w Pythonie 3.0, dlatego wersja 2.6 przestaje być obsługiwana. Z kolei dekoratory funkcji z argumentami działają w obu wersjach Pythona.

Co jednak bardziej istotne, przenosząc specyfikację sprawdzania poprawności do nagłówka, tak naprawdę ograniczamy funkcję do *jednej roli*. Ponieważ adnotacja pozwala nam na zapisanie w kodzie tylko jednego wyrażenia na argument, może pełnić tylko jeden cel. Przykładowo nie możemy wykorzystać adnotacji testów przedziałów do żadnej innej roli.

Z kolei, ponieważ argumenty dekoratora zapisywane są w kodzie poza samą funkcją, nie tylko łatwiej jest je usunąć, ale są one również *bardziej ogólne*. Kod samej funkcji nie wymusza jednego celu dekoracji. Tak naprawdę dzięki zagnieżdżeniu dekoratorów z argumentami możemy zastosować większą liczbę kroków rozszerzających do tej samej funkcji. Adnotacja w sposób bezpośredni obsługuje tylko jeden krok. Po zastosowaniu argumentów dekoratora sama funkcja zachowuje także prostszy, normalny wygląd.

Mimo to, jeśli mamy na myśli jeden konkretny cel i możemy się zdecydować na obsługę jedynie Pythona 3.0, wybór pomiędzy adnotacjami a argumentami dekoratora jest w dużej mierze subiektywny i sprowadza się do stylistyki. Jak to się często zdarza w życiu, adnotacje jednej osoby dla drugiej mogą być składniowym bałaganem...

Inne zastosowania — sprawdzanie typów (skoro nalegamy!)

Wzorzec kodu, na jakim skończyliśmy przetwarzanie argumentów w dekoratorach, można również zastosować w innych kontekstach. Sprawdzanie typów danych argumentów w czasie programowania jest na przykład prostym rozszerzeniem kodu.

```
def typecheck(**argchecks):  
    def onDecorator(func):  
        ...  
        def onCall(*pargs, **kargs):  
            positionals = list(allargs)[:len(pargs)]  
            for (argname, type) in argchecks.items():  
                if argname in kargs:  
                    if not isinstance(kargs[argname], type):  
                        ...  
                        raise TypeError(errmsg)
```

```

        elif argname in positionals:
            position = positionals.index(argname)
            if not isinstance(pargs[position], type):
                ...
                raise TypeError(errmsg)
            else:
                # Zakładamy, że nie przekazano: wartości domyślne
                return func(*pargs, **kargs)
        return onCall
    return onDecorator

@typetest(a=int, c=float)
def func(a, b, c, d):                      # func = typetest(...)(func)
    ...
    func(1, 2, 3.0, 4)                      # OK
    func('mielonka', 2, 99, 4)               # Poprawnie uruchamia wyjątek

```

Tak naprawdę moglibyśmy nawet uogólnić to narzędzie jeszcze bardziej, przekazując funkcję sprawdzającą, podobnie jak zrobiliśmy to wcześniej w dekoracjach atrybutów jako publicznych. Pojedyncza kopia tego typu kodu wystarczyłaby zarówno do sprawdzania przedziałów, jak i typów. Wykorzystanie adnotacji funkcji zamiast argumentów dekoratora dla takiego dekoratora, zgodnie z opisem z poprzedniego rozdziału, sprawiłoby, że kod w jeszcze większym stopniu przypominałby deklaracje typów z innych języków programowania.

```

@typetest
def func(a: int, b, c: float, d):          # func = typetest(func)
    ... # Ach!...

```

Czego jednak każdy powinien był się nauczyć z niniejszej książki, ta akurat rola jest ogólnie złym pomysłem w działającym kodzie, nie mówiąc o tym, że zupełnie nie jest w pythonowym stylu (często jest zresztą traktowana jako symptom pierwszych prób użycia Pythona przez byłego programistę języka C++).

Sprawdzanie typów ogranicza działanie funkcji do określonych typów, zamiast pozwalać jej działać na dowolnych typach ze zgodnymi interfejsami. W rezultacie ogranicza nasz kod i łamie jego elastyczność. Z drugiej strony, od każdej reguły są wyjątki — sprawdzanie typów może się przydać w wyizolowanych przypadkach w trakcie debugowania i pisania interfejsów dla kodu napisanego w bardziej restrykcyjnym języku, takim jak C++. Ten ogólny wzorzec przetwarzania argumentów może także mieć zastosowanie w różnych mniej kontrowersyjnych rolach.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy dekoratory w odmianach związanych z funkcjami oraz klasami. Zgodnie z tym, czego się nauczyliśmy, dekoratory są sposobem wstawiania kodu, który będzie wykonywany automatycznie przy definiowaniu funkcji bądź klasy. Kiedy dekorator jest wykorzystywany, Python ponownie dowiązuje nazwę funkcji bądź klasy do zwracanego przez niego obiektu wywowalnego. Ten punkt zaczepienia pozwala nam dodać warstwę logiki opakowującej do wywołań funkcji oraz wywołań tworzących instancje klas w celu zarządzania funkcjami iinstancjami.

Jak również widzieliśmy, ten sam efekt można osiągnąć za pomocą funkcji zarządzających oraz ponownego dowiązywania nazw, jednak dekoratory udostępniają rozwiązanie bardziej jednorodne i jawnie. Jak zobaczymy w kolejnym rozdziale, dekoratory klas można także

wykorzystywać do zarządzania samymi klasami, a nie tylko ich instancjami. Ponieważ ta funkcjonalność pokrywa się z metaklasami, tematem kolejnego rozdziału, dalsza część tej historii znajduje się nieco dalej. Najpierw jednak należy wykonać quiz kończący rozdział. Ponieważ rozdział ten skupiał się w przeważającej mierze na większych przykładach, w quizie Czytelnik zostanie poproszony o zmodyfikowanie części kodu w celu sprawdzenia go.

Sprawdź swoją wiedzę — quiz

1. Jak wspomniano w jednej ze wskazówek rozdziału, dekorator funkcji mierzącej czas, z argumentami dekoratora, napisany w podrozdziale zatytułowanym „Dodawanie argumentów dekoratora”, można zastosować jedynie do prostych *funkcji*, ponieważ wykorzystuje on zagnieżdżoną klasę z metodą przeciążania operatora `__call__` przechwytyującą wywołania. Struktura ta nie działa jednak dla *metod* klas, ponieważ do `self` przekazywana jest instancja dekoratora, a nie instancja klasy podmiotowej. Należy przepisać ten dekorator w taki sposób, by można go było zastosować zarówno do prostych funkcji, jak i metod klas, a następnie przetestować na funkcjach i metodach. (Uwaga: wskazówka należy szukać w podrozdziale zatytułowanym „Uwagi na temat klas I — dekorowanie metod klas”). Warto zauważyc, że można skorzystać z przypisywania atrybutów obiektów funkcji w celu śledzenia całkowitego czasu, ponieważ nie będziemy mieli zagnieżdżonej klasy dla celów zachowywania stanu i nie możemy uzyskać dostępu do zmiennych nielokalnych spoza kodu dekoratora.
2. Dekoratory klas `Public` i `Private`, napisane w niniejszym rozdziale, dodają pewne obciążenie związane z każdym pobraniem atrybutu w udekorowanej klasie. Choć moglibyśmy po prostu usunąć wiersz z dekoracją `@` w celu zyskania szybkości kodu, możemy również rozszerzyć sam dekorator w taki sposób, by sprawdzał przełącznik `__debug__` i nie wykonywał opakowywania wtedy, gdy opcja `-O` Pythona została przekazana w wierszu poleceń (tak samo, jak robiliśmy to w przypadku dekoratorów sprawdzających przedziały argumentów). W ten sposób możemy zwiększyć szybkość działania programu bez zmiany jego źródła za pomocą samych argumentów wiersza poleceń (`python -O main.py...`). Należy napisać kod takiego rozszerzenia oraz odpowiedniego testu.

Sprawdź swoją wiedzę — odpowiedzi

1. Poniżej znajduje się jeden ze sposobów zapisania w kodzie rozwiązania, a także jego wynik (choć z wykorzystaniem metod klas, które wykonywane są zbyt szybko, by można było zmierzyć czas ich działania). Sztuczka polega na zastąpieniu zagnieżdżonych klas *funkcjami zagnieżdżonymi*, tak by argument `self` nie był instancją dekoratora, a także przypisaniu całkowitego czasu do samej funkcji dekoratora, tak by można go było pobrać później za pomocą oryginalnej, dowiązanej ponownie nazwy (więcej szczegółów na ten temat można znaleźć w podrozdziale „Możliwości w zakresie zachowania informacji o stanie” — funkcje obsługują dołączanie dowolnych atrybutów, a nazwa funkcji jest w tym kontekście referencją do zakresu zawierającego).

```

import time

def timer(label='', trace=True):
    def onDecorator(func):
        def onCall(*args, **kargs):
            start = time.clock()
            result = func(*args, **kargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
                values = (label, func.__name__, elapsed, onCall.alltime)
                print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator

# Przetestowanie na funkcjach

@timer(trace=True, label='[CCC]==>')
def listcomp(N):
    return [x * 2 for x in range(N)]                                     # Dla argumentów dekoratora: zachowanie argumentów
                                                                     # Dla dekoracji @: zachowanie udekorowanej funkcji
                                                                     # W momencie wywołania: wywołanie oryginału
                                                                     # Stan to zakresy – atrybuty funkcji

@timer(trace=True, label='[MMM]==>')
def mapcall(N):
    return list(map(lambda x: x * 2, range(N))) # list() dla widoków z Pythona 3.0

for func in (listcomp, mapcall):
    result = func(5)                                              # Czas dla tego wywołania, wszystkich wywołań, zwieracana wartość
    func(5000000)
    print(result)
    print('allTime = %s\n' % func.alltime)      # Całkowity czas wszystkich wywołań

# Przetestowanie na metodach

class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @timer()
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)                                # giveRaise = timer()(giveRaise)
                                                                # tracer pamięta giveRaise

    @timer(label='**')
    def lastName(self):
        return self.name.split()[-1]                             # lastName = timer(...)(lastName)
                                                                # Całkowity czas per klasa, nie instancja

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
bob.giveRaise(.10)
anna.giveRaise(.20)
print(bob.pay, anna.pay)                                         # Wykonuje onCall(anna, .10)
print(bob.lastName(), anna.lastName())                         # Wykonuje onCall(bob), pamięta lastName
print('%.5f %.5f' % (Person.giveRaise.alltime, Person.lastName.alltime))

# Oczekiwane wyniki

[CCC]==>listcomp: 0.00002, 0.00002
[CCC]==>listcomp: 1.19636, 1.19638
[0, 2, 4, 6, 8]
allTime = 1.19637775192

```

```
[MMM]==>mapcall: 0.00002, 0.00002
[MMM]==>mapcall: 2.29260, 2.29262
[0, 2, 4, 6, 8]
allTime = 2.2926232943

giveRaise: 0.00001, 0.00001
giveRaise: 0.00001, 0.00002
55000.0 120000.0
**lastName: 0.00001, 0.00001
**lastName: 0.00001, 0.00002
Zielony Czerwona
0.00002 0.00002
```

1. Poniższe rozwiązywanie odpowiada na drugie pytanie. Kod został rozszerzony, tak by zwracać oryginalną klasę w trybie zoptymalizowanym (-O), aby dostęp do atrybutów nie wpływał negatywnie na szybkość działania programu. Tak naprawdę jedyną czynnością, którą wykonałem, było dodanie instrukcji sprawdzających tryb debugowania, a także dalsza indentacja klasy w prawą stronę. Jeśli chcemy obsługiwać delegację metod przeciążania operatorów do klasy podmiotowej w Pythonie 3.0, należy dodać ich ponowne definicje w klasie opakowującej (wersja 2.6 przekierowuje te metody za pośrednictwem `__getattr__`, natomiast wersja 3.0 i klasy w nowym stylu z 2.6 tego nie robią).

```
traceMe = False
def trace(*args):
    if traceMe: print('[' + ' '.join(map(str, args)) + ']')

def accessControl(failIf):
    def onDecorator(aClass):
        if not __debug__:
            return aClass
        else:
            class onInstance:
                def __init__(self, *args, **kargs):
                    self.__wrapped = aClass(*args, **kargs)
                def __getattribute__(self, attr):
                    trace('pobranie:', attr)
                    if failIf(attr):
                        raise TypeError('pobranie atrybutu prywatnego: ' + attr)
                    else:
                        return getattr(self.__wrapped, attr)
                def __setattr__(self, attr, value):
                    trace('ustawienie:', attr, value)
                    if attr == '__onInstance_wrapped__':
                        self.__dict__[attr] = value
                    elif failIf(attr):
                        raise TypeError('modyfikacja atrybutu prywatnego: ' + attr)
                    else:
                        setattr(self.__wrapped, attr, value)
            return onInstance
    return onDecorator

def Private(*attributes):
    return accessControl(failIf=(lambda attr: attr in attributes))

def Public(*attributes):
    return accessControl(failIf=(lambda attr: attr not in attributes))

# Kod testu: by ponownie wykorzystać dekorator, należy wydzielić test do osobnego pliku

@Private('age')                                     # Person = Private('age')(Person)
class Person:                                         # Person = onInstance ze stanem
    def __init__(self, name, age):
```

```

        self.name = name
        self.age = age           # Dostęp z wewnętrz działa normalnie

X = Person('Robert', 40)
print(X.name)                 # Dostęp z zewnątrz jest sprawdzany
X.name = 'Anna'
print(X.name)
#print(X.age)                # NIE POWIEDZIE SIE, o ile nie mamy "python -O"
#X.age = 999                  # Tak samo
#print(X.age)                # Tak samo

@Public('name')
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

X = Person('bob', 40)
print(X.name)                 # X to onInstance
X.name = 'Anna'               # onInstance osadza Person
print(X.name)
#print(X.age)                # NIE POWIEDZIE SIE, o ile nie mamy "python -O"
#X.age = 999                  # Tak samo
#print(X.age)                # Tak samo

```


Metaklasy

W poprzednim rozdziale omówiliśmy dekoratory i zapoznaliśmy się z różnymi przykładami ich użycia. W ostatnim rozdziale książki nadal pozostaniemy w kręgu tworzenia narzędzi i przedstawimy kolejne zaawansowane zagadnienie — *metaklasy* (ang. *metaclasses*).

W pewnym sensie metaklasy po prostu rozszerzają model wstawiania kodu dekoratorów. Jak wiemy z poprzedniego rozdziału, dekoratory funkcji i klas pozwalają nam przechwytywać i rozszerzać wywołania funkcji oraz wywołania tworzące instancje klasy. W podobny sposób metaklasy pozwalają nam przechwytywać i rozszerzać *tworzenie klas* — udostępniają API służące do wstawiania dodatkowej logiki, która ma być wykonana na zakończenie instrukcji `class`. Metaklasy robią to jednak w sposób odmienny od generatorów. Tym samym udostępniają ogólny protokół zarządzania obiektami klas w programie.

Tak jak wszystkie kwestie omawiane w tej części książki, jest to *zagadnienie zaawansowane*, z którym można się zapoznać w miarę potrzeby. W praktyce metaklasy pozwalają nam uzyskać wyższy poziom kontroli nad działaniem zbioru klas. Mają one spore możliwości i nie są przeznaczone dla większości programistów aplikacji (ani też, mówiąc szczerze, dla osób o słabym sercu!).

Z drugiej strony metaklasy otwierają drzwi do różnych wzorców programowania, które w inny sposób mogą być trudne lub niemożliwe do osiągnięcia, i są szczególnie interesujące dla programistów chcących pisać elastyczne API czy narzędzia programistyczne przeznaczone dla innych osób. Nawet jeśli ktoś nie mieści się w tej kategorii osób, metaklasy mogą go sporo nauczyć o ogólnym modelu klas Pythona.

Tak jak w poprzednim rozdziale, również tutaj naszym celem jest zaprezentowanie nieco bardziej realistycznych przykładów kodu, niż miało to miejsce we wcześniejszych częściach książki. Choć metaklasy są zagadnieniem z dziedziny samego jądra języka i nie przynależą do dziedziny aplikacji, celem części niniejszego rozdziału jest wzbudzenie chęci zapoznania się z bardziej rozbudowanymi przykładami programowania aplikacji po zakończeniu książki.

Tworzyć metaklasy czy tego nie robić?

Metaklasy są chyba najbardziej zaawansowanym zagadnieniem omawianym w niniejszej książce — jeśli nie w ogóle w Pythonie. Pozwole sobie zapozyczyć cytat z listy dyskusyjnej `comp.lang.python`, napisany przez tworzącego jądro Pythona programistę weterana Timu Petersa (który jest także autorem słynnego motta `z import this`):

[Metaklasy] to magia wyższego poziomu, którą 99% użytkowników Pythona nigdy nie będzie musiało zwracać sobie głowy. Jeśli zastanawiasz się, czy są ci one potrzebne, to nie są (osoby, które faktycznie będą ich potrzebowały, będą o tym wiedziały z całą pewnością i nie będą potrzebowały wyjaśnień, dlaczego tak jest).

Innymi słowy: metaklasy przeznaczone są przede wszystkim dla programistów tworzących API i narzędzi, z których będą korzystały inne osoby. W wielu przypadkach (jeśli nie ich większości) nie będą najlepszym wyborem w pracy nad aplikacjami. Jest tak szczególnie wtedy, gdy tworzy się kod, z którego w przyszłości będą korzystały inne osoby. Pisanie czegoś dlatego, że „wydaje się modne”, nie jest zazwyczaj rozsądny uzasadnieniem, o ile oczywiście akurat nie eksperymentujemy albo się nie uczymy.

Mimo to metaklasy mają szeroką gamę potencjalnych zastosowań i dobrze jest wiedzieć, kiedy mogą się przydać. Przykładowo można je wykorzystać do ulepszania klas za pomocą opcji takich jak śledzenie, trwałość obiektów czy logowanie wyjątków. Można ich także użyć między innymi do konstruowania części klasy w czasie wykonywania w oparciu o pliki konfiguracyjne, do uniwersalnego stosowania dekoratorów funkcji do każdej metody klasy oraz do sprawdzania zgodności z oczekiwany interfejsami.

W bardziej zaawansowanych wcieleniach metaklasy można nawet wykorzystać do implementowania alternatywnych wzorców kodu, na przykład takich jak programowanie zorientowane aspektowo czy odwzorowania obiektowo-relacyjne (ORM) dla baz danych. Choć często istnieją alternatywne sposoby uzyskiwania takich rezultatów (jak zobaczymy, role dekoratorów klasy oraz metaklas często się pokrywają), metaklasy udostępniają model formalny przygotowany z myślą o tych zadaniach. Nie mamy tutaj miejsca, by omówić w tym rozdziale tego typu zastosowania, jednak po zapoznaniu się z podstawami zachęcam każdego do poszukania w Internecie dodatkowych przypadków użycia.

Najbardziej istotnym dla niniejszej książki uzasadnieniem zapoznania się z metaklasami jest to, że zagadnienie to może wspomóc poznanie ogólnej mechaniki klas Pythona. Choć każdy może kiedyś tworzyć lub wykorzystywać utworzone przez kogoś metaklasy, ale nie musi tego robić, zrozumienie podstaw działania metaklas daje głębsze zrozumienie samego Pythona.

Zwiększające się poziomy magii

Większość niniejszej książki skupiała się na prostych technikach tworzenia kodu aplikacji, gdyż na ogół programiści większość czasu poświęcają na pisanie modułów, funkcji oraz klas służących do rzeczywistych celów. Programiści ci mogą wykorzystywać klasy i tworzyć instancje, a nawet w jakimś stopniu przeciągać operatory, jednak raczej nie będą się zbytnio zagłębiać w to, jak ich klasy naprawdę działają.

W niniejszej książce widzieliśmy także różne narzędzia pozwalające w uniwersalny sposób kontrolować działanie Pythona, które często mają więcej wspólnego z mechanizmami wewnętrznymi Pythona i tworzeniem narzędzi niż z dziedziną programowania aplikacji:

Atrybuty introspektywne

Atrybuty specjalne, takie jak `__class__` czy `__dict__`, pozwalają nam badać wewnętrzne aspekty implementacyjne obiektów Pythona, tak by można je było przetwarzać w sposób ogólny — wyświetlić wszystkie atrybuty obiektu czy nazwę klasy.

Metody przeciążania operatorów

Metody o specjalnych nazwach, takie jak `_str_` oraz `_add_`, zakodowane w klasach przechwytyują i udostępniają zastosowane do instancji klas wbudowane działania, takie jak wyświetlanie czy operatory wyrażeń. Są one wykonywane automatycznie w odpowiedzi na wbudowane operacje i pozwalają klasom na zachowanie zgodności z oczekiwanyymi interfejsami.

Metody przechwytywania atrybutów

Specjalna kategoria metod przeciążania operatorów udostępnia sposób przechwycenia dostępu do atrybutów w sposób ogólny — `_getattr_`, `_setattr_` oraz `_getattribute_` pozwalając klasom opakowującym na wstawianie automatycznie wykonywanego kodu, który może sprawdzać poprawność żądań atrybutów i delegować je do obiektów osadzonych. Pozwalają one na obliczenie przy dostępie dowolnej liczby atrybutów obiektu — albo wybranych atrybutów, albo wszystkich w ogóle.

Właściwości klas

Wbudowana funkcja `property` pozwala na wiązanie z określonym atrybutem klasy kodu, który jest wykonywany automatycznie przy pobieraniu, przypisywaniu lub usuwaniu atrybutu. Choć właściwości nie są tak uniwersalne jak narzędzia z poprzedniego akapitu, pozwalają one na automatyczne wywoływanie kodu przy dostępie do określonych atrybutów.

Deskryptory atrybutów klas

Tak naprawdę funkcja `property` jest zwięzłym sposobem definiowania deskryptora atrybutu, który automatycznie wykonuje funkcje w momencie dostępu. Deskryptory pozwalają definiować odrębne metody obsługi klas `_get_`, `_set_` oraz `_delete_`, wykonywane automatycznie w momencie dostępu do atrybutu przypisanego do instancji tej klasy. Udostępniają one ogólny sposób wstawiania automatycznie wykonywanego kodu w momencie dostępu do określonego atrybutu i są uruchamiane po normalnym wyszukiwaniu atrybutu.

Dekoratory funkcji i klas

Jak widzieliśmy w rozdziale 38., specjalna składnia dekoratorów z `@` pozwala nam dodać logikę, która zostanie wykonana automatycznie po wywołaniu funkcji lub utworzeniu instancji klasy. Ta logika opakowująca może śledzić lub ustawać w czasie wywołania, sprawdzać poprawność argumentów, zarządzać wszystkimi instancjami klasy, rozszerzać instancje o dodatkowe działania, takie jak sprawdzanie poprawności pobieranych atrybutów i wiele innych. Składnia dekoratorów wstawia logikę dowiązywania nazw, która ma być wykonywana na końcu instrukcji definicji funkcji oraz klas — nazwy udekorowanych funkcji i klas są ponownie dowiązywane do obiektów, które można wywoływać, przechwytyujących późniejsze wywołania.

Jak wspomniano we wprowadzeniu do niniejszego rozdziału, *metaklasy* są kontynuacją tej tematyki — pozwalają wstawić logikę, która będzie wykonywana automatycznie, kiedy tworzony jest obiekt klasy — na końcu instrukcji `class`. Logika ta nie dowiązuje ponownie nazwy klasy do obiektu wywoylanego dekoratora, a raczej przekierowuje samo tworzenie klasy do wyspecjalizowanej logiki.

Innymi słowy, metaklasy są właściwie innym sposobem definiowania *kodu wykonywanego automatycznie*. Za pomocą metaklas oraz pozostałych wymienionych narzędzi Python umożliwia nam wstawianie logiki w różnych kontekstach — w momencie analizy operatorów, dostępu do atrybutów, wywołań funkcji, tworzenia instancji klas, a teraz tworzenia obiektów klas.

W przeciwnieństwie do dekoratorów klas, które zazwyczaj dodają logikę wykonywaną w momencie tworzenia *instancji*, metaklasy wykonywane są w momencie tworzenia *klas*. Tym samym są punktami zaczepienia wykorzystywanymi przede wszystkim do zarządzania klasami (nie instancjami) lub rozszerzania ich.

Przykładowo metaklasy można wykorzystać między innymi do automatycznego dodania dekoracji do wszystkich metod klas, rejestrowania wszystkich klas do użycia w API, automatycznego dodawania logiki interfejsu użytkownika czy tworzenia lub rozszerzania klas z uproszczonych specyfikacji w plikach tekstowych. Ponieważ możemy kontrolować sposób tworzenia klas (a także za pomocą pośrednika zachowanie ich instancji), dziedzina zastosowania jest w tym przypadku bardzo szeroka.

Jak widzieliśmy, wiele z tych zaawansowanych narzędzi Pythona ma role, które się na siebie *nakładają*. Atrybutami można na przykład zarządzać za pomocą właściwości, deskryptorów lub metod przechwytywania atrybutów. Jak zobaczymy w niniejszym rozdziale, dekoratorów klas oraz metaklas można także używać zamiennie. Choć dekoratory klas są często wykorzystywane do zarządzania instancjami, można ich zamiast tego użyć do zarządzania klasami. I podobnie — choć metaklasy zaprojektowano w celu rozszerzania konstrukcji klas, mogą także wstawać kod, zarządzając tym samym instancjami. Ponieważ wybór, z której techniki korzystać, jest czasami bardzo subiektywny, znajomość alternatywnych rozwiązań może pomóc wybrać narzędzie odpowiednie dla zadania.

Wady funkcji pomocniczych

Tak jak dekoratory z poprzedniego rozdziału, metaklasy teoretycznie często są opcjonalne. Zazwyczaj ten sam efekt możemy uzyskać, przekazując obiekty klas za pomocą *funkcji zarządzających* (nazywanych czasami także funkcjami pomocniczymi), w podobny sposób, jak możemy osiągnąć cele uzyskiwane za pomocą dekoratorów, przekazując funkcje i instancje za pomocą kodu zarządzającego. Podobnie jak dekoratory, tak i metaklasy:

- Udostępniają bardziej formalną i jawną strukturę.
- Pomagają upewnić się, że programiści aplikacji nie zapomną rozszerzyć klas zgodnie z wymaganiami API.
- Pozwalają uniknąć powtarzalności kodu i związanych z nią kosztów utrzymywania, dokonując faktoryzacji logiki dostosowującej klasę do własnych potrzeb w jednym miejscu — w metaklasie.

By to zilustrować, założymy, że chcemy automatycznie wstawić metodę do zbioru klas. Oczywiście moglibyśmy to zrobić za pomocą prostego *dziedziczenia*, jeśli metoda jest znana w momencie tworzenia kodu klas. W takim przypadku kod metody wpisujemy do klasy nadzędnej, a wszystkie klasy docelowe będą ją stamtąd dziedziczyć:

```
class Extras:  
    def extra(self, args):  
        ...  
  
    # Normalne dziedziczenie: zbyt statyczne  
  
class Client1(Extras): ...  
class Client2(Extras): ...  
class Client3(Extras): ...  
  
    # Klasy Client dziedziczą dodatkowe metody  
  
X = Client1()  
X.extra()  
  
    # Utworzenie instancji  
    # Wykonanie dodatkowych metod
```

Czasami jednak nie da się przewidzieć takiego rozszerzenia na etapie pisania kodu klas. Rozważmy przypadek, w którym klasy rozszerzane są w odpowiedzi na wybór dokonany przez użytkownika w czasie wykonywania lub specyfikację wpisaną do pliku konfiguracyjnego. Choć moglibyśmy napisać kod każdej klasy w naszym wyimaginowanym zbiorze, tak by to *ręcznie* sprawdzał, jest to naprawdę spore wymaganie w stosunku do klientów (required jest tu abstrakcyjne — to coś, co ma być wypełnione):

```
def extra(self, arg): ...

class Client1: ...
if required():
    Client1.extra = extra

class Client2: ...
if required():
    Client2.extra = extra

class Client3: ...
if required():
    Client3.extra = extra

X = Client1()
X.extra()
```

Klasa Client rozszerza: zbyt rozdrobnione

Możemy w ten sposób dodawać metody do klasy już po instrukcji `class`, ponieważ metoda klasy jest po prostu funkcją powiązaną z klasą i ma pierwszy argument, w którym przekazujemy instancję `self`. Choć powyższe rozwiązanie działa, cały ciężar rozszerzania kładzie na klasach klienta (i zakłada, że to one mają pamiętać, by to wszystko zrobić!).

Z punktu widzenia utrzymywania kodu lepszym rozwiązaniem byłoby wyizolowanie logiki wyboru w jednym miejscu. Możemy hermetyzować część tych dodatkowych działań, przekierowując klasy za pomocą *funkcji zarządzającej*. Taka funkcja zarządzająca rozszerzyłaby klasę zgodnie z wymaganiami i poradziła sobie z wszystkimi kwestiami wynikającymi z testowania w czasie wykonania oraz konfiguracją:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra

class Client1: ...
extras(Client1)

class Client2: ...
extras(Client2)

class Client3: ...
extras(Client3)

X = Client1()
X.extra()
```

Funkcja zarządzająca: zbyt ręczna

Powyższy kod powoduje wykonanie klasy za pośrednictwem funkcji zarządzającej bezpośrednio po utworzeniu. Choć funkcje zarządzające tego typu mogą tutaj spełnić nasze cele, nadal zbyt duży ciężar spoczywa na twórcach klas, którzy muszą rozumieć wymagania i stosować się do nich w kodzie. Byłoby lepiej, gdyby istniał prosty sposób wymuszania rozszerzania

w klasach docelowych, tak by musiały się one zajmować rozszerzaniem ani o nim nie zapominały. Innymi słowy, chcielibyśmy móc wstawić na końcu instrukcji `class` jakiś kod, który będzie wykonywany *automatycznie* i spowoduje rozszerzenie klasy.

Właśnie do tego służą *metaklasy*. Deklarując metaklasę, mówimy Pythonowi, by przekierował tworzenie obiektu klasy do innej przekazanej mu klasy:

```
def extra(self, arg): ...

class Extras(type):
    def __init__(Class, classname, superclasses, attributedict):
        if required():
            Class.extra = extra

class Client1(metaclass=Extras): ...
class Client2(metaclass=Extras): ...          # Jedynie deklaracja metaklasy
class Client3(metaclass=Extras): ...          # Klasa Client jest instancją metaklasy

X = Client1()                                # X jest instancją klasy Client1
X.extra()
```

Ponieważ Python wywołuje metaklasę automatycznie na końcu instrukcji `class`, kiedy tworzona jest nowa klasa, może rozszerzać klasę, rejestrować ją lub w inny sposób nią zarządzać. Co więcej, jedynym wymaganiem ze strony klas klienta jest to, że muszą one deklarować metaklasę. Każda klasa, która to robi, automatycznie pobiera rozszerzenie udostępniane przez metaklasę — zarówno teraz, jak i w przyszłości, jeśli metaklasa się zmieni. Choć może być trudno zobaczyć to w powyższym krótkim przykładzie, metaklasy zazwyczaj radzą sobie z takimi zadaniami lepiej od pozostałych rozwiązań.

Metaklasy a dekoratory klas — runda 1.

Warto również zauważyć, że opisane w poprzednim rozdziale *dekoratory klas* czasami nakładają się z metaklasami w zakresie funkcjonalności. Choć zazwyczaj wykorzystywane są do zarządzania instancjami lub rozszerzania ich, dekoratory klas mogą także rozszerzać klasy niezależnie od wszelkich utworzonych instancji.

Przypuśćmy na przykład, że nasza funkcja zarządzająca ma zwracać rozszerzoną klasę, zamiast po prostu modyfikować ją w miejscu. Pozwoli to na większą elastyczność, ponieważ funkcja zarządzająca może zwracać dowolny typ obiektu implementujący oczekiwany interfejs klasy:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

class Client1: ...
Client1 = extras(Client1)

class Client2: ...
Client2 = extras(Client2)

class Client3: ...
Client3 = extras(Client3)

X = Client1()
X.extra()
```

Osoby, którym się wydaje, że kod ten zaczyna przypominać dekoratory klas, mają rację. W poprzednim rozdziale zaprezentowaliśmy dekoratory klas jako narzędzie służące do rozszerzania wywołań tworzących *instancje*. Ponieważ jednak działają one automatycznie, dowiązując nazwę klasy do wyniku funkcji, nie ma żadnego powodu, byśmy nie mogli użyć ich do rozszerzenia klasy przed utworzeniem jakichkolwiek instancji. Oznacza to, że dekoratory klas mogą zastosować dodatkową logikę do *klas*, a nie tylko *instancji*, w momencie tworzenia:

```
def extra(self, arg): ...

def extras(Class):
    if required():
        Class.extra = extra
    return Class

@extras
class Client1: ...                                # Client1 = extras(Client1)

@extras
class Client2: ...                                # Ponownie dowiązuje klasę niezależnie od instancji

@extras
class Client3: ...

X = Client1()                                      # Tworzy instancję rozszerzonej klasy
X.extra()                                           # X jest instancją oryginalnej klasy Client1
```

Dekoratory zasadniczo automatyzują tutaj ręczne dowiązywanie nazwy z poprzedniego przykładu. Tak jak w przypadku metaklas, ponieważ dekorator zwraca oryginalną klasę, instancje są tworzone z niego, a nie z obiektu opakowującego. Tak naprawdę tworzenie instancji w ogóle nie jest przechwytywane.

W tym akurat przypadku — dodawania metod do klasy, kiedy jest ona tworzona — wybór pomiędzy metaklasami a dekoratorami jest dość dowolny. Dekoratory można wykorzystać do zarządzania zarówno instancjami, jak i klasami, a w tej drugiej roli pokrywają się one z metaklasami.

Tak naprawdę jednak odpowiada to tylko jednemu trybowi działania metaklas. Jak zobaczymy, dekoratory są w tej roli odpowiednikiem metod `__init__` metaklasy, jednak to metaklasy mają dodatkowe punkty zaczepienia pozwalające na dostosowanie do własnych potrzeb. Jak się również przekonamy, poza inicjalizacją klas, metaklasy mogą wykonywać dowolne zadania konstrukcyjne, które mogą być trudniejsze do wykonania za pomocą dekoratorów.

Co więcej, choć dekoratory potrafią zarówno instancjami, jak i klasami, odwrotna zależność nie występuje — metaklasy zaprojektowano do zarządzania klasami i zastosowanie ich dla celów zarządzania *instancjami* jest o wiele mniej proste. Omówimy tę różnicę na przykładzie kodu nieco później w niniejszym rozdziale.

Większość kodu z tego podrozdziału była abstrakcyjna, jednak w dalszej części rozdziału przejdziemy także do prawdziwego i działającego przykładu. By jednak w pełni zrozumieć, jak działają metaklasy, musimy najpierw uzyskać jaśniejszy obraz modelu leżącego u ich podstaw.

Model metaklasy

By naprawdę zrozumieć, jak działają metaklasy, musimy dowiedzieć się nieco więcej o modelu typów Pythona, a także tym, co dzieje się na końcu instrukcji `class`.

Klasy są instancjami obiektu `type`

Dotychczas w książce wykonywaliśmy większość pracy, tworząc instancje typów wbudowanych, takich jak listy i łańcuchy znaków, a także instancje klas tworzonych przez nas samych. Jak widzieliśmy, instancje klas mają pewne własne atrybuty z informacjami o stanie, jednak dziedziczą także atrybuty związane z dziedziczeniem po klasach, z których zostały utworzone. Tak samo jest w przypadku typów wbudowanych — instancje list mają na przykład własne wartości, jednak dziedziczą metody po typie listy.

Choć z takimi obiekty instancji możemy sporo zrobić, model typów Pythona okazuje się nieco bogatszy, niż to przedstawiłem. Tak naprawdę w modelu opisywanym dotychczas znajduje się pewna luka — skoro instancje tworzone są z klas, co tworzy same klasy? Okazuje się, że także klasy są instancją czegoś:

- W Pythonie 3.0 obiekty klas zdefiniowanych przez użytkownika są instancjami obiektu o nazwie `type`, który sam jest klasą.
- W Pythonie 2.6 klasy w nowym stylu dziedziczą po `object` będącym klasą podzieloną `type`. Klasy klasyczne są instancjami `type` i nie są tworzone z klasami.

Omawialiśmy pojęcie typów w rozdziale 9., a związek klas z typami w rozdziale 31., jednak wróćmy tutaj do podstaw, by pokazać, jakie zastosowanie ma to do metaklas.

Przypomnijmy, że wbudowana funkcja `type` zwraca typ dowolnego obiektu (który sam jest obiektem). W przypadku typów wbudowanych, takich jak listy, typem instancji jest wbudowany typ listy, jednak typ typu listy sam jest typem `type`. Obiekt `type` znajdujący się na górze hierarchii tworzy określone typy, natomiast konkretne typy tworzą instancje. Można to zobaczyć samemu w sesji interaktywnej. Na przykład w Pythonie 3.0:

```
C:\misc> c:\python30\python
>>> type([])
<class 'list'>
# W 3.0 lista jest instancją typu list
>>> type(type([]))
<class 'type'>
# Dla list typem jest klasa type
>>> type(list)
<class 'type'>
# To samo, ale z nazwami typów
>>> type(type)
<class 'type'>
# Typem type jest type: szczyt hierarchii
```

Zgodnie z tym, czego dowiedzieliśmy się o zmianach dotyczących klas w nowym stylu w rozdziale 31., tak samo jest w Pythonie 2.6 (i starszych wersjach), jednak typy nie są do końca tym, co klasy — `type` jest unikalnym rodzajem wbudowanego obiektu stojącego na szczytach hierarchii typów i wykorzystywany do konstruowania typów:

```
C:\misc> c:\python26\python
>>> type([])
<type 'list'>
# W 2.6 type jest czymś nieco innym
>>> type(type([]))
<type 'type'>
```

```
>>> type(list)
<type 'type'>
>>> type(type)
<type 'type'>
```

Okazuje się, że nasz relacja między typami a instancjami jest prawdziwa również dla klas — instancje tworzone są z klas, a klasy tworzone są z `type`. W Pythonie 3.0 pojęcie typu zostało jednak połączone z pojęciem klasy. Tak naprawdę te dwa pojęcia są właściwie synonimami — *klasy są typami, a typy są klasami*. A zatem:

- Typy definiowane są przez klasy pochodzące od `type`.
- Klasy zdefiniowane przez użytkownika są instancjami klas typów.
- Klasy zdefiniowane przez użytkownika są typami generującymi własne instancje.

Jak widzieliśmy wcześniej, ta równoważność wpływa na kod sprawdzający typ instancji — typem instancji jest klasa, z której została ona utworzona. Ma to także wpływ na sposób tworzenia klas, który okazuje się kwestią kluczową dla tematu niniejszego rozdziału. Ponieważ klasy normalnie domyślnie tworzone są z głównej klasy typu, większość programistów nie musi się zastanawiać nad równoważnością typów i klas. Otwiera to jednak przed nami nowe możliwości dostosowywania klas i ich instancji do własnych potrzeb.

Przykładowo klasy w wersji 3.0 (a także klasy w nowym stylu z Pythona 2.6) są instancjami klasy `type`, natomiast obiekty instancji są instancjami ich klas. Klasa mają teraz atrybut specjalny `__class__`, który wskazuje `type`, tak samo jak instancje mają atrybut `__class__` wskazujący klasę, z której powstała instancja:

```
C:\misc> c:\python30\python
>>> class C: pass
# Obiekt klasy z 3.0 (nowy styl)
...
>>> X = C()
# Obiekt instancji klasy
# Instancja jest instancją klasy
# Klasa instancji
>>> type(X)
<class '__main__.C'>
>>> X.__class__
<class '__main__.C'>

>>> type(C)
<class 'type'>
>>> C.__class__
<class 'type'>
# Klasa jest instancją type
# Klasa klasy to type
```

Warto zwrócić szczególną uwagę na dwa ostatnie wiersze — klasy są instancjami klasy `type` w taki sam sposób, jak normalne instancje są instancjami klasy. Działa to tak samo zarówno dla klas wbudowanych, jak i typów klas zdefiniowanych przez użytkownika w wersji 3.0. Klasa nie są wcale tak naprawdę odrębną koncepcją — to po prostu typy zdefiniowane przez użytkownika, a sam `type` definiowany jest za pomocą klasy.

W Pythonie 2.6 działa to podobnie w przypadku klas w nowym stylu, pochodzących od `object`, ponieważ uruchamia to działanie z wersji 3.0:

```
C:\misc> c:\python26\python
>>> class C(object): pass
# W klasach w nowym stylu z 2.6
# także klasy mają klasę
...
>>> X = C()
# W klasach w nowym stylu z 2.6
# także klasy mają klasę

>>> type(X)
<class '__main__.C'>
>>> type(C)
<class 'type'>
```

```
<type 'type'>  
>>> X.__class__  
<class '__main__.C'>  
>>> C.__class__  
<type 'type'>
```

Tradycyjne klasy z wersji 2.6 są jednak nieco inne — ponieważ odzwierciedlają model klas ze starszych wydań Pythona, nie mają odnośnika `__class__` i tak jak typy wbudowane z wersji 2.6 są one instancjami `type`, a nie klasy typu:

```
C:\misc> c:\python26\python  
>>> class C: pass  
...  
>>> X = C()  
  
>>> type(X)  
<type 'instance'>  
>>> type(C)  
<type 'classobj'>  
  
>>> X.__class__  
<class '__main__.C' at 0x005F85A0>  
>>> C.__class__  
AttributeError: class C has no attribute '__class__'
```

Metaklasy są klasami podrzędnymi klasy `type`

Dlaczego zatem ma dla nas znaczenie, że w wersji 3.0 klasy są instancjami klasy `type`? Okaże się, że to właśnie jest punkt zaczepienia, który pozwala nam pisać metaklasy. Ponieważ pojęcie *typu* jest obecnie równoznaczne z *klasą*, możemy tworzyć klasy podrzędne dla `type` za pomocą normalnych technik zorientowanych obiektywem, a także składni klas umożliwiającej dostosowanie ich do własnych potrzeb. A ponieważ klasy są tak naprawdę instancjami klasy `type`, tworzenie ich z dostosowanych do naszych potrzeb klas podrzędnych `type` pozwala na implementację własnych rodzajów klas. Uwzględniając wszystkie szczegóły, całość działa dosłownie — w wersji 3.0, a także klasach w nowym stylu z wersji 2.6:

- `type` jest klasą generującą klasy zdefiniowane przez użytkownika.
- Metaklasy są klasami podrzędnymi klasy `type`.
- Obiekty klas są instancjami klasy `type` lub inaczej jej klasami podrzędnymi.
- Obiekty instancji generowane są z klasy.

Innymi słowy, by kontrolować sposób tworzenia klas i rozszerzania ich działania, wystarczy podać, że klasa zdefiniowana przez użytkownika ma być tworzona z metaklasy zdefiniowanej przez użytkownika, a nie z normalnej klasy `type`.

Warto zauważyc, że ten związek *instancji typów* nie jest tym samym, co *dziedziczenie*. Klasy zdefiniowane przez użytkownika także mogą mieć klasy nadzędne, po których one same oraz ich instancje dziedziczą atrybuty (klasy nadzędne dziedziczenia wymienione są w nawiasach w instrukcji `class` i pokazują się w krotce `__bases__` klasy). Typ, na bazie którego tworzona jest klasa, i którego jest ona instancją, to inny rodzaj związku. W kolejnym podrozdziale opisano procedurę stosowaną przez Pythona do implementacji związku „bycia instancją typu”.

Protokół instrukcji class

Utworzenie klasy podrzędnej dla type w celu dostosowania jej do własnych potrzeb to tak naprawdę połowa magii stojącej za metaklasami. Nadal musimy w jakiś sposób skierować tworzenie klasy do metaklasy zamiast do domyślnej type. By w pełni zrozumieć sposób wykonywania tego, musimy także wiedzieć, w jaki sposób działają instrukcje class.

Wiemy już, że kiedy Python znajduje instrukcję class, wykonuje zagnieźdzony w niej blok kodu w celu utworzenia atrybutów klasy — wszystkie nazwy przypisane na najwyższym poziomie zagnieżdzonego bloku kodu generują atrybuty w wynikowym obiekcie klasy. Te nazwy są zazwyczaj funkcjami metod utworzonymi za pomocą zagnieżdzonych instrukcji def, jednak mogą to być także dowolne inne atrybuty przypisane w celu utworzenia danych klasy wspólnie dzielonych przez wszystkie jej instancje.

Z technicznego punktu widzenia Python postępuje w tym celu zgodnie ze standardowym protokołem. Na końcu instrukcji class i po wykonaniu całego zagnieżdzonego w niej kodu w słowniku przestrzeni nazw wywołuje obiekt type w celu utworzenia obiektu klasy:

```
class = type(nazwa_klasy, klasy_nadzędne, słownik_atrybutów)
```

Obiekt type definiuje z kolei metodę przeciążania operatorów __call__, która wykonuje dwie pozostałe metody, gdy wywoływany jest obiekt type:

```
type.__new__(klaśa_tupu, nazwa_klasy, klasy_nadzędne, słownik_atrybutów)  
type.__init__(klaśa, nazwa_klasy, klasy_nadzędne, słownik_atrybutów)
```

Metoda __new__ tworzy i zwraca nowy obiekt class, a następnie metoda __init__ inicjalizuje nowo utworzony obiekt. Jak zobaczymy za moment, są to właśnie punkty zaczepienia wykorzystywane przez metaklasy (będące klasami podrzędnymi type) do dostosowywania klas do własnych potrzeb.

Mając na przykład poniższą definicję klasy:

```
class Spam(Eggs):  
    data = 1  
    def meth(self, arg):  
        pass  
# Dziedziczy po Eggs  
# Atrybut danych klasy  
# Atrybut metody klasy
```

wewnętrznie Python wykona zagnieżdzony blok kodu w celu utworzenia dwóch atrybutów klasy (data oraz meth), a następnie wywoła obiekt type, by wygenerować obiekt klasy na końcu instrukcji class:

```
Spam = type('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

Ponieważ wywołanie to odbywa się na końcu instrukcji class, jest to idealny punkt zaczepienia dla rozszerzania klasy lub przetwarzania jej w inny sposób. Sztuczka polega na zastąpieniu type za pomocą własnej klasy podrzędnej, która przechwyci to wywołanie. W kolejnym podrozdziale pokażemy, jak to zrobić.

Deklarowanie metaklas

Jak widzieliśmy przed chwilą, klasy domyślnie tworzone są za pomocą klasy type. Żeby przekazać Pythonowi, by tworzył klasę za pomocą własnej metaklasy, musimy po prostu zadeklarować metaklasę w taki sposób, by przechwytywała ona normalne wywołania tworzące

klasy. Sposób wykonania tego uzależniony jest od wykorzystywanej wersji Pythona. W Pythonie 3.0 należy wymienić pożądaną metaklasę jako argument ze słowem kluczowym w nagłówku instrukcji `class`:

```
class Spam(metaclass=Meta): # Wersja 3.0 i nowsze
```

Klasy nadrzędne, po których się dziedziczy, także można wymienić w nagłówku, przed metaklasą. W poniższym przykładzie nowa klasa `Spam` dziedziczy po `Eggs`, ale jest równocześnie instancją klasy `Meta`, przez którą jest tworzona:

```
class Spam(Eggs, metaclass=Meta): # Dopuszczalne inne klasy nadrzędne
```

Ten sam efekt uzyskamy w Pythonie 2.6, jednak metaklasę musimy podać w inny sposób — za pomocą atrybutu klasy, a nie argumentu ze słowem kluczowym. Pochodzenie od obiektu jest wymagane do utworzenia klasy w nowym stylu, jednak poniższy zapis nie działa już w wersji 3.0, ponieważ atrybut zostanie po prostu zignorowany:

```
class spam(object): # Tylko wersja 2.6  
    __metaclass__ = Meta
```

W Pythonie 2.6 zmienna globalna modułu `__metaclass__` jest dostępna i łączy wszystkie klasy w module z metaklasą. W wersji 3.0 nie jest to już obsługiwane, gdyż zamierzone było jako tymczasowy sposób ułatwiający domyślne tworzenie klas w nowym stylu bez wyprodawdzania każdej klasy od `object`.

Po zadeklarowaniu w powyższy sposób wywołanie tworzące obiekt `class` wykonywane na końcu instrukcji `class` jest modyfikowane, tak by wywołać metaklasę zamiast domyślnej klasy `type`:

```
class = Meta(nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)
```

A ponieważ metaklasa jest klasą podzieloną `type`, metoda `__call__` klasy `type` deleguje wywołania tworzące i inicjalizujące nowe obiekty klas do metaklasy, jeśli definiuje ona własne wersje poniższych metod:

```
Meta.__new__(Meta, nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)  
Meta.__init__(klaś, nazwa_klasy, klasy_nadrzędne, słownik_atrybutów)
```

By to zademonstrować, poniżej zaprezentowano ponownie przykład z poprzedniego podrozdziału, rozszerzony o specyfikację metaklasy z wersji 3.0:

```
class Spam(Eggs, metaclass=Meta): # Dziedziczy po Eggs, instancji Meta  
    data = 1 # Atrybut danych klas  
    def meth(self, arg): # Atrybut metody klas  
        pass
```

Na końcu powyższej instrukcji `class` Python wewnętrznie wykonuje poniższy kod w celu utworzenia obiektu `class`:

```
Spam = Meta('Spam', (Eggs,), {'data': 1, 'meth': meth, '__module__': '__main__'})
```

Jeśli metaklasa definiuje własne wersje metod `__new__` lub `__init__`, zostaną one wywołane w trakcie tego wywołania przez odziedziczoną po `type` metodę `__call__` w celu utworzenia i zainicjalizowania nowej klasy. W kolejnym podrozdziale pokazane zostanie, jak mógłby wyglądać kod ostatniej części układanki z metaklasami.

Tworzenie metaklas

Dotychczas widzieliśmy, w jaki sposób Python skierowuje tworzenie klasy do metaklasy, o ile taka zostanie podana. W jaki sposób jednak tworzymy metaklasę dostosowującą type do naszych wymagań?

Okazuje się, że większość zadania jest nam już znana — metaklasy tworzy się za pomocą normalnych instrukcji `class` i semantyki Pythona. Jedyna znacząca różnica polega na tym, że Python wywołuje je automatycznie na końcu instrukcji `class` i że muszą się one stosować do interfejsów oczekiwanych przez klasę nadziedziedzianą `type`.

Prosta metaklasa

Chyba najprostsza metaklasa, jaką można utworzyć, to po prostu klasa podzielona `type` z metodą `__new__` tworzącą obiekt klasy za pomocą wykonania wersji domyślnej w `type`. Taka metoda `__new__` metaklasy wykonywana jest przez metodę `__call__` dziedziczoną po `type`. Zazwyczaj wykonuje pożądane dostosowanie do naszych potrzeb i wywołuje metodę `__new__` klasy nadziedziedzianej `type` w celu utworzenia i zwrócenia nowego obiektu klasy:

```
class Meta(type):
    def __new__(metaklasa, nazwa_klasy, klasy_nadziedzne, słownik_atrybutów):
        # Wykonywane przez dziedziczone type.__call__
        return type.__new__(metaklasa, nazwa_klasy, klasy_nadziedzne, słownik_atrybutów)
```

Ta metaklasa tak naprawdę nic nie robi (możemy również dobrze pozwolić domyślnej klasie `type` na utworzenie klasy), jednak demonstruje sposób wejścia metaklasy w miejsce punktu zaczepienia w celu dostosowania klasy. Ponieważ metaklasa wywoływana jest na końcu instrukcji `class`, a metoda `__call__` obiektu `type` odsyła do metod `__new__` oraz `__init__`, kod udostępniony w tych metodach jest w stanie zarządzać wszystkimi klasami utworzonymi z metaklasą.

Oto ponownie nasz przykład w akcji, z dodanymi do metaklasy wywołaniami `print` i plikiem do śledzenia:

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('W MetaOne.new:', classname, supers, classdict, sep='\n... ')
        return type.__new__(meta, classname, supers, classdict)

class Eggs:
    pass

print('tworzenie klasy')
class Spam(Eggs, metaclass=MetaOne):
    data = 1
    def meth(self, arg):
        pass

    # Dziedziczy po Eggs, instancji Meta
    # Atrybut danych klasy
    # Atrybut metody klasy

print('tworzenie instancji')
X = Spam()
print('dane:', X.data)
```

W powyższym kodzie `Spam` dziedziczy po `Eggs` i jest instancją `MetaOne`, natomiast `X` jest instancją klasy `Spam`, po której dziedziczy. Po wykonaniu tego kodu w Pythonie 3.0 można zauważyć, że metaklasa wywoływana jest na końcu instrukcji `class`, zanim jeszcze utworzymy instancję — metaklasy służą do przetwarzania klas, natomiast klasy służą do przetwarzania instancji:

```
tworzenie klasy
W MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AEBA08>}
tworzenie instancji
dane: 1
```

Dostosowywanie tworzenia do własnych potrzeb oraz inicjalizacja

Metaklasy mogą także wejść w miejsce protokołu `__init__` wywołanego przez metodę `__call__` obiektu `type`. Metoda `__new__` tworzy i zwraca obiekt klasy, natomiast `__init__` inicjalizuje utworzoną już klasę. Metaklasy mogą wykorzystać oba punkty zaczepienia do zarządzania klasami w momencie tworzenia:

```
class MetaOne(type):
    def __new__(meta, classname, supers, classdict):
        print('W MetaOne.new:', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(Class, classname, supers, classdict):
        print('W MetaOne init:', classname, supers, classdict, sep='\n...')
        print('...obiekt zainicjalizowanej klasy:', list(Class.__dict__.keys()))

class Eggs:
    pass

print('tworzenie klasy')
class Spam(Eggs, metaclass=MetaOne):
    data = 1
    def meth(self, arg):
        pass

    # Dziedziczy po Eggs, instancji Meta
    # Atrybut danych klasy
    # Atrybut metody klasy

print('tworzenie instancji')
X = Spam()
print('dane:', X.data)
```

W tym przypadku metoda inicjalizacji klasy wykonana zostaje po metodzie tworzenia klasy, a obie wykonywane są na końcu instrukcji `class` przed utworzeniem jakichkolwiek instancji:

```
tworzenie klasy
W MetaOne.new:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AAB810>}
W MetaOne init:
...Spam
...(<class '__main__.Eggs'>,)
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02AAB810>}
...obiekt zainicjalizowanej klasy: ['__module__', 'data', 'meth', '__doc__']
tworzenie instancji
dane: 1
```

Pozostałe sposoby tworzenia metaklas

Choć redefiniowanie metod `__new__` oraz `__init__` klasy nadrzędnej `type` jest najczęstszym sposobem wstawiania logiki do procesu tworzenia obiektu klasy za pomocą metaklasy, możliwe są także inne rozwiązania.

Użycie prostych funkcji fabrycznych

Metaklasy tak naprawdę nie muszą wcale być klasami. Jak już wiemy, instrukcja `class` wykonuje proste wywołanie tworzące klasę na końcu jej przetwarzania. Z tego powodu każdy *obiekt, który można wywoływać*, może być wykorzystany jako metaklasa, pod warunkiem że przyjmuje przekazane argumenty i zwraca obiekt zgodny z zamierzoną klasą. Tak naprawdę prosta funkcja fabryczna obiektu zadziała również dobrze jak klasa:

```
# Prosta funkcja może także działać jak metaklasa

def MetaFunc(classname, supers, classdict):
    print('W MetaFunc: ', classname, supers, classdict, sep='\n...')
    return type(classname, supers, classdict)

class Eggs:
    pass

print('tworzenie klasy')
class Spam(Eggs, metaclass=MetaFunc):
    data = 1
    def meth(self, args):
        pass

print('tworzenie instancji')
X = Spam()
print('dane:', X.data)
```

Wykonanie prostej funkcji na końcu
Funkcja zwraca klasę

Przy wykonywaniu funkcja wywoływana jest na końcu instrukcji deklarującej klasę i zwraca oczekiwany obiekt nowej klasy. Funkcja ta po prostu przechwytuje wywołanie, które normalnie domyślnie przechwytuje metoda `__call__` obiektu `type`:

```
tworzenie klasy
W MetaFunc:
...Spam
...(<class '__main__.Eggs'>,
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B8B6A8>}
tworzenie instancji
dane: 1
```

Przeciążenie wywołań tworzących klasę za pomocą metaklas

Ponieważ metaklasy biorą udział w normalnych mechanizmach programowania zorientowanego obiektowo, mogą także przechwytywać *wywołania tworzące klasy* bezpośrednio na końcu instrukcji `class`, redefiniując metodę `__call__` obiektu `type`. Wymagany do tego protokół jest jednak nieco skomplikowany:

```
# Metodę __call__ można redefineować, metaklasy mogą mieć metaklasy

class SuperMeta(type):
    def __call__(meta, classname, supers, classdict):
        print('W SuperMeta.call: ', classname, supers, classdict, sep='\n...')
        return type.__call__(meta, classname, supers, classdict)

class SubMeta(type, metaclass=SuperMeta):
    def __new__(meta, classname, supers, classdict):
        print('W SubMeta.new: ', classname, supers, classdict, sep='\n...')
        return type.__new__(meta, classname, supers, classdict)

    def __init__(self, classname, supers, classdict):
        print('W SubMeta init: ', classname, supers, classdict, sep='\n...')
        print('...obiekt zainicjalizowanej klasy:', list(self.__dict__.keys()))
```

```

class Eggs:
    pass

print('tworzenie klasy')
class Spam(Eggs, metaclass=SubMeta):
    data = 1
    def meth(self, arg):
        pass

print('tworzenie instancji')
X = Spam()
print('dane:', X.data)

```

Po wykonaniu powyższego kodu wszystkie trzy redefiniowane metody są wykonywane po kolej. To samo mniej więcej domyślnie robi obiekt type:

```

tworzenie klasy
W SuperMeta.call:
...Spam
...(<class '__main__.Eggs'>,
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
W SubMeta.new:
...Spam
...(<class '__main__.Eggs'>,
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
W SubMeta.init:
...Spam
...(<class '__main__.Eggs'>,
...{'__module__': '__main__', 'data': 1, 'meth': <function meth at 0x02B7BA98>}
...obiekt zainicjalizowanej klasy: ['__module__', 'data', 'meth', '__doc__']
tworzenie instancji
dane: 1

```

Przeciążenie wywołań tworzących klasę za pomocą normalnych klas

Poprzedni przykład komplikuje fakt, iż metaklasy wykorzystane zostały do tworzenia obiektów klas, jednak same nie generują instancji. Z tego powodu w przypadku metaklas reguły wyszukiwania nazw są nieco odmienne od tego, do czego jesteśmy przyzwyczajeni. Metoda `__call__` na przykład wyszukiwana jest w klasie obiektu. W przypadku metaklas oznacza to metaklasę metaklasy.

By skorzystać z normalnego wyszukiwania nazw opartego na dziedziczeniu, możemy wykorzystać ten sam efekt z normalnymi klasami iinstancjami. Rezultat poniższego kodu będzie taki sam jak wersji poprzedniej, jednak warto zauważyc, że metody `__new__` oraz `__init__` muszą tutaj mieć inne nazwy, gdyż inaczej zostaną one wykonane w momencie utworzenia instancji SubMeta, a nie kiedy SubMeta będzie później wywołana jako metaklasa:

```

class SuperMeta:
    def __call__(self, classname, supers, classdict):
        print('W SuperMeta.call: ', classname, supers, classdict, sep='\n...')
        Class = self.__New__(classname, supers, classdict)
        self.__Init__(Class, classname, supers, classdict)
        return Class

class SubMeta(SuperMeta):
    def __New__(self, classname, supers, classdict):
        print('W SubMeta.new: ', classname, supers, classdict, sep='\n...')
        return type(classname, supers, classdict)

    def __Init__(self, Class, classname, supers, classdict):
        print('W SubMeta.init: ', classname, supers, classdict, sep='\n...')
        print('...obiekt zainicjalizowanej klasy:', list(Class.__dict__.keys()))

```

```

class Eggs:
    pass

print('tworzenie klasy')
class Spam(Eggs, metaclass=SubMeta()):          # Meta jest normalną instancją klasy
    data = 1                                     # Wywołane na końcu instrukcji
    def meth(self, arg):
        pass

print('tworzenie instancji')
X = Spam()
print('dane:', X.data)

```

Choć powyższe alternatywne rozwiązania działają, większość metaklas spełnia swoje zadania, redefiniując metody `__new__` oraz `__init__` klasy nadzędnej `type`. W praktyce daje to tyle kontroli, ile jest potrzebne, i często jest prostsze od innych sposobów. Jak jednak zobaczymy później, prosta metaklasa oparta na funkcji może działać w dużej mierze tak jak dekorator klasy, co pozwala metaklasom zarządzaćinstancjami oraz klasami.

Instancje a dziedziczenie

Ponieważ metaklasy określane są w sposób podobny do klas nadzędnych dziedziczenia, na pierwszy rzut oka mogą być nieco mylące. Kilka ważnych uwag powinno pomóc podsumować i wyjaśnić ten model:

- **Metaklasy dziedziczą po klasie `type`.** Choć pełnią specjalną rolę, metaklasy tworzone są za pomocą instrukcji `class` i zachowują się zgodnie ze zwykłym modelem programowania zorientowanego obiektowo w Pythonie. Przykładowo jako klasy podzędne `type` mogą redefiniować metody obiektu `type`, w miarę potrzeby nadpisując je i dostosowując do własnych wymagań. Metaklasy zazwyczaj redefiniują metody `__new__` i `__init__` klasy `type` w celu dostosowania tworzenia klasy oraz inicjalizacji do własnych potrzeb, jednak mogą także redefiniować metodę `__call__`, jeśli chcą bezpośrednio przechwytywać wywołanie tworzące klasy na jej końcu. Choć jest to rzadziej spotykane, mogą nawet być prostymi funkcjami zwracającymi dowolne obiekty, a nie tylko klasami podzędnymi `type`.
- **Deklaracje metaklas dziedziczone są przez klasy podzędne.** Deklaracja `metaclass=M` w klasie definiowanej przez użytkownika `dziedziczona` jest także przez jej klasy podzędne, przez co metaklasa zostanie wykonana w trakcie tworzenia każdej klasy dziedziczącej tę specyfikację w łańcuchu klas nadzędnych.
- **Atrybuty metaklas nie są dziedziczone przez instancje klas.** Deklaracje metaklas określają relację *instancji*, która nie jest tym samym, co dziedziczenie. Ponieważ klasy są instancjami metaklas, działanie zdefiniowane w metaklasie ma zastosowanie do klasy, jednak nie do późniejszych instancji klasy. Instancje pozyskują zachowanie ze swoich klas i klas nadzędnych, jednak nie z metaklas. Wyszukiwanie atrybutów instancji przeszukuje jedynie słowniki `__dict__` instancji i wszystkich jej klas. Metaklasa nie jest zawarta w wyszukiwaniu dziedziczenia.

By zilustrować dwa ostatnie punkty, rozważmy poniższy przykład:

```

class MetaOne(type):
    def __new__(meta, classname, supers, classdict): # Redefiniuje metodę klasy type
        print('W MetaOne.new:', classname)
        return type.__new__(meta, classname, supers, classdict)
    def toast(self):
        print('tost')

```

```

class Super(metaclass=MetaOne):
    def spam(self):
        print('mielonka')

class C(Super):
    def eggs(self):
        print('jajka')

X = C()
X.eggs()
X.spam()
X.toast()

```

Metaklasy dziedziczone także przez klasy podrzędne
MetaOne wykonana 2 razy dla 2 klas

Klasa nadrzędna: dziedziczenie a instancja
Klasy dziedziczą po klasach nadrzędnych
Ale nie po metaklasach

Odziedziczone po C
Odziedziczone po Super
Nieodziedziczone po metaklasie

Po wykonaniu powyższego kodu metaklasa obsługuje tworzenie *obu* klas klienta, a instancje dziedziczą atrybuty klas, jednak atrybutów metaklasy już *nie*:

```

W MetaOne.new: Super
W MetaOne.new: C
jajka
mielonka
AttributeError: 'C' object has no attribute 'toast'

```

Choć szczegóły mają znaczenie, istotne jest, by przy zajmowaniu się metaklasami pamiętać o szerokiej perspektywie. Metaklasy takie jak widziane powyżej będą wykonywane automatycznie dla każdej klasy je deklarującej. W przeciwieństwie do omawianych wcześniej funkcji pomocniczych takie klasy automatycznie pobierają wszelkie rozszerzenia udostępniane przez metaklasę. Co więcej, zmiany w takim rozszerzeniu wystarczy zapisać tylko w jednym miejscu — w metaklasie, co znacznie upraszcza wprowadzanie modyfikacji w miarę ewolucji naszych potrzeb. Tak jak wiele innych narzędzi Pythona metaklasy ułatwiają utrzymywanie kodu, eliminując jego powtarzalność. By w pełni wypróbować ich możliwości, musimy jednak przejść do jakichś większych przykładów użycia.

Przykład — dodawanie metod do klas

W tym i kolejnym podrozdziale przestudujemy przykłady dwóch częstych przypadków użycia metaklas — dodawania metod do klasy i automatycznego dekorowania wszystkich metod. Są to tylko dwie z wielu ról metaklas, które niestety pochłoną całe miejsce, jakie zostało nam w niniejszym rozdziale. Bardziej zaawansowane zastosowania metaklas można znaleźć w Internecie. Przykłady te są jednak reprezentatywne dla działania metaklas i wystarczą do zilustrowania podstaw.

Co więcej, oba umożliwiają nam skontrastowanie dekoratorów klas i metaklas — nasz pierwszy przykład porównuje implementacje rozszerzania klas i opakowywania instancji oparte na metaklasach oraz dekoratorach, natomiast drugi najpierw zastosuje dekorator za pomocą metaklasy, a następnie za pomocą innego dekoratora. Jak zobaczymy, te dwa narzędzia są często wymienne, a nawet się dopełniają.

Ręczne rozszerzanie

We wcześniejszej części niniejszego rozdziału przyjrzaliśmy się szkieletowi kodu rozszerzającego klasy poprzez dodawanie do nich metod na różne sposoby. Jak widzieliśmy, proste dziedziczenie oparte na klasach wystarczy, jeśli dodatkowe metody są znane statycznie

w momencie tworzenia klasy. Składanie za pomocą osadzania obiektów może często dać ten sam efekt. W bardziej dynamicznych scenariuszach wymagane są czasami inne techniki. Wystarczające mogą być zazwyczaj funkcje pomocnicze, jednak metaklasy udostępniają jawną strukturę i minimalizują koszty utrzymywania zmian w przyszłości.

Wcielmy zatem te pomysły w życie za pomocą działającego kodu. Rozważmy następujący przykład ręcznego rozszerzania klas. Dodaje on dwie metody do dwóch klas po ich utworzeniu:

Ręczne rozszerzanie — dodawanie nowych metod do klas

```
class Client1:
    def __init__(self, value):
        self.value = value
    def spam(self):
        return self.value * 2

class Client2:
    value = 'ni?'

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'szynka'

Client1.eggs = eggsfunc
Client1.ham = hamfunc

Client2.eggs = eggsfunc
Client2.ham = hamfunc

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bekon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bekon'))
```

Powyższy kod działa, ponieważ metody mogą zawsze być przypisywane do klasy po jej utworzeniu, dopóki przypisywane metody są funkcjami z dodatkowym pierwszym argumentem otrzymującym instancję `self`. Argument ten można wykorzystać do uzyskania dostępu do informacji o stanie dostępnych z instancji klasy, nawet jeśli funkcja definiowana jest niezależnie od klasy.

Po wykonaniu powyższego kodu otrzymujemy wynik metody zapisanej wewnątrz pierwszej klasy, a także dwóch metod dodanych do klas po fakcie:

```
Ni!Ni!
Ni!Ni!Ni!Ni!
bekonszynka
ni?i?ni?ni?
bekonszynka
```

Takie rozwiązanie działa dobrze w wyizolowanych przypadkach i można je wykorzystać do uzupełnienia klasy w dowolny sposób w czasie wykonywania. Ma ono jednak potencjalnie istotną wadę — musimy powtarzać kod rozszerzający dla każdej klasy potrzebującej tych metod. W naszym przypadku dodanie dwóch metod do dwóch klas nie było zbyt pracochłonne,

jednak w bardziej skomplikowanych scenariuszach takie rozwiązanie może pochłaniać sporo czasu i być podatne na błędy. Jeśli kiedykolwiek zapomnimy o robieniu tego w sposób spójny lub będziemy musieli zmienić rozszerzenie, możemy napotkać problemy.

Rozszerzanie oparte na metaklasie

Choć rozszerzanie ręczne działa, w większych programach lepiej byłoby, gdybyśmy mogli automatycznie zastosować takie zmiany do całego zbioru klas. W ten sposób unikamy możliwości zepsucia rozszerzenia dla dowolnej klasy. Co więcej, zapisanie rozszerzenia w jednym miejscu lepiej obsługuje przyszłe zmiany — wszystkie klasy zbioru automatycznie pobiorą modyfikacje.

Jednym ze sposobów spełnienia tego celu jest użycie metaklas. Jeśli zapiszemy rozszerzenie w metaklasie, każda klasa deklarująca tę metaklasę będzie rozszerzona w ten sam, poprawny sposób i automatycznie pobierze wszelkie zmiany dokonane w przyszłości. Demonstruje to poniższy kod:

```
# Rozszerzenie za pomocą metaklas — lepiej obsługuje przyszłe zmiany

def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'szynka'

class Extender(type):
    def __new__(meta, classname, supers, classdict):
        classdict['eggs'] = eggsfunc
        classdict['ham'] = hamfunc
        return type.__new__(meta, classname, supers, classdict)

class Client1(metaclass=Extender):
    def __init__(self, value):
        self.value = value
    def spam(self):
        return self.value * 2

class Client2(metaclass=Extender):
    value = 'ni?'

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bekon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bekon'))
```

Tym razem obie klasy klienta rozszerzane są za pomocą nowych metod, ponieważ są instancjami metaklasy dokonującej rozszerzenia. Po wykonaniu powyższego kodu wynik będzie taki sam jak wcześniej — nie zmienialiśmy tego, co robi kod, po prostu dokonaliśmy refaktoryzacji, tak by rozszerzenie było wykonane w sposób bardziej czysty:

```
Ni!Ni!
Ni!Ni!Ni!Ni!
bekonszynka
ni?ni?ni?ni?
bekonszynka
```

Warto zwrócić uwagę na to, że w tym przykładzie metaklasa nadal wykonuje stosunkowo statyczne zadanie — dodaje dwie znane metody do każdej klasy ją deklarującej. Tak naprawdę, gdyby jedynym naszym zadaniem było zawsze dodawanie tych samych dwóch metod do zbioru klas, równie dobrze moglibyśmy zapisać je w normalnej klasie nadzędnej i odziedziczyć w klasach podrzędnych. W praktyce jednak struktura metaklasy obsługuje także o wiele bardziej dynamiczne zachowania. Przykładowo podmiotową klasę można także skonfigurować w oparciu o dowolną logikę w czasie wykonywania:

Można również skonfigurować klasę w oparciu o testy w czasie wykonywania

```
class MetaExtend(type):
    def __new__(meta, classname, supers, classdict):
        if sometest():
            classdict['eggs'] = eggsfunc1
        else:
            classdict['eggs'] = eggsfunc2
        if someothertest():
            classdict['ham'] = hamfunc
        else:
            classdict['ham'] = lambda *args: 'Nie jest obsługiwane'
        return type.__new__(meta, classname, supers, classdict)
```

Metaklasy a dekoratory klas — runda 2.

Na wypadek gdyby niniejszy rozdział nie spowodował jeszcze u czytających bólu głowy, warto sobie przypomnieć, że dekoratory klas z poprzedniego rozdziału często pokrywają się z metaklasami z tego rozdziału w zakresie funkcjonalności. Wynika to z faktu, że:

- Dekoratory klas dowiązują ponownie nazwy klas do wyniku funkcji na końcu instrukcji `class`.
- Metaklasy działają dzięki przekierowaniu tworzenia obiektu klasy przez obiekt na końcu instrukcji `class`.

Choć są to nieco odmienne modele, w praktyce mogą zazwyczaj spełniać te same cele, choć w różny sposób. Tak naprawdę dekoratory klas można wykorzystać do zarządzania zarówno instancjami klasy, jak i samą klasą. Choć dekoratory mogą zarządzać klasami w sposób dość naturalny, nieco mniej oczywiste jest zarządzanie instancjami przez metaklasy. Metaklasy najlepiej przydają się do zarządzania obiektami klas.

Rozszerzenie oparte na dekoratorach

Przykład z metaklasami z poprzedniego podrozdziału, dodający metody do klasy w czasie tworzenia, można także zapisać w postaci dekoratora klasy. W tym trybie dekoratory mniej więcej odpowiadają metodzie `__init__` metaklas, ponieważ obiekt klasy został już utworzony, zanim wywołany zostaje dekorator. Podobnie jak w przypadku metaklas typ oryginalnej klasy zostaje zachowany, ponieważ nie zostaje wstawiona żadna warstwa obiektu opakowującego. Wynik poniższego kodu będzie taki sam jak poprzedniego kodu z metaklasą:

Rozszerzenie za pomocą metaklasy — to samo co udostępnienie __init__ w metaklasie

```
def eggsfunc(obj):
    return obj.value * 4

def hamfunc(obj, value):
    return value + 'szynka'
```

```

def Extender(aClass):
    aClass_eggs = eggsfunc
    aClass_ham = hamfunc
    return aClass

@Extender
class Client1:
    def __init__(self, value):
        self.value = value
    def spam(self):
        return self.value * 2

@Extender
class Client2:
    value = 'ni?'

X = Client1('Ni!')
print(X.spam())
print(X.eggs())
print(X.ham('bekon'))

Y = Client2()
print(Y.eggs())
print(Y.ham('bekon'))

```

Zarządza klasą, a nie instancją
Odpowiednik __init__ metaklasy

Client1 = Extender(Client1)
Dowiązane ponownie na końcu instrukcji class

X jest instancją Client1

Innymi słowy, przynajmniej w pewnych przypadkach dekoratory potrafią zarządzać klasami równie łatwo, co metaklasy. Odwrotna zależność nie jest jednak tak oczywista — metaklasy można wykorzystać do zarządzania instancjami, jednak jedynie z użyciem pewnej dozy sztuczek. Zademonstrujemy to w kolejnym podrozdziale.

Zarządzanie instancjami zamiast klasami

Jak widzieliśmy przed chwilą, dekoratory klas mogą pełnić tę samą rolę w zakresie *zarządzania klasami*, co metaklasy. Metaklasy często spełniają także tę samą rolę, co dekoratory w zakresie *zarządzania instancjami*, jednak jest to nieco bardziej skomplikowane. A zatem:

- Dekoratory klas mogą zarządzać zarówno klasami, jak i instancjami.
- Metaklasy mogą zarządzać zarówno klasami, jak i instancjami, jednak instancje wymagają dodatkowej pracy.

To powiedziawszy, pewne aplikacje lepiej jest pisać w jeden lub drugi sposób. Rozważmy na przykład poniższy przypadek dekoratora klas z poprzedniego przykładu. Wykorzystywany jest on do wyświetlania komunikatu śledzenia za każdym razem, gdy dowolny, mający normalną nazwę atrybut instancji klasy jest pobierany:

```

# Dekorator klasy śledzi zewnętrzne pobrania atrybutów instancji

def Tracer(aClass): # Dla dekoratora @
    class Wrapper:
        def __init__(self, *args, **kargs):
            self.wrapped = aClass(*args, **kargs)
        def __getattr__(self, attrname):
            print('Śledzenie:', attrname)
            return getattr(self.wrapped, attrname)
    return Wrapper

@Tracer
class Person:
    def __init__(self, name, hours, rate):
        self.name = name

```

W momencie tworzenia instancji
Użycie nazwy zakresu obejmującego

Przechwytuje wszystko poza .wrapped
Deleguje do obiektu wrapped

Person = Tracer(Person)
Wrapper pamięta Person

```

        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

# Pobranie wewnętrz metod nie jest śledzone

bob = Person('Robert', 40, 50)
# bob to tak naprawdę Wrapper
print(bob.name)                      # Wrapper osadza Person
print(bob.pay())                       # Wywołuje __getattr__

```

Po wykonaniu powyższego kodu dekorator wykorzystuje ponowne dowiązanie nazwy w celu opakowania obiektów instancji w obiekt, który tworzy wiersze śledzenia w poniższym wyniku:

```

Śledzenie: name
Robert
Śledzenie: pay
2000

```

Choć metaklasa może dać nam ten sam efekt, z konceptualnego punktu widzenia wydaje się to mniej oczywiste. Metaklasy zaprojektowane są specjalnie w celu zarządzania tworzeniem obiektów klas i mają interfejs przystosowany do tego celu. By wykorzystać metaklasę do zarządzania instancjami, musimy w pewnej mierze polegać na magii. Poniższa metaklasa daje ten sam efekt i dane wyjściowe, co poprzedni dekorator:

Zarządzanie instancjami jak w poprzednim przykładzie, jednak za pomocą metaklasy

```

def Tracer(classname, supers, classdict):
    aClass = type(classname, supers, classdict)
    class Wrapper:
        def __init__(self, *args, **kargs):
            self.wrapped = aClass(*args, **kargs)
        def __getattribute__(self, attrname):
            print('Śledzenie:', attrname)
            return getattr(self.wrapped, attrname)
    return Wrapper

class Person(metaclass=Tracer):
    def __init__(self, name, hours, rate):
        self.name = name
        self.hours = hours
        self.rate = rate
    def pay(self):
        return self.hours * self.rate

# W momencie wywołania tworzącego klasę
# Utworzenie klasy klienta
# W momencie tworzenia instancji
# Przechwytyuje wszystko poza .wrapped
# Deleguje do obiektu wrapped
# Tworzy Person za pomocą Tracer
# Wrapper pamięta Person
# Pobranie wewnętrz metod nie jest śledzone
# bob to tak naprawdę Wrapper
# Wrapper osadza Person
# Wywołuje __getattr__

```

Takie rozwiązanie działa, jednak opiera się na dwóch sztuczках. Po pierwsze, musi wykorzystywać prostą funkcję zamiast klasy, ponieważ klasy podzielone `type` muszą się stosować do pewnych protokołów tworzenia obiektów. Po drugie, musi ręcznie tworzyć klasę podmiotową, wywołując `type`. Musi zwrócić opakowanie instancji, jednak metaklasy są także odpowiedzialne za tworzenie i zwracanie klasy podmiotowej. Tak naprawdę protokół metaklasy wykorzystujemy w tym przykładzie w taki sposób, by imitował on dekoratory, a nie odwrotnie. Ponieważ oba wykonywane są na końcu instrukcji `class`, w wielu rolach są one jedynie wariacjami na jeden temat. Powyższa wersja kodu z metaklasami daje po wykonaniu takie same dane wyjściowe, co dekorator:

```

Śledzenie: name
Robert
Śledzenie: pay
2000

```

Warto przestudiować obie wersje tego przykładu samodzielnie, by rozważyć ich wady i zalety. Generalnie jednak metaklasy z uwagi na swój projekt chyba najlepiej nadają się do zarządzania klasami. Dekoratory klas mogą zarządzać albo instancjami, albo klasami, choć mogą nie być najlepszą opcją w przypadku bardziej zaawansowanych ról pełnionych przez metaklasy, na które nie mamy w niniejszej księdze miejsca. Osoby, które chcą się dowiedzieć więcej o dekoratorach oraz metaklasach, po skończeniu lektury niniejszego rozdziału odsyłam do Internetu lub dokumentacji standardowej Pythona. W kolejnym, kończącym niniejszy rozdział podrozdziale omówimy jeszcze jeden częsty przypadek użycia — automatyczne zastosowanie operacji do metod klas.

Przykład — zastosowanie dekoratorów do metod

Jak widzieliśmy w poprzednim podrozdziale, ponieważ metaklasy oraz dekoratory wykonywane są na końcu instrukcji `class`, często można je wykorzystywać *zamiennie*, choć z użyciem innej składni. Wybór pomiędzy tymi dwoma rozwiązaniami jest w wielu kontekstach dowolny. Można także wykorzystywać oba narzędzia *razem*, jako wzajemne dopełnienie. W niniejszym podrozdziale omówimy przykład tego typu połączenia — zastosowanie dekoratora funkcji do wszystkich metod klasy.

Ręczne śledzenie za pomocą dekoracji

W poprzednim rozdziale utworzyliśmy dwa dekoratory funkcji — jeden śledzący i zliczający wszystkie wywołania wykonywane do dekorowanej funkcji i drugi do obliczania czasu takich wywołań. Przybierały one różne formy; część z nich miała zastosowanie zarówno do funkcji, jak i metod, inne nie. Poniższa wersja zbiera ostateczne formy obu dekoratorów w pliku modułu, tak by można było je wykorzystać ponownie i tu się do nich odwołać.

```
# Plik mytools.py — wybrane narzędzia dekoratorów

def tracer(func):
    calls = 0
    def onCall(*args, **kwargs):
        nonlocal calls
        calls += 1
        print('wywołanie %s %s' % (calls, func.__name__))
        return func(*args, **kwargs)
    return onCall

import time
def timer(label='', trace=True):
    def onDecorator(func):
        def onCall(*args, **kargs):
            start = time.clock()
            result = func(*args, **kargs)
            elapsed = time.clock() - start
            onCall.alltime += elapsed
            if trace:
                format = '%s%s: %.5f, %.5f'
                values = (label, func.__name__, elapsed, onCall.alltime)
                print(format % values)
            return result
        onCall.alltime = 0
        return onCall
    return onDecorator
```

Użycie z `__call__` funkcji, a nie klasy
Inaczej `self` będzie jedynie instancją dekoratora

Dla argumentów dekoratora — zachowanie argumentów
Dla @ — zachowanie udekorowanej funkcji
W momencie wywołania — wywołanie oryginału
Stan to zakresy + atrybuty funkcji

Jak wiemy z poprzedniego rozdziału, by użyć dekoratorów w sposób ręczny, po prostu importujemy je z modułu i piszemy kod ze składnią @ dekoratora przed każdą metodą, którą chcemy śledzić czy zmierzyć:

```
from mytools import tracer

class Person:
    @tracer
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay

    @tracer
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)          # giveRaise = tracer(giveRaise)
                                                # onCall pamięta giveRaise

    @tracer
    def lastName(self):
        return self.name.split()[-1]         # lastName = tracer(lastName)

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
print(bob.name, anna.name)
anna.giveRaise(.10)                      # Wykonuje onCall(anna, .10)
print(anna.pay)
print(bob.lastName(), anna.lastName())   # Wykonuje onCall(bob), pamięta lastName
```

Po wykonaniu powyższego kodu otrzymujemy następujące dane wyjściowe. Wywołania udekorowanych metod przekierowywane są do logiki przechwytyjącej, a następnie delegującej wywołanie, ponieważ oryginalne nazwy metod zostały dowiązane do dekoratora:

```
wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona
```

Śledzenie metaklas oraz dekoratorów

Ręczny schemat dekoracji z poprzedniego podrozdziału działa, jednak wymaga dodania składni dekoracji przed każdą metodą, którą chcemy śledzić, a następnie późniejszego usunięcia tej składni, kiedy śledzenie nie będzie już pożądane. Jeśli chcemy śledzić każdą metodę klasy, w większych programach może się to stać dość żmudne. Byłoby lepiej, gdybyśmy mogli w jakiś sposób automatycznie zastosować dekorator tracer do wszystkich metod klasy.

W przypadku metaklas możemy to zrobić — ponieważ są one wykonywane, kiedy klasa jest tworzona, są naturalnym miejscem na dodanie opakowań dekorujących do metod klas. Przeglądając słownik atrybutów klasy i sprawdzając istnienie w nim obiektów funkcji, możemy automatycznie wykonać metody za pośrednictwem dekoratora i ponownie dowiązać oryginalne nazwy do wyników. Efekt będzie taki sam, jak w przypadku automatycznego ponownego dowiązania nazw metod za pomocą dekoratorów, jednak możemy go zastosować w bardziej globalny sposób:

```

# Metaklasa dodająca dekorator śledzący do każdej metody klasy klienta

from types import FunctionType
from mytools import tracer

class MetaTrace(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:
                classdict[attr] = tracer(attrval)
        return type.__new__(meta, classname, supers, classdict)

class Person(metaclass=MetaTrace):
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
print(bob.name, anna.name)
anna.giveRaise(.10)
print(anna.pay)
print(bob.lastName(), anna.lastName())

```

Po wykonaniu powyższego kodu wyniki są takie jak poprzednio. Wywołania metod przekierowywane są najpierw do dekoratora śledzącego, a później propagowane do oryginalnej metody:

```

wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona

```

Widoczny powyżej wynik jest kombinacją działania dekoratora i metaklasy. Metaklasa automatycznie stosuje dekorator funkcji do każdej metody klasy w czasie tworzenia, a dekorator funkcji automatycznie przechwytuje wywołania metod w celu wyświetlenia komunikatów śledzenia w danych wyjściowych. Ta kombinacja po prostu działa dzięki uniwersalności obu narzędzi.

Zastosowanie dowolnego dekoratora do metod

Poprzedni przykład z metaklasą działa jedynie dla jednego określonego dekoratora funkcji — śledzenia. Uogólnienie kodu w taki sposób, by można było zastosować dowolny dekorator do wszystkich metod klasy, jest jednak trywialne. Wystarczy dodać zewnętrzną warstwę zakresu, by zachować pożądany dekorator, ponownie jak robiliśmy to w przypadku dekoratorów z poprzedniego rozdziału. W poniższym kodzie zapisano na przykład takie uogólnienie, a następnie wykorzystano je w celu ponownego zastosowania dekoratora śledzącego:

```

# Fabryka metaklas — zastosowanie dowolnego dekoratora do wszystkich metod klasy

from types import FunctionType
from mytools import tracer, timer

def decorateAll(decorator):

```

```

class MetaDecorate(type):
    def __new__(meta, classname, supers, classdict):
        for attr, attrval in classdict.items():
            if type(attrval) is FunctionType:
                classdict[attr] = decorator(attrval)
        return type.__new__(meta, classname, supers, classdict)
    return MetaDecorate

class Person(metaclass=decorateAll(tracer)): # Zastosowanie dekoratora do wszystkich metod
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
print(bob.name, anna.name)
anna.giveRaise(.10)
print(anna.pay)
print(bob.lastName(), anna.lastName())

```

Po wykonaniu kodu w takiej postaci wynik będzie taki sam jak w poprzednich przykładach — nadal dekorujemy każdą metodę klasy klienta dekoratorem funkcji `tracer`, jednak robimy to w sposób bardziej ogólny:

```

wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0
wywołanie 1 lastName
wywołanie 2 lastName
Zielony Czerwona

```

By teraz zastosować inny dekorator do metod, możemy po prostu zastąpić nazwę dekoratora w wierszu nagłówka klasy. By użyć na przykład pokazanego wyżej dekoratora funkcji `timer`, moglibyśmy wykorzystać dowolny z dwóch ostatnich wierszy nagłówka w poniższym kodzie, definiując naszą klasę — pierwszy przyjmuje domyślne argumenty dekoratora `timer`, natomiast drugi określa tekst podpisu:

```

class Person(metaclass=decorateAll(tracer)): # Zastosowanie dekoratora tracer
class Person(metaclass=decorateAll(timer())): # Zastosowanie dekoratora timer, argumenty domyślne
class Person(metaclass=decorateAll(timer(label='***'))): # Argumenty dekoratora

```

Warto zauważyć, że to rozwiązanie nie może obsługiwać niedomyślnych argumentów dekoratora różniących się dla poszczególnych metod, jednak można przekazać argumenty dekoratora mające zastosowanie dla wszystkich metod, jak wyżej. By to przetestować, wykorzystamy ostatnią z powyższych deklaracji metaklas w celu zastosowania dekoratora zliczającego czas i dodamy poniższe wiersze na końcu skryptu:

```

# Przy użyciu dekoratora timer — całkowity czas dla metody

print('*'*40)
print('.%f' % Person.__init__.alltime)
print('.%f' % Person.giveRaise.alltime)
print('.%f' % Person.lastName.alltime)

```

Nowy wynik będzie następujący. Metaklasa opakowuje teraz metody w dekoratory zliczające czas, dzięki czemu możemy dla każdej metody klasy zobaczyć, ile trwa każde wywołanie:

```

**__init__: 0.00001, 0.00001
**__init__: 0.00001, 0.00002
Robert Zielony Anna Czerwona
**giveRaise: 0.00001, 0.00001
110000.0
**lastName: 0.00001, 0.00001
**lastName: 0.00001, 0.00002
Zielony Czerwona
-----
0.00002
0.00001
0.00002

```

Metaklasy a dekoratory klas — runda 3.

Dekoratory klas również w tym punkcie pokrywają się w metaklasami. Poniższa wersja kodu zastępuje metaklasę z poprzedniego przykładu dekoratorem klasy. Definiuje i wykorzystuje *dekorator klasy*, który zastosuje dekorator funkcji do wszystkich metod klasy. Choć poprzednie zdanie może brzmieć bardziej jak instrukcja zen niż techniczny opis, wszystko to działa dość naturalnie — dekoratory Pythona obsługują dowolne zagnieżdżanie i kombinacje:

```

# Fabryka dekoratora klas — zastosowanie dowolnego dekoratora do wszystkich metod klas
from types import FunctionType
from mytools import tracer, timer

def decorateAll(decorator):
    def DecoDecorate(aClass):
        for attr, attrval in aClass.__dict__.items():
            if type(attrval) is FunctionType:
                setattr(aClass, attr, decorator(attrval)) # Nie __dict__
        return aClass
    return DecoDecorate

@decorateAll(tracer)
class Person:
    def __init__(self, name, pay):
        self.name = name
        self.pay = pay
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)
    def lastName(self):
        return self.name.split()[-1]

bob = Person('Robert Zielony', 50000)
anna = Person('Anna Czerwona', 100000)
print(bob.name, anna.name)
anna.giveRaise(.10)
print(anna.pay)
print(bob.lastName(), anna.lastName())

```

*# Użycie dekoratora klasy
Stosuje dekorator funkcji do metod
Person = decorateAll(..)(Person)
Person = DecoDecorate(Person)*

Po wykonaniu kodu w powyższej postaci dekorator klasy zastosuje dekorator funkcji `tracer` do każdej metody i przy wywołaniu zwróci komunikaty śledzenia (wynik będzie taki sam jak w poprzedniej wersji tego przykładu z metaklasą):

```

wywołanie 1 __init__
wywołanie 2 __init__
Robert Zielony Anna Czerwona
wywołanie 1 giveRaise
110000.0

```

```
wywołanie 1 lastName  
wywołanie 2 lastName  
Zielony Czerwona
```

Warto zauważyc, że dekorator klasy zwraca oryginalną, rozszerzoną klasę, a nie jej warstwę opakowującą (co często się zdarza, kiedy zamiast tego opakowuje się obiekty instancji). W przypadku wersji z metaklasą zachowujemy typ oryginalnej klasy — instancja Person jest instancją klasy Person, a nie jakiejś klasy opakowującej. Tak naprawdę ten dekorator klasy zajmuje się jedynie tworzeniem klasy. Wywołania tworzące instancje nie są w ogóle przechwytywane.

To rozróżnienie ma znaczenie w programach, które wymagają sprawdzania typów dla instancji w celu zwrocenia oryginalnej klasy, a nie opakowania. Przy rozszerzaniu klasy zamiast instancji dekoratory klas mogą zachować typ oryginalnej klasy. Metody klas nie są ich oryginalnymi funkcjami, ponieważ są ponownie dowiązywane do dekoratorów, jednak w praktyce jest to mniej istotne; w rozwiązaniu z metaklasą jest zresztą tak samo.

Warto również zauważyc, że tak jak w wersji z metaklasą struktura ta nie może obsługiwać argumentów dekoratorów funkcji różniących się dla poszczególnych metod, jednak jest w stanie obsługiwać takie argumenty, jeśli zostaną one zastosowane do wszystkich metod. By na przykład użyć tego schematu do zastosowania dekoratora zliczającego czas, wystarczający będzie jeden z dwóch ostatnich wierszy zaprezentowanych poniżej, wstawiony tuż przed definicją klasy — pierwszy z nich wykorzystuje domyślne argumenty dekoratora, natomiast drugi przekazuje argument w sposób jawny:

```
@decorateAll(tracer)          # Udekorowanie wszystkiego za pomocą dekoratora tracer  
@decorateAll(timer())          # Udekorowanie wszystkiego za pomocą dekoratora timer, argumenty domyślne  
@decorateAll(timer(label='@@')) # To samo, ale z przekazanym argumentem dekoratora
```

Tak jak wcześniej użyjemy ostatniego wiersza z dekoratorem i dodamy następujący kod na końcu skryptu w celu przetestowania naszego przykładu z innym dekoratorem:

```
# Przy użyciu dekoratora timer — całkowity czas dla metody
```

```
print('*'*40)  
print('%.5f' % Person.__init__.alltime)  
print('%.5f' % Person.giveRaise.alltime)  
print('%.5f' % Person.lastName.alltime)
```

Widoczny będzie ten sam wynik — dla każdej metody zliczamy czas każdego z wywołań, jednak tym razem do dekoratora timer przekazaliśmy inny argument podpisu:

```
@@__init__: 0.00001, 0.00001  
@@__init__: 0.00001, 0.00002  
Robert Zielony Anna Czerwona  
@@giveRaise: 0.00001, 0.00001  
110000.0  
@@lastName: 0.00001, 0.00001  
@@lastName: 0.00001, 0.00002  
Zielony Czerwona  
-----  
0.00002  
0.00001  
0.00002
```

Jak widać, metaklasy i dekoratory klas są nie tylko wymienne, ale często się także dopełniają. Oba rozwiązania udostępniają zaawansowane sposoby dostosowywania obiektów i klas do własnych potrzeb, a także zarządzania nimi, ponieważ pozwalają na wstawianie kodu do procesu

tworzenia klas. Choć niektóre bardziej zaawansowane zastosowania mogą być zapisane w kodzie z użyciem jednego bądź drugiego rozwiązania, wybór lub połączenie tych dwóch narzędzi w dużej mierze pozostaje w gestii programisty.

„Opcjonalne” możliwości języka

Na początku tego rozdziału zamieściłem cytat dotyczący metaklas, mówiący, że dla 99% programistów Pythona są one mało interesujące, by podkreślić relatywną egzotyczność tego narzędzia. To stwierdzenie nie jest jednak do końca dokładne, i to nie tylko statystycznie.

Autorem tego cytatu jest mój przyjaciel z najwcześniejszych dni Pythona, a ja sam nie chcę nikomu niczego niesprawiedliwie wytykać. Co więcej, sam często czynię komentarze na temat tego, że jakaś opcja języka jest egzotyczna — tak naprawdę nawet w niniejszej książce.

Problem polega jednak na tym, że takie stwierdzenia mają zastosowanie jedynie do osób, które pracują same i używają jedynie kodu napisanego samodzielnie. Kiedy tylko jakaś „opcjonalna” możliwość języka zostanie użyta przez *kogokolwiek* w jakiejś organizacji, przestaje być opcjonalna — jest właściwie narzucona *każdemu* członkowi tej organizacji. Tak samo jest w przypadku oprogramowania zewnętrznego wykorzystywanego w naszych systemach — jeśli autor tego oprogramowania wykorzystuje jakąś zaawansowaną opcję języka, przestaje ona być dla nas opcjonalna, ponieważ by używać kodu lub go modyfikować, musimy ją opanować.

Takie spostrzeżenie ma zastosowanie do wszystkich narzędzi zaawansowanych wymienionych na początku niniejszego rozdziału — dekoratorów, właściwości, deskryptorów czy metaklas. Jeśli jakaś osoba lub program, z którymi musimy współpracować, z nich korzystają, narzędzia te automatycznie stają się elementem wymaganym, który musimy sobie przyswoić. Oznacza to, że *nic nie jest naprawdę „opcjonalne”, skoro nic nie jest faktycznie opcjonalne*. Większość z nas nie może sobie wybierać tego, co nam się podoba.

Dlatego właśnie niektórzy weterani Pythona (ze mną na czele) czasami podnoszą lament, że Python z czasem wydaje się rosnąć i stawać się coraz bardziej skomplikowany. Nowe opcje dodawane przez weteranów wydają się z kolei podnosić intelektualną poprzeczkę dla nowicjuszy. Choć podstawy jądra Pythona, jakie jak typy dynamiczne czy typy wbudowane, pozostały w zasadzie bez zmian, opcje zaawansowane mogą się stać wymaganą wiedzą dla każdego programisty Pythona. Z tego właśnie powodu zdecydowałem się omówić tutaj te zagadnienia, pomimo pomijania ich w poprzednich wydaniach książki.

Nie da się pominąć zaawansowanych tematów, jeśli znajdują się one w kodzie, który musimy zrozumieć. Z drugiej strony wiele osób dopiero się uczących może przyswoić sobie te tematy dopiero wtedy, gdy będzie to niezbędne. I szczerze mówiąc, programiści aplikacji zdają się spędzać większość czasu na pracy z *bibliotekami i rozszerzeniami*, a nie zaawansowanymi i czasami dość egzotycznymi opcjami języka. Przykładowo w książce *Programming Python*, będącej kontynuacją niniejszej, w większości zajmujemy się połączeniem Pythona i bibliotek aplikacji służących do zadań takich jak graficzne interfejsy użytkownika czy Internet, a nie egzotycznymi narzędziami z tego języka.

Skutkiem ubocznym tego rozwoju jest to, że Python ma *coraz większe możliwości*. Kiedy się ich używa w sposób poprawny, narzędzia takie jak dekoratory i metaklasy są nie tylko „modne”, ale pozwalają też kreatywnym programistom tworzyć bardziej elastyczne i użyteczne API, z których mogą korzystać inni programiści. Jak widzieliśmy, mogą także być dobrym rozwiązaniem dla problemów związanych z hermetyzacją i utrzymywaniem kodu.

O tym, czy uzasadnia to potencjalne rozszerzenie wymaganej wiedzy z Pythona, każdy musi zadecydować sam. Niestety poziom umiejętności często decyduje o tym za nas — programiści bardziej zaawansowani lubią bardziej zaawansowane narzędzia i często zdają się zapominać o wpływie swoich wyborów na pozostałe grupy osób. Na szczęście jednak nie jest to zawsze prawdziwe. Dobrzy programiści rozumieją także, że *dobra inżynieria to również prostota*, a z zaawansowanych narzędzi należy korzystać tylko wtedy, gdy jest to niezbędne. Jest tak w każdym języku programowania, choć w szczególności w języku takim jak Python, który często udostępniany jest nowicjuszom jako narzędzie do rozszerzania aplikacji.

Jeśli ten argument nie jest przekonujący, warto przypomnieć, że wielu użytkowników Pythona nie czuje się dobrze nawet z podstawami programowania zorientowanego obiektywo oraz klas. Można mi w tym punkcie zaufać — spotkałem tysiące takich osób. Systemy oparte na Pythonie, które wymagają, by ich użytkownik opanował niuanse metaklas, dekoratorów i podobnych narzędzi, powinny najprawdopodobniej odpowiednio wyważyc swoje oczekiwania w zakresie oczekiwani rynku.

Podsumowanie rozdziału

W niniejszym rozdziale omówiliśmy metaklasy, a także przyjrzaliśmy się przykładom ich użycia. Metaklasy umożliwiają nam wejście w protokół tworzenia klas w Pythonie w celu zarządzania klasami zdefiniowanymi przez użytkownika lub rozszerzenia ich. Ponieważ pozwalają one automatyzować ten proces, mogą być lepszym rozwiązaniem dla twórców API od kodu pisanej ręcznie czy funkcji pomocniczych. Ponieważ hermetyzują kod tego typu, mogą zminimalizować koszty utrzymywania kodu o wiele lepiej od innych rozwiązań.

Przy okazji widzieliśmy także, że role dekoratorów klas oraz metaklas często się pokrywają. Ponieważ oba narzędzia wykonywane są na końcu instrukcji `class`, czasami można ich używać wymiennie. Dekoratory klas można wykorzystać do zarządzania obiektami zarówno klas, jak i instancji. Metaklasy także to potrafią, choć są przeznaczone bardziej dla klas.

Ponieważ niniejszy rozdział omawiał zagadnienie zaawansowane, przejdziemy przez tylko kilka pytań quizu omawiających podstawy (każdy, kto w rozdziale o metaklasach dotarł aż tak daleko, zasługuje na dodatkową nagrodę!). A ponieważ jest to ostatnia część książki, opuścimy także ćwiczenia kończące tę część. Polecam zajrzenie teraz do dodatków zawierających wskazówki dotyczące instalacji, a także rozwiązań ćwiczeń z poprzednich części książki.

Po skończeniu quizu oficjalnie dochodzimy do końca niniejszej książki. Po poznaniu Pythona na wylot kolejnym krokiem powinno być zapoznanie się z bibliotekami, technikami i narzędziami dostępnym w dziedzinie aplikacji, w której chcemy pracować. Ponieważ Python jest tak szeroko wykorzystywany, można znaleźć mnóstwo zasobów dotyczących wykorzystywania go w prawie dowolnej dziedzinie, jaką można sobie wymyślić — od graficznych interfejsów użytkownika po Internet, bazy danych, programowanie numeryczne, robotykę i administrowanie systemami.

W tym właśnie punkcie Python staje się prawdziwą zabawą, jednak tutaj także kończy się niniejsza książka, a zaczynają inne. Wskazówki dotyczące tego, gdzie warto się skierować po skończeniu lektury tej książki, znajdują się w „Przedmowie” w postaci listy polecanych tekstów. Powodzenia! I oczywiście: „Zawsze patrz na życie z humorem”!

Sprawdź swoją wiedzę — quiz

1. Czym jest metaklasa?
2. W jaki sposób deklaruje się metaklasę klasy?
3. W jaki sposób dekoratory klas pokrywają się z metaklasami w zakresie zarządzania klasami?
4. W jaki sposób dekoratory klas pokrywają się z metaklasami w zakresie zarządzania instancjami?
5. Co można zaliczyć do głównych metod programisty Pythona — dekoratory klas czy metaklasy? (Odpowiedź proszę podać zgodnie z popularnym skeczem Monty Pythona).

Sprawdź swoją wiedzę — odpowiedzi

1. Metaklasa to klasa wykorzystywana do utworzenia klasy. Normalne klasy domyślnie są instancjami klasy `type`. Metaklasy są zazwyczaj klasami podrzędnymi `type`, redefiniującymi protokół tworzenia klas w celu dostosowania wywołania tworzącego klasę do własnych potrzeb na końcu instrukcji `class`. Zazwyczaj redefiniują one metody `__new__` oraz `__init__` w celu uzyskania dostępu do protokołu tworzenia klas. Metaklasy można także zapisywać w inny sposób — na przykład jako proste funkcje — jednak są one odpowiedzialne za tworzenie i zwracanie obiektów dla nowej klasy.
2. W Pythonie 3.0 należy wymienić pożądaną metaklasę jako argument ze słowem kluczowym w nagłówku instrukcji `class` — w postaci `class C(metaclass=M)`. W Pythonie 2.X należy zamiast tego użyć atrybutu klasy — formy `__metaclass__ = M`. W wersji 3.0 wiersz nagłówka klasy może także wymieniać normalne klasy nadzędne (inaczej klasy bazowe) przed argumentem ze słowem kluczowym `metaclass`.
3. Ponieważ obie wersje są wywoływanie automatycznie na końcu instrukcji `class`, dekoratory klas oraz metaklasy mogą być wykorzystane do zarządzania klasami. Dekoratory dowiązują ponownie nazwy klas do wyników obiektów wywoływalnych, natomiast metaklasy przekierowują tworzenie klasy do obiektu wywoływalnego, jednak oba punkty zaczepienia można wykorzystywać w podobnych celach. By zarządzać klasami, dekoratory po prostu rozszerzają i zwracają oryginalny obiekt klasy. Metaklasy rozszerzają klasę po utworzeniu jej.
4. Ponieważ obie wersje wywoływanie są automatycznie na końcu instrukcji `class`, dekoratory klas oraz metaklasy mogą być wykorzystane do zarządzania instancjami klas, wstawiając obiekt opakowujący, który przechwytuje wywołania tworzące instancje. Dekoratory mogą dowiązywać ponownie nazwę klasy do obiektu wywoływalnego wykonywanego w momencie tworzenia instancji, który zachowuje oryginalny obiekt klasy. Metaklasy mogą robić to samo, jednak muszą także tworzyć obiekt klasy, dlatego ich użycie jest w tej roli nieco bardziej skomplikowane.
5. Nasza główna metoda to dekoratory... dekoratory i metaklasy... metaklasy i dekoratory... Dwie metody: metaklasy i dekoratory... i bezwzględna skuteczność... Trzy metody: metaklasy, dekoratory i bezwzględna skuteczność... i niemalże fanatyczne oddanie Guido... Cztery... Nie... Wśród naszych metod... Wśród naszych metod są takie elementy jak... metaklasy, dekoratory... Wejdziemy jeszcze raz...

Dodatki

Instalacja i konfiguracja

W niniejszym dodatku znajdują się wskazówki dotyczące instalacji i konfiguracji Pythona, które mogą być pomocne nowicjuszom.

Instalowanie interpretera Pythona

Ponieważ do wykonywania skryptów Pythona potrzebny jest interpreter tego języka, pierwszym krokiem do używania Pythona jest zazwyczaj zainstalowanie go. O ile Python nie jest jeszcze dostępny na naszym komputerze, będziemy musieli go pobrać, zainstalować i prawdopodobnie skonfigurować jakąś aktualną wersję. Wykonuje się to na każdym komputerze tylko raz, a jeśli będziemy wykonywać zamrożone pliki binarne (opisane w rozdziale 2.) bądź korzystać z samoinstalującego się systemu, być może nie będziemy musieli robić nic więcej.

Czy Python jest już zainstalowany?

Zanim zrobimy cokolwiek innego, należy sprawdzić, czy nie mamy już na komputerze zainstalowanej jakiejś aktualnej wersji Pythona. Osoby pracujące na systemach operacyjnych Linux, Mac OS X czy niektórych systemach uniksowych mają najprawdopodobniej Pythona już zainstalowanego, choć może on być o jedno czy dwa wydania starszy od najbliższego. Oto, jak można to sprawdzić:

- W systemie Windows należy sprawdzić, czy w menu *Wszystkie programy* przycisku *Start* (po lewej stronie ekranu na dole) nie ma wpisu *Python*.
- W systemie Mac OS X należy otworzyć okno Terminala (*Aplikacje/Narzędzia/Terminal*) i w powłoce wpisać *python*.
- W systemach Linux i Unix należy wpisać *python* w powłoce systemowej (inaczej nazywanej oknem terminala) w celu sprawdzenia, co się stanie. Alternatywnie można spróbować wyszukać słowa „*python*” w typowych lokalizacjach — */usr/bin*, */usr/local/bin* i podobnych.

Jeśli znajdziemy Pythona, należy się upewnić, że jest to najnowsza wersja. Choć na potrzeby większej części niniejszej książki wystarczy dowolna aktualna wersja, to wydanie koncentruje się na Pythonie 3.0 i 2.6, dlatego by wykonać niektóre przykłady z książki, potrzebna będzie jedna z tych wersji.

A skoro już mowa o wersjach, osobom uczącym się Pythona od podstaw i niemuszącym pracować z istniejącym kodem w wersji 2.X polecam zacząć od wersji 3.0 lub nowszej. W innym przypadku należy korzystać z Pythona 2.6. Niektóre popularne systemy oparte na Pythonie nadal wykorzystują jeszcze starsze wydania (wersja 2.5 jest ciągle bardzo rozpowszechniona), dlatego osoby pracujące z istniejącymi systemami muszą się upewnić, że korzystają z wersji odpowiadającej ich potrzebom. Poniżej opisane są miejsca, z których można pobrać różne wersje Pythona.

Skąd pobrać Pythona

Jeśli na naszym komputerze Python nie jest zainstalowany, będziemy musieli sami go zainstalować. Dobra wiadomość jest taka, że Python jest systemem open source, dostępnym za darmo w Internecie i na większości platform bardzo łatwym do zainstalowania.

Najnowsze i najlepsze standardowe wydanie Pythona można zawsze pobrać ze strony <http://www.python.org>, oficjalnej witryny internetowej Pythona. Na tej stronie należy szukać odnośnika *Downloads*; później wystarczy wybrać platformę, na której będziemy pracować. Można tam znaleźć między innymi gotowy instalator przeznaczony dla systemu Windows (w celu zainstalowania należy go wykonać), pliki Installer Disk Images dla systemu Mac OS X (instalowane zgodnie z regułami komputerów Mac) czy dystrybucje pełnych kodów źródłowych (zazwyczaj komplikowane w systemach Linux, Unix lub OS X w celu wygenerowania interpretera).

Choć Python jest obecnie w systemie Linux standardem, w Internecie można także znaleźć pliki RPM przeznaczone dla tego systemu (należy je rozpakować za pomocą narzędzia *rpm*). Na stronie internetowej Pythona znajdują się również odnośniki do stron zewnętrznych, na których można znaleźć utrzymywane przez Python.org lub podmioty zewnętrzne wersje przeznaczone dla innych platform. Pakiety Pythona można także znaleźć za pomocą wyszukiwarki internetowej. Wśród innych platform znajdziemy gotowe pliki Pythona przeznaczone dla urządzeń takich, jak iPod, Palm, telefony komórkowe firmy Nokia, PlayStation i PSP, systemy Solaris, AS/400 czy Windows Mobile.

Osoby troskające się o środowiskiem Uniksa na komputerze z systemem Windows mogą także chcieć zainstalować bibliotekę *Cygwin* wraz z jej wersją Pythona (więcej informacji na ten temat można znaleźć na stronie <http://www.cygwin.com>). Cygwin to biblioteka i zbiór narzędzi na licencji GPL, udostępniający pełną funkcjonalność Uniksa na komputerach z systemem Windows. Zawiera także gotowego Pythona, który korzysta ze wszystkich udostępnionych narzędzi Uniksa.

Pythona można również znaleźć na płytach CD udostępnianych z dystrybucjami Linuksa jako część produktów czy systemów komputerowych, a także w niektórych książkach poświęconych temu językowi programowania. Zazwyczaj wersje z płyt są nieco w tyle za aktualnym wydaniem, choć nie aż tak bardzo.

Dodatkowo Pythona można znaleźć w niektórych pakietach programistycznych, zarówno darmowych, jak i komercyjnych. Przykładowo firma o nazwie ActiveState dystrybuuje Pythona jako część pakietu *ActivePython*. Pakiet ten łączy standardową implementację Pythona z rozszerzeniami przeznaczonymi dla programowania w systemie Windows, takimi jak PyWin32, zintegrowanym środowiskiem programistycznym o nazwie PythonWin (opisany w rozdziale 3.), a także innymi często wykorzystywany rozszerzeniami. Pythona można obecnie

także znaleźć w *Enthought Python Distribution* — pakietie przeznaczonym na potrzeby obliczeń naukowych — a także *Portable Python*, skonfigurowanym w taki sposób, by móc go uruchamiać bezpośrednio z urządzeń przenośnych. Więcej informacji na ten temat można znaleźć w Internecie.

Wreszcie jeśli interesują nas alternatywne implementacje Pythona, powinniśmy wyszukać w Internecie opisanych w rozdziale 2. implementacji *Jython* (Pythona przeznaczonego dla środowiska języka Java) oraz *IronPython* (dla świata .NET i języka C#). Instalacja tych dwóch systemów wykracza poza zakres niniejszej książki.

Instalacja Pythona

Po pobraniu Pythona będziemy musieli go zainstalować. Poszczególne etapy instalacji są specyficzne dla określonej platformy, poniżej znajduje się jednak kilka wskazówek przeznaczonych dla najważniejszych środowisk, w jakich Python jest wykorzystywany.

Windows

W systemie Windows Python pobierany jest jako samoinstalujący plik programu MSI. Wystarczy tylko dwukrotnie kliknąć jego ikonę i odpowiadać *Yes (Tak)* lub *Next (Dalej)* na każde pytanie, by wykonać domyślną instalację. Domyślna instalacja zawiera zbiór dokumentacji Pythona oraz obsługę graficznego interfejsu użytkownika *tkinter* (w Pythonie 2.6 *Tkinter*), baz danych modułu *shelve*, a także środowisko programistyczne IDLE. Python 3.0 i 2.6 są normalnie instalowane w katalogach *C:\Python30* oraz *C:\Python26*, choć można to zmienić w czasie instalacji.

Dla wygody po zainstalowaniu Python pojawia się w menu *Wszystkie programy* dostępnym pod przyciskiem *Start*. Menu Pythona zawiera pięć opcji umożliwiających szybki dostęp do często wykonywanych zadań — uruchomienia interfejsu użytkownika IDLE, odczytania dokumentacji modułu, uruchomienia sesji interaktywnej, wczytania dokumentacji biblioteki standardowej do przeglądarki oraz odinstalowania. Większość tych opcji obejmuje koncepcje omówione szczegółowo w niniejszej książce.

Po zainstalowaniu w systemie Windows domyślnie Python automatycznie rejestrze się jako program otwierający pliki Pythona po kliknięciu ich ikon (technika uruchamiania programów opisana w rozdziale 3.). Można również w systemie Windows zbudować Pythona z kodu źródłowego, choć nie jest to często stosowane.

Jedna uwaga dla użytkowników systemu Windows Vista: opcje zabezpieczeń niektórych wersji tego systemu zmieniają pewne reguły dotyczące używania plików instalacyjnych MSI. Choć w momencie czytania tych słów może to już przestać być problemem, więcej informacji na ten temat można znaleźć w ramce „Instalator MSI Pythona w systemie Windows Vista” w niniejszym dodatku. Opisano w niej, co można zrobić, jeśli aktualny instalator Pythona nie działa lub nie umieszcza Pythona w odpowiednim miejscu na naszym komputerze.

Linux

W systemie Linux Python dostępny jest jako jeden lub większa liczba plików RPM, które rozpakowuje się w normalny sposób (po szczegóły należy siegnąć do dokumentacji RPM). W zależności od wybranych plików RPM możemy otrzymać jeden z samym Pythonem i kolejne dodające obsługę graficznego interfejsu użytkownika *tkinter* oraz środowiska IDLE. Ponieważ Linux jest systemem podobnym do Uniksa, poniższy akapit ma również zastosowanie do tego systemu.

Unix

W systemach uniksowych Python jest zazwyczaj komplikowany z pełnej dystrybucji kodu źródłowego w języku C. Zwykle wymaga to jedynie rozpakowania pliku i wykonania prostych poleceń `config` oraz `make`. Python automatycznie konfiguruje swoją procedurę budowy zgadnie z systemem, na jakim jest komplikowany. Więcej informacji na temat tego procesu można znaleźć w pliku *README* pakietu. Ponieważ Python jest produktem open source, jego kod źródłowy może być wykorzystywany i dystrybuowany za darmo.

W przypadku innych platform szczegóły instalacji mogą różnić się między sobą, jednak zazwyczaj zgodne są z konwencjami dla danej platformy. Instalacja *Pippy*, wersji Pythona przeznaczonej dla systemu PalmOS, wymaga na przykład operacji synchronizacji typu HotSync. Python dla opartej na Linuksie serii PDA o nazwie Zaurus, produkowanych przez firmę Sharp, ma postać jednego lub większej liczby plików *.ipk*, które uruchamia się w celu zainstalowania. Ponieważ dodatkowe procedury instalacji dla form wykonywalnych oraz opartych na kodzie źródłowym są dobrze opisane, pominiemy tutaj szczegóły.

Konfiguracja Pythona

Po zainstalowaniu Pythona możemy chcieć skonfigurować pewne ustawienia systemowe wpływające na sposób wykonywania kodu przez Pythona. Jeśli dopiero zaczynamy swoją przygodę z tym językiem, możemy prawdopodobnie całkowicie pominąć ten podrozdział. W przypadku podstawowych programów nie ma najczęściej potrzeby wprowadzania zmian do ustawień systemowych.

Mówiąc ogólnie, część sposobu działania interpretera Pythona można skonfigurować za pomocą ustawień zmiennych środowiskowych oraz opcji wiersza poleceń. W niniejszym podrozdziale przyjrzymy się krótko obu opcjom, jednak należy sprawdzić inne źródła dokumentacji w celu uzyskania szczegółowych informacji na przedstawione tutaj tematy.

Zmienne środowiskowe Pythona

Zmienne środowiskowe, znane również jako zmienne powłoki lub zmienne DOS, są ustawieniami systemowymi, które znajdują się poza Pythonem i tym samym mogą być wykorzystywane do dostosowania interpretera do naszych wymagań za każdym razem, gdy jest on uruchamiany na danym komputerze. Python rozpoznaje pewną liczbę ustawień zmiennych środowiskowych, jednak tylko niektóre z nich są używane na tyle często, by należało je tu omówić. W tabeli A.1 przedstawiono najważniejsze ustawienia zmiennych środowiskowych związanych z Pythonem.

Tabela A.1. Istotne zmienne środowiskowe

Zmienna	Rola
PATH (lub path)	Systemowa ścieżka wyszukiwania powłoki (służąca do odnalezienia słowa python)
PYTHONPATH	Ścieżka wyszukiwania modułów Pythona (importowanie)
PYTHONSTARTUP	Ścieżka do interaktywnego pliku uruchomieniowego
TCL_LIBRARY, TK_LIBRARY	Zmienne rozszerzeń GUI (tkinter)

Instalator MSI Pythona w systemie Windows Vista

Kiedy piszę ten tekst, instalator Pythona dla systemu Windows jest plikiem instalacyjnym *.msi*. Format ten dobrze działał w systemie Windows XP (wystarczyło dwukrotnie kliknąć plik i gotowe), może on jednak sprawiać pewne problemy w niektórych wersjach systemu Windows Vista. W szczególności uruchamianie instalatora MSI poprzez klikanie może spowodować zainstalowanie Pythona w katalogu głównym dysku C: zamiast w poprawnym katalogu C:\PythonXX. Python zainstalowany w tym katalogu działa poprawnie, jednak nie jest to odpowiednie miejsce na jego umieszczenie.

Problem ten związany jest z bezpieczeństwem w systemie Windows Vista. W skrócie, pliki MSI nie są prawdziwymi plikami wykonywalnymi, dlatego nie dziedziczą one poprawnie uprawnień administratora, nawet jeśli wykonuje je użytkownik mający prawa administratora. Zamiast tego pliki MSI wykonywane są przez rejestr systemu — ich nazwy powiązane są z programem instalacyjnym MSI.

Problem ten wydaje się związany z określona wersją Pythona lub Visty. Na moim nowym laptopie Python 2.6 i 3.0 zainstalował się bez problemów. By jednak zainstalować Pythona 2.5.2 na komputerze przenośnym OQO z systemem Windows Vista, musiałem zastosować rozwiązanie oparte na wierszu poleceń, tak by wymusić wymagane uprawnienia administratora.

Jeśli Python nie zainstaluje się w poprawnym miejscu, oto recepta na obejście tego problemu. Należy z przycisku *Start* przejść do wpisu *Wszystkie programy*, wybrać *Akcesoria*, kliknąć prawym przyciskiem myszy opcję *Wiersz poleceń*, wybrać *Uruchom jako administrator* i w oknie dialogowym wybrać *Kontynuuj*. Teraz w oknie systemowego wiersza poleceń należy wpisać polecenie `cd` w celu przejścia do katalogu, w którym znajduje się nasz plik instalacyjny Pythona (na przykład `cd C:\user\downloads`), a następnie ręcznie uruchomić instalator MSI, wpisując polecenie w formie `msiexec /i python-2.5.2.msi`. Później należy postępować jak w normalnej interakcji z graficznym interfejsem użytkownika w celu ukończenia instalacji.

Takie zachowanie może się oczywiście z czasem zmienić. Procedura ta może nie być wymagana we wszystkich wersjach systemu Windows Vista; mogą się również pojawić inne metody obejścia tego problemu (takie jak wyłączenie zabezpieczeń Visty, jeśli ktoś ma odwagę to zrobić). Możliwe również, że instalator Pythona może być udostępniany w innym formacie, naprawiającym ten problem — na przykład jako prawdziwy plik wykonywalny. Należy pamiętać, by najpierw wypróbować działanie instalatora, po prostu klikając jego ikonę w celu sprawdzenia, czy działa poprawnie, zamiast odwoływać się do innych rozwiązań.

Powyższe zmienne środowiskowe są proste w użyciu, jednak warto umieścić tu kilka wskazówek.

PATH

Ustawienie PATH wymienia zbiór katalogów, które system operacyjny przeszukuje pod kątem programów wykonywalnych. Normalnie zmienna ta powinna zawierać katalog, w którym znajduje się nasz interpreter Pythona (program `python` w systemie Unix lub plik `python.exe` w systemie Windows).

Nie musimy ustawać tej zmiennej, jeśli zamierzamy pracować w katalogu, w którym Python się znajduje, lub wpisywać pełną ścieżkę do Pythona w wierszu poleceń. W systemie Windows na przykład zmienna PATH nie ma znaczenia, jeśli wykonamy polecenie `cd C:\Python30` przed wykonaniem jakiegokolwiek kodu (w celu zmiany na katalog, w którym znajduje się Python) lub zawsze zamiast samego `python` będziemy wpisywać `C:\Python30\python` (podając pełną ścieżkę). Warto również zauważyć, że ustawienia

zmiennej środowiskowej PATH służą przede wszystkim do uruchamiania programów z wiersza poleceń. Zazwyczaj nie mają znaczenia, kiedy programy uruchamia się za pomocą klikania ich ikon lub za pośrednictwem środowisk programistycznych.

PYTHONPATH

Ustawienie PYTHONPATH pełni rolę podobną do PATH. Interpreter Pythona konsultuje się ze zmienną PYTHONPATH w celu zlokalizowania plików modułów, kiedy się je importuje w programie. Jeśli korzystamy z tej zmiennej, zostaje ona ustawiona na listę nazw katalogów w postaci specyficznej dla określonej platformy, rozdzielonych w systemie Unix za pomocą dwukropka, a w systemie Windows — średnika. Lista ta normalnie zawiera tylko nasze własne katalogi z kodem źródłowym. Jej zawartość łączona jest ze ścieżką wyszukiwania importowanych modułów sys.path, wraz z katalogiem skryptu, ustawieniami z plików ścieżek i katalogami biblioteki standardowej.

Nie musimy ustawiać tej zmiennej, o ile nie wykonujemy operacji importowania pomiędzy różnymi katalogami. Ponieważ Python zawsze automatycznie przeszukuje katalog główny pliku najwyższego poziomu programu, to ustawienie jest niezbędne tylko wtedy, gdy moduł potrzebuje zimportować inny moduł znajdujący się w innym katalogu. Warto zobaczyć również omówienie plików ścieżek .pth w dalszej części dodatku, ponieważ są one alternatywą dla zmiennej PYTHONPATH. Więcej informacji na temat ścieżki wyszukiwania modułów znajduje się w rozdziale 21.

PYTHONSTARTUP

Jeśli zmienna PYTHONSTARTUP zostanie ustawiona na ścieżkę pliku z kodem Pythona, Python wykonuje ten kod pliku automatycznie za każdym razem, gdy uruchamiamy interpreter interaktywny, tak jakbyśmy wpisali wywołanie pliku w wierszu sesji interaktywnej. Jest to wykorzystywane rzadko, jednak przydaje się do upewnienia się, że zawsze ładujemy pewne narzędzia w pracy interaktywnej. Oszczędza nam to konieczności importowania.

Ustawienia tkinter

Jeśli chcemy skorzystać z zestawu narzędzi graficznego interfejsu użytkownika tkinter (w Pythonie 2.6 występującego pod nazwą Tkinter), być może konieczne będzie ustawienie dwóch zmiennych GUI z ostatniego wiersza tabeli A.1 na nazwy katalogów biblioteki źródeł systemów Tcl oraz Tk (podobnie jak robi się to w przypadku PYTHONPATH). Te ustawienia nie są jednak wymagane w systemie Windows (gdzie obsługa tkinter jest instalowana wraz z Pythonem) i zazwyczaj nie są konieczne nigdzie indziej, jeśli Tcl i Tk znajdują się w swoich standardowych katalogach.

Warto zauważyc, że ponieważ ustawienia zmiennych środowiskowych są zewnętrzne dla samego Pythona, moment skonfigurowania ich nie ma zazwyczaj znaczenia. Można to zrobić przed instalacją Pythona lub już po niej; wystarczy pamiętać, że ustawienia powinny mieć odpowiednie wartości przed samym *wykonaniem* Pythona.

Jak ustawić opcje konfiguracyjne?

Metoda ustawiania zmiennych środowiskowych związanych z Pythonem, a także to, na co je należy ustawić, zależy od typu komputera, na jakim pracujemy. I znów należy pamiętać, że nie musimy koniecznie ustawiać zmiennych środowiskowych od razu — w szczególności kiedy pracujemy w IDLE (aplikacji opisanej w rozdziale 3.), ich konfiguracja nie jest wymagana z góry.

Obsługa graficznego interfejsu użytkownika tkinter (oraz IDLE) w systemie Linux

Interfejs IDLE opisany w rozdziale 2. jest programem napisanym w Pythonie i opartym na graficznym interfejsie użytkownika tkinter. Moduł `tkinter` (w Pythonie 2.6 znany jako `Tkinter`) to zestaw narzędzi GUI, który jest kompletnym, standardowym komponentem Pythona w systemie Windows i kilku innych platformach. W niektórych systemach Linux biblioteka GUI będąca jego podstawą może nie być standardowym, zainstalowanym komponentem. By dodać obsługę graficznego interfejsu użytkownika do Pythona zainstalowanego na Linuksie, można spróbować wpisać w wierszu poleceń `yum tkinter`, by automatycznie zainstalować biblioteki będące podstawą `tkinter`. Powinno to działać we wszystkich dystrybucjach Linuksa (i niektórych innych systemach), w których dostępny jest program instalacyjny `yum`.

Załóżmy jednak, dla celów ilustracyjnych, że w katalogach `utilities` oraz `package1` gdzieś na komputerze mamy uniwersalnie przydatne pliki modułów, które chcemy móc importować do plików umieszczonych w innych katalogach. Żeby na przykład załadować plik `spam.py` z katalogu `utilities`, chcemy móc napisać po prostu:

```
import spam
```

w innym pliku znajdującym się w dowolnym miejscu komputera. By to zadziałało, musimy skonfigurować ścieżkę wyszukiwania modułów w taki sposób, by obejmowała ona katalog zawierający plik `spam.py`. Poniżej znajduje się kilka wskazówek dotyczących tego procesu.

Zmienne powłoki systemu Unix i Linux

W systemach Unix sposób ustawienia zmiennych środowiskowych uzależniony jest od wykorzystywanej powłoki. W przypadku powłoki `csh` możemy do pliku `.cshrc` lub `.login` dodać poniższy wiersz, by ustawić ścieżkę wyszukiwania modułów Pythona.

```
setenv PYTHONPATH /usr/home/pycode/utilities:/usr/lib/pycode/package1
```

Polecenie to każe Pythonowi szukać importowanych modułów w dwóch katalogach zdefiniowanych przez użytkownika. Alternatywnie, jeśli korzystamy z powłoki `ksh`, ustawienie to może zamiast tego pojawić się w pliku `.kshrc` i wyglądać tak, jak poniższy kod.

```
export PYTHONPATH="/usr/home/pycode/utilities:/usr/lib/pycode/package1"
```

Inne powłoki mogą wykorzystywać odmienną (choć analogiczną) składnię.

Zmienne DOS (system Windows)

Jeśli korzystamy z systemu MS-DOS lub starszych odmian systemu Windows, być może będzie namusili dodać polecenie konfigurujące zmienną środowiskową do pliku `C:\autoexec.bat` i uruchomić ponownie komputer, tak by zmiany zaczęły działać. Polecenie konfiguracyjne w takich komputerach wykorzystuje składnię unikalną dla systemu DOS.

```
set PYTHONPATH=c:\pycode\utilities;d:\pycode\package1
```

Takie polecenie można również wpisać w oknie konsoli DOS, jednak ustawienie takie będzie wtedy aktywne jedynie dla tego jednego okna konsoli. Modyfikacja pliku `.bat` sprawia, że zmiana będzie stała i globalna dla wszystkich programów.

Graficzny interfejs użytkownika zmiennych środowiskowych Windows

W nowszych wersjach systemu Windows, w tym XP i Vista, możemy zamiast tego ustawić PYTHONPATH oraz inne zmienne za pomocą graficznego interfejsu użytkownika systemowych zmiennych komputera. W systemie Windows XP należy wybrać *Panel sterowania*, później ikonę *System*, wybrać środowiskowych, bez konieczności edycji plików i ponownego uruchamiania zakładkę *Zaawansowane* i kliknąć przycisk *Zmienne środowiskowe* w celu wykonania edycji lub dodania nowych zmiennych (PYTHONPATH jest zazwyczaj zmienną użytkownika). Należy użyć tych samych nazw zmiennych i składni wartości co zaprezentowane wcześniej w poleceniu DOS set.

Nie trzeba uruchamiać ponownie komputera, jednak należy pamiętać o ponowym uruchomieniu Pythona, jeśli jest on otwarty, tak by pobrał on wprowadzone zmiany (ścieżka konfigurowana jest jedynie w czasie uruchamiania Pythona). Jeśli pracujemy w oknie *Wiersza poleceń* systemu Windows, będziemy najprawdopodobniej musieli ponownie uruchomić również ten program.

Rejestr systemu Windows

Zaawansowani użytkownicy tego systemu operacyjnego mogą również skonfigurować ścieżkę wyszukiwania modułów za pomocą *Edytora rejestru*. Należy wybrać z menu *Start* opcję *Uruchom...* i wpisać *regedit*. Zakładając, że na naszym komputerze znajduje się typowe narzędzie do obsługi rejestru, możemy teraz przejść do wpisów dotyczących Pythona i wprowadzić do nich odpowiednie zmiany. Jest to jednak procedura delikatna i podatna na błędy, dlatego osobom niezaznajomionym z rejestrem systemu Windows odradzam jej stosowanie. Właściwie przypomina to trochę operację na mózgu komputera, dlatego należy bardzo uważać.

Pliki ścieżek

Wreszcie jeśli zdecydujemy się rozszerzyć ścieżkę wyszukiwania modułów za pomocą pliku *.pth* zamiast zmiennej PYTHONPATH, możemy zamiast tego utworzyć plik tekstowy przypominający poniższy z systemu Windows (plik *C:\Python30\mypath.pth*).

```
c:\pycode\utilities  
d:\pycode\package1
```

Jego zawartość będzie różna dla różnych platform, a katalog zawierający może się zmieniać zarówno dla różnych systemów operacyjnych, jak i wersji Pythona. Python lokalizuje ten plik automatycznie w momencie uruchomienia.

Nazwy katalogów w plikach ścieżki mogą być bezwzględne lub określane względem katalogu zawierającego plik ścieżki. Można stosować większą liczbę plików *.pth* (dodane zostaną wszystkie ich katalogi), a same pliki *.pth* mogą się pojawiać w różnych automatycznie sprawdzanych katalogach specyficznych dla platformy i wersji Pythona. Generalnie dla wersji Pythona oznaczonej jako *N.M* pliki ścieżek szukane są zazwyczaj w katalogach *C:\PythonNM* oraz *C:\PythonNM\Lib\site-packages* w systemie Windows i w */usr/local/lib/pythonN.M/site-packages* oraz */usr/local/lib/site-python* w systemach Unix oraz Linux. Więcej informacji na temat wykorzystywania plików ścieżek w celu konfiguracji ścieżki wyszukiwania importowanych modułów sys.path można znaleźć w rozdziale 21.

Ponieważ ustawienia środowiskowe są często opcjonalne, a także dlatego, że książka ta nie jest poświęcona powłokom systemowym, po pozostałe szczegóły odsyłam do innych źródeł. Należy sięgnąć do dokumentacji powłoki systemowej, a jeśli mamy kłopot z odpowiednimi ustawieniami, warto poprosić o pomoc administratora systemu czy innego lokalnego eksperta.

Opcje wiersza poleceń Pythona

Kiedy uruchamiamy Pythona z systemowego wiersza poleceń, możemy przekazać różne opcje umożliwiające kontrolowanie tego, w jaki sposób działa Python. W przeciwieństwie do ustawionych dla całego systemu zmiennych środowiskowych opcje wiersza poleceń mogą być inne za każdym razem, gdy wykonujemy skrypt. Pełna forma wywołania Pythona z wiersza poleceń w wersji 3.0 wygląda następująco (w 2.6 jest mniej więcej tak samo, różni się jedynie kilka opcji):

```
python [-bBdEhiOsSuvVWx?] [-c polecenie | -m nazwa-modułu | skrypt | -] [argumenty]
```

Większość wierszy poleceń wykorzystuje jedynie części *skrypt* oraz *argumenty* z tego formatu w celu wykonania pliku źródłowego programu z argumentami, które mają być wykorzystane przez sam program. By to zilustrować, rozważmy poniższy skrypt *main.py*, wyświetlający listę argumentów wiersza poleceń udostępnioną skryptowi jako *sys.argv*:

```
# Plik main.py
import sys
print(sys.argv)
```

W poniższym wierszu poleceń *python* oraz *main.py* mogą również być pełnymi ścieżkami do katalogów, a trzy argumenty (*a* *b* *-c*) przeznaczone dla skryptu pokazują się na liście *sys.argv*. Pierwszym elementem *sys.argv* jest zawsze nazwa pliku skryptu, nawet jeśli jest ona znana.

```
c:\Python30> python main.py a b -c          # Najpopularniejsze rozwiązanie: wykonanie pliku skryptu
['main.py', 'a', 'b', '-c']
```

Inne opcje formatu kodu pozwalają nam określić kod Pythona do wykonania w samym wierszu poleceń (*-c*), przyjmować kod do wykonania ze standardowego strumienia wejścia (*a* — oznacza wczytanie z potoku lub przekierowanego pliku strumienia wejścia) i tak dalej:

```
c:\Python30> python -c "print(2 ** 100)" # Wczytanie kodu z argumentu polecenia
1267650600228229401496703205376
```

```
c:\Python30> python -c "import main"      # Zimportowanie pliku w celu wykonania jego kodu
['-c']
```

```
c:\Python30> python - < main.py a b -c    # Wczytanie kodu ze standardowego wejścia
[-', 'a', 'b', '-c']
```

```
c:\Python30> python - a b -c < main.py     # Ten sam efekt co poprzedni wiersz
[-', 'a', 'b', '-c']
```

Opcja *-m* lokalizuje moduł w ścieżce wyszukiwania modułów Pythona (*sys.path*) i wykonuje go jako skrypt najwyższego poziomu (jako moduł *__main__*). Opuszczamy tutaj rozszerzenie *.py*, ponieważ nazwa pliku jest modułem.

```
c:\Python30> python -m main a b -c        # Zlokalizowanie i wykonanie modułu jako skryptu
['c:\\Python30\\main.py', 'a', 'b', '-c']
```

Opcja *-m* obsługuje również wykonywanie modułów w pakietach za pomocą składni importowania względnego, w tym modułów znajdujących się w archiwach *.zip*. Jest często wykorzystywana

do uruchamiania modułów debugera pdb i narzędzia profilującego profile z wiersza poleceń w celu wywołania skryptu zamiast wykonywania interaktywnego, choć ten tryb użycia wydaje się nieco odmienny w wersji 3.0 (profile wydaje się dotknięty skutkami usunięcia execfile w wersji 3.0, natomiast pdb wchodzi w nadmiernie rozbudowany kod wejścia-wyjścia z nowego modułu io z Pythona 3.0):

```
c:\Python30> python -m pdb main.py a b -c          # Debugowanie skryptu
--Return--
> c:\python30\lib\io.py(762)closed()->False
-> return self.raw.closed()
(Pdb) c

c:\Python30> C:\python26\python -m pdb main.py a b -c  # W wersji 2.6 jest lepiej?
> c:\python30\main.py(1)<module>()
-> import sys
(Pdb) c

c:\Python30> python -m profile main.py a b -c        # Profilowanie skryptu

c:\Python30> python -m cProfile main.py a b -c        # Narzędzie profilujące o mniejszych wymaganiach
```

Natychmiast po słowie python i przed wyznaczeniem kodu, który ma być wykonany, Python przyjmuje dodatkowe argumenty kontrolujące jego własne działanie. Argumenty te są przez niego pobierane i nie są przeznaczone dla wykonywanego skryptu. Przykładowo -O uruchamia Pythona w trybie zoptymalizowanym, -u wymusza brak bufora dla standardowych strumieni, natomiast -i wchodzi do trybu interaktywnego po wykonaniu skryptu:

```
c:\Python30> python -u main.py a b -c              # Standardowe strumienie bez bufora
```

Python 2.6 obsługuje dodatkowe opcje włączające zgodność z wersją 3.0 (-3, -Q) oraz wykrywanie niespójnego użycia indentacji kodu, co w Pythonie 3.0 jest zawsze wykrywane i zgłasiane (-t, patrz rozdział 12.). Więcej informacji na temat dostępnych opcji wiersza poleceń można znaleźć w dokumentacji Pythona lub innych tekstach. A jeszcze lepiej będzie zapytać o to samego Pythona — wykonując polecenie takie jak poniższe:

```
c:\Python30> python -?
```

w celu uzyskania wyświetlania pomocy Pythona, dokumentującej dostępne opcje wiersza poleceń. Jeśli mamy do czynienia z bardziej skomplikowanymi wierszami poleceń, warto zajrzeć do modułów getopt oraz optparse biblioteki standardowej, które oferują bardziej wyszukane przetwarzanie wierszy poleceń.

Uzyskanie pomocy

Dzisiejszy zbiór dokumentacji Pythona zawiera wartościowe wskazówki dotyczące używania tego języka na różnych platformach. Dokumentacja biblioteki standardowej dostępna jest w systemie Windows pod menu *Start* od razu po zainstalowaniu Pythona (opcja *Python Manuals*), a także w Internecie pod adresem <http://www.python.org>. Więcej wskazówek dotyczących wykorzystywania Pythona na różnych platformach, a także aktualne informacje dotyczące środowiska i wierszy poleceń na poszczególnych platformach można znaleźć w dziale *Using Python* dokumentacji.

Jak zawsze pomocy można także szukać w Internecie, zwłaszcza w tej dziedzinie, która często zmienia się bardzo szybko — szybciej, niż można uaktualnić książki. Biorąc pod uwagę szerokie rozpowszechnienie Pythona, jest bardzo prawdopodobne, że odpowiedzi na wszystkie pytania dotyczące korzystania z tego języka znajdziemy za pomocą wyszukiwarki.

Rozwiązania ćwiczeń podsumowujących poszczególne części książki

Część I Wprowadzenie

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części pierwszej” na końcu rozdziału 3.

1. *Interakcja.* Zakładając, że Python jest poprawnie skonfigurowany, interakcja powinna wyglądać mniej więcej tak, jak poniżej — sesję interaktywną można uruchomić w dowolny sposób, na przykład w IDLE czy z powłoki systemowej:

```
% python
...wiersze z informacjami o prawach autorskich...
>>> "Witaj, sir Robinie!"
'Witaj, sir Robinie!'
>>>                                         # By wyjść z sesji interaktywnej, należy użyć Ctrl+D lub Ctrl+Z bądź
      zamknąć okno.
```

2. *Programy.* Plik z kodem (czyli moduł) o nazwie *module1.py* wraz z działaniami wykonywanymi w powłoce systemowej powinien wyglądać następująco:

```
# Plik module1.py
print('Witaj, module!')  
  
% python module1.py
Witaj, module!
```

Można uruchomić ten plik na inne sposoby — na przykład klikając jego ikonę czy korzystając z opcji *Run Module* z menu *Run* programu IDLE.

3. *Moduły.* Poniższy listing kodu z sesji interaktywnej ilustruje wykonywanie kodu modułu za pomocą importowania:

```
% python
>>> import module1
Witaj, module!
>>>
```

Należy pamiętać, że by wykonać moduł ponownie bez zatrzymywania i ponownego uruchamiania interpretera, należy moduł przeładować. Pytanie z przeniesieniem pliku do innego katalogu i ponownym go zimportowaniem jest podchwytliwe — jeśli Python

wygenerował plik *module1.pyc* w oryginalnym katalogu, wykorzysta go przy importie modułu, nawet jeśli plik z kodem źródłowym (*.py*) zostanie przeniesiony do katalogu niebędącego w ścieżce wyszukiwania Pythona. Plik *.pyc* jest zapisywany automatycznie, jeśli Python ma dostęp do katalogu pliku źródłowego, i zawiera skompilowany kod bajtowy modułu.Więcej informacji na temat modułów znajduje się w rozdziale 3.

4. *Skrypty*. Zakładając, że nasza platforma obsługuje sztuczkę z `#!`, rozwiążanie powinno wyglądać jak poniżej (choć nasz wiersz ze znakami `#!` może zawierać inną ścieżkę):

```
#!/usr/local/bin/python  
print('Witaj, module!')  
% chmod +x module1.py  
  
% module1.py  
Witaj, module!
```

5. *Błędy*. Poniższa sesja interaktywna (wykonana w Pythonie 3.0) demonstruje rodzaj komunikatów o błędzie, jakie możnatrzymać w trakcie wykonywania ćwiczenia. Tak naprawdę wywołujemy wyjątki Pythona. Domyślne zachowanie w momencie wystąpienia błędu kończy wykonywanie programu w Pythonie i wyświetla komunikat o błędzie wraz ze śladem stosu. Stos wywołań pokazuje, w którym miejscu programu byliśmy w momencie wystąpienia wyjątku (jeśli wywołania funkcji są aktywne, kiedy występuje błąd, część „Traceback” wyświetla wszystkie aktywne poziomy wywołań). W siódmej części książki pokażemy, że wyjątki można przechwytywać za pomocą instrukcji `try`, a później przetwarzać je w dowolny sposób. Jak się zresztą okaże, Python zawiera debugger kodu źródłowego, który świetnie sprawdza się przy specjalnych wymaganiach w zakresie wykrywania błędów. Jak na razie warto zauważyc, że Python w momencie wystąpienia błędów podaje znaczące komunikaty (a nie po cichu kończy działanie programu).

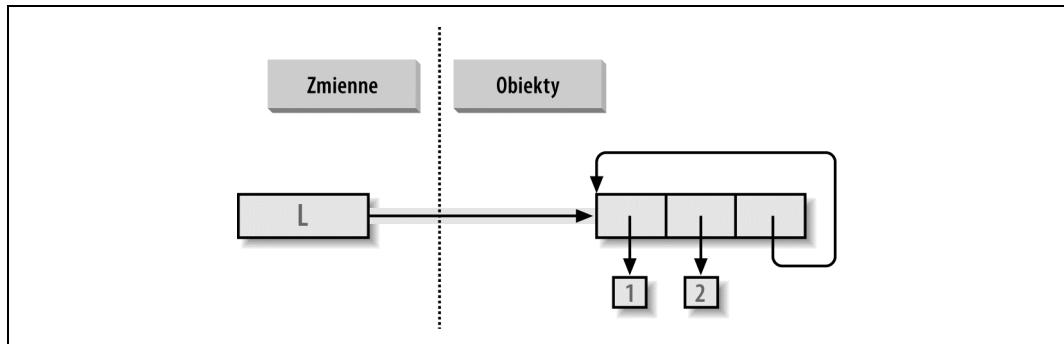
```
% python  
=>>> 2 ** 500  
3273390607896141870013189696827599152216642046043064789483291368096133796404674554  
=>88327009232590415715086684127560071009217256545885393053328527589376  
=>>>  
=>>> 1 / 0  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: int division or modulo by zero  
=>>>  
=>>> spam  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'spam' is not defined
```

6. *Przerwania i cykle*. Kiedy wpiszemy poniższy kod:

```
L = [1, 2]  
L.append(L)
```

tworzymy w Pythonie cykliczną strukturę danych. W wersjach Pythona starszych od 1.5.1 cykle w obiektach nie były wykrywane, przez co Python wyświetlał niekończący się strumień [1, 2, [1, 2, [1, 2, [1, 2, i tak dalej — aż do wywołania kombinacji klawiszy przerwywającej działanie programu (co z technicznego punktu widzenia powoduje wystąpienie wyjątku związanego z przerwaniem i wyświetlenie standardowego komunikatu). Od wersji 1.5.1 Python wykrywa cykle i wyświetla zamiast poprzedniego zapisu znaki [[...]], by nas poinformować o wykryciu pętli w strukturze obiektu i uniknięciu utknięcia związanego z nieskończonymi próbami wyświetlania.

Powód wystąpienia cyklu jest dość subtelny i wymaga informacji, które będziemy omawiać w drugiej części książki, dlatego na razie jedynie o tym wspomnijmy. Mówiąc w skrócie, przypisanie w Pythonie zawsze generuje *referencje* do obiektów, a nie ich kopie. Obiekty możemy sobie wyobrazić jako fragmenty pamięci, natomiast referencje — jako niejawne wskaźniki, którymi podążamy. Kiedy wykonujemy pierwsze przypisanie, zmienna `L` staje się nazwaną referencją do obiektu listy dwuelementowej — wskaźnikiem do miejsca w pamięci. Listy Pythona to tak naprawdę tablice referencji do obiektów z metodą `append`, która zmienia tablicę w miejscu, dołączając do niej (na końcu) inną referencję do obiektu. Tutaj metoda `append` dodaje po `L` referencję do `L`, co prowadzi do cyklu widocznego na rysunku B.1: na końcu listy znajduje się wskaźnik, który kieruje z powrotem do początku listy.



Rysunek B.1. Obiekt cykliczny utworzony poprzez dodanie listy do samej siebie. Domyslnie Python dodaje referencje do oryginalnej listy, a nie kopię tej listy

Poza specjalnym sposobem wyświetlania obiekty cykliczne muszą również być w specjalny sposób obsługiwane przez mechanizm czyszczenia pamięci Pythona (o czym dowiemy się w rozdziale 6.) — inaczej zajmowane przez nie miejsce nie zostanie zwolnione, kiedy przestaną one być w użyciu. Mimo że w praktyce jest to stosunkowo rzadkie, w niektórych programach przechodzących obiekty lub struktury wykrywanie cykli może być konieczne, by zapobiec zapętleniu. Choć trudno w to uwierzyć, cykliczne struktury danych mogą czasami być użyteczne, pomimo że wyświetlane są w specjalny sposób.

Część II Typy i operacje

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części drugiej” na końcu rozdziału 9.

1. *Podstawy.* Poniżej widać rezultaty, jakie można otrzymać, wraz z kilkoma komentarzami dotyczącymi ich znaczenia. Ponownie warto zwrócić uwagę, że w kilku przypadkach wykorzystano znak ; w celu zmieszczenia większej liczby instrukcji w jednym wierszu (znak ; jest separatorem instrukcji), natomiast przecinki tworzą krótki wyświetlane w nawiasach. Należy również pamiętać, że wynik operacji dzielenia na górze kodu będzie różny dla Pythona 2.6 oraz 3.0 (więcej informacji na ten temat można znaleźć w rozdziale 5.), natomiast opakowanie wywołań metod słownika w list jest niezbędne do wyświetlenia wyników w wersji 3.0, jednak w 2.6 już nie (więcej o tym w rozdziale 8.).

```

# Liczby

>>> 2 ** 16                                # 2 do potęgi 16
65536

>>> 2 / 5, 2 / 5.0                         # Liczba całkowita — odcięcie w Pythonie 2.6, jednak w wersji 3.0 już nie
(0.40000000000000002, 0.40000000000000002)

# Łańcuchy znaków

>>> "mielonka" + "jajka"                  # Konkatenacja
'mielonkajajka'

>>> S = "szynka"
>>> "jajka " + S
'jajka szynka'
>>> S * 5                                  # Powtórzenie
'szynkaszynkaszynkaszynkaszynka'
>>> S[:0]
''

>>> "zielone %s i %s" % ("jajka", S)      # Formatowanie
'zielone jajka i szynka'
>>> "zielone {0} i {1}".format('jajka', S)
'zielone jajka i szynka'

# Krotki

>>> ('x')[0]                               # Indeksowanie krotki jednoelementowej
'x'

>>> ('x', 'y')[1]                          # Indeksowanie krotki dwuelementowej
'y'

# Listy

>>> L = [1,2,3] + [4,5,6]                  # Operacje na listach
>>> L, L[:], L[:0], L[-2], L[-2:]
([1, 2, 3, 4, 5, 6], [1, 2, 3, 4, 5, 6], [], 5, [5, 6])
>>> [(1,2,3)+(4,5,6)][2:4]
[3, 4]
>>> [L[2], L[3]]                          # Pobranie elementów o wartości przesunięcia; przechowanie w liście
[3, 4]
>>> L.reverse(); L                        # Metoda: odwracanie listy w miejscu
[6, 5, 4, 3, 2, 1]
>>> L.sort(); L                          # Metoda: sortowanie listy w miejscu
[1, 2, 3, 4, 5, 6]
>>> L.index(4)                           # Metoda: wartość przesunięcia pierwszego elementu 4 (wyszukiwanie)
3

# Słowniki

>>> {'a':1, 'b':2}['b']                  # Indeksowanie słownika po kluczu
2

>>> D = {'x':1, 'y':2, 'z':3}
>>> D['w'] = 0                            # Utworzenie nowego wpisu
>>> D['x'] + D['w']
1
>>> D[(1,2,3)] = 4                      # Krotka użyta jako klucz (jest niezmienna)

>>> D
{'w': 0, 'z': 3, 'y': 2, (1, 2, 3): 4, 'x': 1}

>>> list(D.keys()), list(D.values()), (1,2,3) in D    # Metody, sprawdzenie kluczy
([ 'w', 'z', 'y', (1, 2, 3), 'x'], [0, 3, 2, 4, 1], True)

```

Puste obiekty

```
>>> [[], "",[],(),{},None]
([[], '', [], (), {}, None])
```

Dużo niczego: puste obiekty

2. *Indeksowanie i wycinki.* Indeksowanie poza granicami listy (na przykład `L[4]`) powoduje wystąpienie błędu. Python zawsze sprawdza, czy wszystkie wartości przesunięcia mieszczą się w granicach sekwencji.

Z drugiej strony, wycinek wykraczający poza granice listy (jak `L[-1000:100]`) działa, ponieważ Python skaluje wycinki tego rodzaju, tak by zawsze pasowały (w razie konieczności granice ustawiane są na zero i długość sekwencji).

Ekstrakcja sekwencji w odwrotnej kolejności, z niższą granicą większą od wyższej (na przykład `L[3:1]`), nie działa. Z powrotem otrzymamy pusty wycinek (`[]`), ponieważ Python skaluje granice wycinka w celu upewnienia się, że niższa granica jest zawsze mniejsza od lub równa wyższej granicy (`L[3:1]` zawsze skalowane jest do `L[3:3]`, pustego miejsca wstawiania w pozycji przesunięcia o wartości 3). Wycinki Pythona zawsze dokonywane są od lewej strony do prawej, nawet jeśli skorzystamy z ujemnych indeksów (najpierw są one przekształcane na indeksy dodatnie poprzez dodanie długości sekwencji). Warto zauważać, że zachowanie to modyfikują nieco wycinki z trzema argumentami wprowadzone w Pythonie 2.3 — na przykład `L[3:-1:-1]` powoduje ekstrakcję elementów od prawej strony do lewej:

```
>>> L = [1, 2, 3, 4]
>>> L[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> L[-1000:100]
[1, 2, 3, 4]
>>> L[3:1]
[]
>>> L
[1, 2, 3, 4]
>>> L[3:1] = ['?']
>>> L
[1, 2, 3, '?', 4]
```

3. *Indeksowanie, wycinki i del.* Nasza interakcja z interpreterem Pythona powinna wyglądać podobnie do poniższego listingu. Warto zauważać, że przypisanie pustej listy do wartości przesunięcia powoduje umieszczenie tam pustej listy, natomiast przypisanie pustej listy do wycinka usuwa ten wycinek. Przypisanie do wycinka oczekuje innej sekwencji — w przeciwnym razie otrzymuje się błąd typu. Wstawia ono elementy do środka przypisanej sekwencji, a nie do samej sekwencji.

```
>>> L = [1,2,3,4]
>>> L[2] = []
>>> L
[1, 2, [], 4]
>>> L[2:3] = []
>>> L
[1, 2, 4]
>>> del L[0]
>>> L
[2, 4]
>>> del L[1:]
>>> L
[2]
```

```
>>> L[1:2] = 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

4. *Przypisywanie krotek.* Wartości X oraz Y zostają zamienione. Kiedy krotki pojawiają się po lewej i prawej stronie symbolu przypisania (=), Python przypisuje obiekty znajdujące się po prawej stronie do obiektów docelowych z lewej strony, zgodnie z ich pozycjami. Najłatwiej będzie to chyba zrozumieć, uwzględniając fakt, że obiekt docelowy z lewej strony nie jest prawdziwą krotką, nawet jeśli tak wygląda — tak naprawdę to zbiór niezależnych celów przypisania. Elementy z prawej strony są krotką, która przy przypisaniu zostaje rozpakowana (krotka udostępnia tymczasowe przypisanie potrzebne do uzyskania efektu zamiany).

```
>>> X = 'mielonka'
>>> Y = 'jajka'
>>> X, Y = Y, X
>>> X
'jajka'
>>> Y
'mielonka'
```

5. *Klucze słowników.* Kluczem słownika może być dowolny obiekt niezmienny — w tym liczby całkowite, krotki czy łańcuchy znaków. To naprawdę jest słownik, choć niektóre z jego kluczy wyglądają jak wartości przesunięcia będące liczbami całkowitymi. Zadziałają również klucze o mieszanych typach.

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Indeksowanie słowników.* Próba zindeksowania nieistniejącego klucza (D['d']) kończy się błędem. Przypisanie do nieistniejącego klucza (D['d']='mielonka') tworzy nowy wpis do słownika. Indeksowanie listy za pomocą nieistniejącej wartości przesunięcia również zwraca błąd, podobnie jak próba przypisania do takiej wartości. Nazwy zmiennych działają jak klucze słowników — przy referencji muszą już być przypisane i są tworzone przy pierwszym przypisaniu. Nazwy zmiennych można tak naprawdę przetwarzać jako klucze słowników, jeśli mamy na to ochotę (są one widoczne w przestrzeni nazw modułu lub słownikach ramki stosu).

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D['a']
1
>>> D['d']
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyError: d
>>> D['d'] = 4
>>> D
{'b': 2, 'd': 4, 'a': 1, 'c': 3}
>>>
>>> L = [0, 1]
>>> L[2]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

```
>>> L[2] = 3
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

7. Operacje uniwersalne. Odpowiedzi na pytania:

- Operator + nie działa na różnych (mieszanych) typach obiektów (na przykład łańcuchu znaków i liście czy liście i krotce).
- Operator + nie działa dla słowników, ponieważ nie są one sekwencjami.
- Metoda append działa tylko dla list, a nie łańcuchów znaków; metoda keys działa tylko dla słowników. Metoda append zakłada, że jej obiekt docelowy jest zmienny, ponieważ jest rozszerzeniem w miejscu — łańcuchy znaków są niezmienne.
- Wycinki i konkatenacja zawsze zwracają nowy obiekt tego samego typu co obiekt przetwarzany.

```
>>> "x" + 1
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
>>>
>>> {} + {}
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>>
>>> [].append(9)
>>> "".append('s')
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'str' object has no attribute 'append'
>>>
>>> list({}.keys())                                # Wywołanie list() niezbędne jest w Pythonie 3.0, w 2.6 nie
[]
>>> [].keys( )
Traceback (innermost last):
  File "<stdin>", line 1, in ?
AttributeError: 'list' object has no attribute 'keys'
>>>
>>> [][:]
[]
>>> ""[:]
''
```

8. Indeksowanie łańcuchów znaków. To nieco podchwytliwe pytanie — ponieważ łańcuchy znaków są kolekcjami łańcuchów jednoznakowych, przy ich indeksowaniu otrzymujemy z powrotem łańcuch znaków, który znowu można zindeksować. S[0][0][0][0][0] po prostu ciągle indeksuje pierwszy znak. Nie działa to dla list (listy mogą przechowywać dowolne obiekty), o ile nie zawierają one łańcuchów znaków.

```
>>> S = "mielonka"
>>> S[0][0][0][0][0]
'm'
>>> L = ['m', 'i']
>>> L[0][0][0]
'm'
```

9. Typy niezmienne. Zadziała dowolne z poniższych rozwiązań. Nie zadziała przypisanie do indeksu, ponieważ łańcuchy znaków są niezmienne.

```

>>> S = "jajo"
>>> S = S[0:3] + 'a'
>>> S
'jaja'
>>> S = S[0] + S[1] + S[2] + 'a'
>>> S
'jaja'

```

(Polecam również zajrzeć do omówienia typu łańcuchów znaków bytearray z Pythona 3.0 w rozdziale 36. — jest on zmienną sekwencją małych liczb całkowitych, przetwarzaną właściwie w ten sam sposób co zwykłe łańcuchy znaków).

10. Zagnieździanie.

Poniżej przykład:

```

>>> me = {'name':('Robert', 'F', 'Zielony'), 'age':'?', 'job':'inżynier'}
>>> me['job']
'inżynier'
>>> me['name'][2]
'Zielony'

```

11. Pliki.

Poniżej zaprezentowano sposób na utworzenie i wczytanie pliku tekstowego w Pythonie (ls to polecenie z Uniksa; w systemie Windows należy skorzystać z dir).

```

# Plik maker.py
file = open('myfile.txt', 'w')
file.write('Witaj, wspaniały świecie!\n')      # Lub: open( ).write( )
file.close()                                     # close nie zawsze jest potrzebne

# Plik reader.py
file = open('myfile.txt')                         # 'r' to domyślny tryb otwierania pliku
print(file.read())                               # Lub: print(open( ).read( ))

% python maker.py
% python reader.py
Witaj, wspaniały świecie!

% ls -l myfile.txt
-rwxrwxrwa    1 0          0                  19 Apr 13 16:33 myfile.txt

```

Część III Instrukcja i składnia

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części trzeciej” na końcu rozdziału 15.

1. *Zapisywanie w kodzie podstawowych pętli.* W miarę wykonywania tego ćwiczenia powinniśmy otrzymać kod wyglądający w następujący sposób:

```

>>> S = 'mielonka'
>>> for c in S:
...     print(ord(c))
...
109
105
101
108
111
110
107
97

>>> x = 0

```

```

>>> for c in S: x += ord(c)           # Lub: x = x + ord(c)
...
>>> x
848

>>> x = []
>>> for c in S: x.append(ord(c))
...
>>> x
[109, 105, 101, 108, 111, 110, 107, 97]

>>> list(map(ord, S))             # Wywołanie list() wymagane jest w Pythonie 3.0, jednak nie w 2.6
[109, 105, 101, 108, 111, 110, 107, 97]

```

2. *Znaki ukośników lewych.* Przykład wyświetla pięćdziesiąt razy znak sygnału dźwiękowego (\a). Zakładając, że nasz komputer potrafi sobie z tym poradzić, i kiedy wykonujemy kod poza IDLE, możemy otrzymać serię pisków (lub jeden długi dźwięk, jeśli nasz komputer jest wystarczająco szybki). Ostrzegałem!
3. *Sortowanie słowników.* Poniżej znajduje się jeden ze sposobów wykonania ćwiczenia (jeśli wydaje się on nie mieć sensu, należy spojrzeć do rozdziału 8. lub rozdziału 14.). Należy pamiętać, że naprawdę musimy rozbić wywołania metod keys i sort w poniższy sposób, gdyż metoda sort zwraca None. W Pythonie 2.2 i późniejszych wersjach można wykonać iterację bezpośrednio po kluczach słownika bez wywoływania metody keys (za pomocą `for key in D:`), jednak lista kluczów nie zostanie posortowana tak, jak w kodzie niżej. W nowszych wersjach Pythona ten sam efekt można uzyskać za pomocą wbudowanej metody sorted.

```

>>> D = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7}
>>> D
{'f': 6, 'c': 3, 'a': 1, 'g': 7, 'e': 5, 'd': 4, 'b': 2}
>>>
>>> keys = list(D.keys())           # Wywołanie list() wymagane jest w Pythonie 3.0, jednak nie w 2.6
>>> keys.sort( )
>>> for key in keys:
...     print(key, '=>', D[key])
...
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
g => 7

>>> for key in sorted(D):          # Lepsze rozwiązanie, w nowszych wersjach Pythona
...     print(key, '=>', D[key])

```

4. *Alternatywne logiki programu.* Poniżej znajdują się różne wersje kodu będące rozwiązaniami. W przypadku kroku e należy przypisać wynik działania `2 ** X` do zmiennej poza pętlami z kroków a oraz b i użyć tej zmiennej wewnętrz pętli. Każda osoba może mieć nieco inne wyniki. To ćwiczenie zostało zaprojektowane przede wszystkim jako umożliwiające zabawę z alternatywnymi wersjami kodu, dlatego punkty można otrzymywać za każde rozsądne rozwiązanie.

```

# a

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

```

```

i = 0
while i < len(L):
    if 2 ** X == L[i]:
        print('pod indeksem', i)
        break
    i = i+1
else:
    print(X, 'nie odnaleziono')

# b

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

for p in L:
    if (2 ** X) == p:
        print((2 ** X), 'odnaleziono pod indeksem', L.index(p))
        break
    else:
        print(X, 'nie odnaleziono')

# c

L = [1, 2, 4, 8, 16, 32, 64]
X = 5

if (2 ** X) in L:
    print((2 ** X), 'odnaleziono pod indeksem', L.index(2 ** X))
else:
    print(X, 'nie odnaleziono')

# d

X = 5
L = []
for i in range(7): L.append(2 ** i)
print(L)

if (2 ** X) in L:
    print((2 ** X), 'odnaleziono pod indeksem', L.index(2 ** X))
else:
    print(X, 'nie odnaleziono')

# e

X = 5
L = list(map(lambda x: 2**x, range(7)))      # Lub [2**x for x in range(7)]
print(L)                                      # list() w celu wyświetlenia wszystkiego w Pythonie 3.0, w 2.6 nie

if (2 ** X) in L:
    print((2 ** X), 'odnaleziono pod indeksem', L.index(2 ** X))
else:
    print(X, 'nie odnaleziono')

```

Część IV Funkcje

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części czwartej” na końcu rozdziału 20.

1. *Podstawy*. W tym ćwiczeniu nie ma nic specjalnego, jednak warto zauważyc, że użycie instrukcji `print` (i tym samym naszej funkcji) jest z technicznego punktu widzenia operacją *polimorficzną*, która wykonuje odpowiednie działanie dla każdego typu obiektu.

```
% python
>>> def func(x): print(x)
...
>>> func("mielonka")
mielonka
>>> func(42)
42
>>> func([1, 2, 3])
[1, 2, 3]
>>> func({'jedzenie': 'mielonka'})
{'jedzenie': 'mielonka'}
```

2. *Argumenty*. Poniżej znajduje się przykładowe rozwiązanie. Należy pamiętać, że by zobaczyć wyniki z wywołań testowych, musimy użyć instrukcji `print`, ponieważ plik nie jest tym samym co kod wpisany w sesji interaktywnej. Python nie zwraca normalnie wyników instrukcji wyrażeń w plikach.

```
def adder(x, y):
    return x + y

print(adder(2, 3))
print(adder('mielonka', 'jajka'))
print(adder(['a', 'b'], ['c', 'd']))

% python mod.py
5
mielonkajajka
['a', 'b', 'c', 'd']
```

3. *Zmienna liczba argumentów*. Dwie alternatywne wersje funkcji `adder` zaprezentowane zostały w pliku `adders.py`. Najtrudniejszym zadaniem jest tu znalezienie sposobu na zainicjalizowanie akumulatora na pustą wartość dowolnego przekazywanego typu. Pierwsze rozwiązanie wykorzystuje ręczne sprawdzanie typów w celu odszukania liczby całkowitej oraz pusty wycinek pierwszego argumentu (który, zgodnie z założeniem, ma być sekwencją), jeśli argument okazuje się nie być liczbą całkowitą. Drugie rozwiązanie wykorzystuje pierwszy argument do zainicjalizowania i przejrzenia elementów od drugiego w góre, podobnie do wariantów funkcji `min` zaprezentowanych w rozdziale 18.

Drugie rozwiązanie jest lepsze. Oba zakładają, że wszystkie argumenty są tego samego typu, i żadne nie działa na słownikach (jak widzieliśmy w drugiej części książki, operator `+` nie działa na typach mieszanych oraz słownikach). Moglibyśmy również dodać sprawdzanie typów i specjalny kod obsługujący słowniki, jednak jest to zadanie dodatkowe.

```
def adder1(*args):
    print('adder1', end=' ')
    if type(args[0]) == type(0):          # Liczba całkowita?
        sum = 0                          # Inicjalizacja do wartości zero
    else:                                # Inaczaj sekwencja
        sum = args[0][:0]                # Wykorzystanie pustego wycinka arg1
    for arg in args:
        sum = sum + arg
    return sum

def adder2(*args):
    print('adder2', end=' ')
    sum = args[0]                      # Inicjalizacja do arg1
```

```

for next in args[1:]:
    sum += next
    # Dodanie elementów 2..N
return sum

for func in (adder1, adder2):
    print(func(2, 3, 4))
    print(func('mielonka', 'jajka', 'tost'))
    print(func(['a', 'b'], ['c', 'd'], ['e', 'f']))

% python adders.py
adder1 9
adder1 mielonkajajkatost
adder1 ['a', 'b', 'c', 'd', 'e', 'f']
adder2 9
adder2 mielonkajajkatost
adder2 ['a', 'b', 'c', 'd', 'e', 'f']

```

4. *Słowa kluczowe.* Poniżej znajduje się moje rozwiązanie pierwszej i drugiej części ćwiczenia (plik *mod.py*). By wykonać iterację po argumentach będących słowami kluczowymi, należy w nagłówku funkcji użyć formy `**args` i wykorzystać pętlę (na przykład `for x in args.keys(): use args[x]`) lub użyć `args.values()`, by przypominało to sumowanie pozycyjnych argumentów `*args`.

```

def adder(good=1, bad=2, ugly=3):
    return good + bad + ugly

print(adder())
print(adder(5))
print(adder(5, 6))
print(adder(5, 6, 7))
print(adder(ugly=7, good=6, bad=5))

% python mod.py
6
10
14
18
18

# Rozwiązanie części drugiej

def adder1(*args):           # Suma dowolnej liczby argumentów pozycyjnych
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder2(**args):          # Suma dowolnej liczby argumentów ze słowami kluczowymi
    argskeys = list(args.keys()) # W Pythonie 3.0 dodano wywołanie list()!
    tot = args[argskeys[0]]
    for key in argskeys[1:]:
        tot += args[key]
    return tot

def adder3(**args):          # To samo, ale przekształcenie na listę wartości
    args = list(args.values()) # W Pythonie 3.0 do indeksowania dodano wywołanie list()
    tot = args[0]
    for arg in args[1:]:
        tot += arg
    return tot

def adder4(**args):          # To samo, ale z ponownym użyciem wersji z argumentami pozycyjnymi
    return adder1(*args.values())

```

```

print(adder1(1, 2, 3), adder1('aa', 'bb', 'cc'))
print(adder2(a=1, b=2, c=3), adder2(a='aa', b='bb', c='cc'))
print(adder3(a=1, b=2, c=3), adder3(a='aa', b='bb', c='cc'))
print(adder4(a=1, b=2, c=3), adder4(a='aa', b='bb', c='cc'))

```

5. (oraz 6.). Poniżej znajdują się moje rozwiązania ćwiczenia 5. oraz 6. (plik *dicts.py*). Są to jedynie ćwiczenia programistyczne, gdyż w Pythonie 1.5 dodano metody słowników `D.copy()` oraz `D1.update(D2)`, które obsługują działania takie, jak kopiowanie oraz dodawanie (łączenie) słowników. Więcej informacji na ten temat można znaleźć w dokumentacji biblioteki standardowej Pythona lub w książce *Python. Leksykon kieszonkowy. Wydanie IV* wydawnictwa Helion. `X[:]` nie działa na słownikach, gdyż nie są one sekwencjami (więcej informacji w rozdziale 8.). Należy również pamiętać, że jeśli wykonamy przypisanie (`e = d`) zamiast kopiowania, wygenerujemy referencję do współdzielonego obiektu słownika, a modyfikacja d pochodzi za sobą zmianę e.

```

def copyDict(old):
    new = {}
    for key in old.keys():
        new[key] = old[key]
    return new

def addDict(d1, d2):
    new = {}
    for key in d1.keys():
        new[key] = d1[key]
    for key in d2.keys():
        new[key] = d2[key]
    return new

% python
>>> from dicts import *
>>> d = {1: 1, 2: 2}
>>> e = copyDict(d)
>>> d[2] = '?'
>>> d
{1: 1, 2: '?'}
>>> e
{1: 1, 2: 2}

>>> x = {1: 1}
>>> y = {2: 2}
>>> z = addDict(x, y)
>>> z
{1: 1, 2: 2}

```

6. Patrz 5.

1. Więcej przykładów dopasowywania argumentów. Poniżej widoczny jest rodzaj interakcji, jaki powinniśmy otrzymać, wraz z komentarzami objaśniającymi odbywające się dopasowania.

```

def f1(a, b): print(a, b)          # Normalne argumenty

def f2(a, *b): print(a, b)          # Zmienna liczba argumentów pozycyjnych

def f3(a, **b): print(a, b)          # Zmienna liczba słów kluczowych

def f4(a, *b, **c): print(a, b, c)  # Tryby mieszane

def f5(a, b=2, c=3): print(a, b, c) # Wartości domyślne

def f6(a, b=2, *c): print(a, b, c)  # Zmienna liczba argumentów pozycyjnych i wartości domyślnych

% python

```

```

>>> f1(1, 2)                                # Dopasowanie po pozycji (kolejność ma znaczenie)
1 2
>>> f1(b=2, a=1)                            # Dopasowanie po nazwie (kolejność nie ma znaczenia)
1 2

>>> f2(1, 2, 3)                            # Dodatkowe argumenty pozycyjne zebrane w krotkę
1 (2, 3)

>>> f3(1, x=2, y=3)                        # Dodatkowe słowa kluczowe zebrane w słownik
1 {'x': 2, 'y': 3}

>>> f4(1, 2, 3, x=2, y=3)                  # Dodatkowe argumenty obu typów
1 (2, 3) {'x': 2, 'y': 3}

>>> f5(1)                                  # Obie wartości domyślne użyte
1 2 3
>>> f5(1, 4)                              # Jedna wartość domyślna użyta
1 4 3

>>> f6(1)                                  # Jeden argument: dopasowanie "a"
1 2 ()
>>> f6(1, 3, 4)                            # Dodatkowe argumenty pozycyjne zebrane w krotkę
1 3 (4, )

```

8. Powrót do liczb pierwszych. Poniżej znajduje się przykład z liczbami pierwszymi opakowany w funkcję oraz moduł (plik *primes.py*), tak by można go było wykonać kilka razy. Dodalem test `if` przechwytyujący liczby ujemne, 0 oraz 1. Zmieniłem również operator `/` na `//`, by rozwiązań to było odporne na zmiany związane z „prawdziwym dzieleniem” w Pythonie 3.0 (omówione w rozdziale 5.), a także by mogło ono obsługiwać liczby zmiennoprzecinkowe (aby zobaczyć różnice z wersji 2.6, należy usunąć komentarz z instrukcji `from __future__ import division` i zmienić operator `//` na `/`).

```

#from __future__ import division

def prime(y):
    if y <= 1:                                     # Dla jakiegoś y > 1
        print(y, 'nie jest liczbą pierwszą')
    else:
        x = y // 2                                 # 3.0 / nie działa
        while x > 1:
            if y % x == 0:                         # Bez reszty?
                print(y, 'ma czynnik', x)
                break
            x -= 1
        else:
            print(y, 'jest liczbą pierwszą')

prime(13); prime(13.0)
prime(15); prime(15.0)
prime(3); prime(2)
prime(1); prime(-3)

```

Poniżej widać działanie tego modułu. Operator `//` pozwala, by funkcja ta działała również dla liczb zmiennoprzecinkowych, nawet jeśli pewnie nie powinno tak być.

```

% python primes.py
13 jest liczbą pierwszą
13.0 jest liczbą pierwszą
15 ma czynnik 5
15.0 ma czynnik 5.0
3 jest liczbą pierwszą
2 jest liczbą pierwszą
1 nie jest liczbą pierwszą
-3 nie jest liczbą pierwszą

```

Funkcja nadal jeszcze nieszczególnie nadaje się do ponownego użycia — mogłaby zwracać wartości, zamiast je wyświetlać — jednak na potrzeby naszych eksperymentów wystarczy. Nie oblicza też liczb pierwotnych w ścisłym, matematycznym znaczeniu tego pojęcia (działa na liczbach zmienoprzecinkowych) i nadal jest niewydajna. Poprawki pozostawiam jako ćwiczenia dla osób o większych zdolnościach matematycznych (wskaźówka: pętla `for` po `range(y, 1, -1)` może być nieco szybsza od instrukcji `while`, jednak algorytm jest tutaj prawdziwym wąskim gardłem). By zmierzyć alternatywne rozwiązania, należy użyć wbudowanego modułu `time` i wzorców programistycznych przypominających te stosowane w poniższym ogólnym czasomierzu wywołania funkcji (po szczegółowe informacje należy siegnąć do dokumentacji biblioteki standardowej).

```
def timer(reps, func, *args):
    import time
    start = time.clock()
    for i in range(reps):
        func(*args)
    return time.clock() - start
```

9. *Listy składane.* Poniżej znajduje się przykład kodu, jaki powinno się utworzyć. Być może mam swoje preferowane rozwiązania, ale go nie zdradzę.

```
>>> values = [2, 4, 9, 16, 25]
>>> import math

>>> res = []
>>> for x in values: res.append(math.sqrt(x))
...
>>> res
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> list(map(math.sqrt, values))
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]

>>> [math.sqrt(x) for x in values]
[1.4142135623730951, 2.0, 3.0, 4.0, 5.0]
```

10. *Narzędzia do pomiaru czasu.* Oto kod, który napisałem w celu pomiaru czasu trwania trzech opcji pierwiastkowania, wraz z wynikami w Pythonie 2.6 oraz 3.0. Ostatni wynik każdej z funkcji jest wyświetlany w celu zweryfikowania tego, czy wszystkie trzy wykonują to samo działanie.

```
# Plik mytimer.py (Python 2.6 oraz 3.0)

...to samo co w rozdziale 20...

# Plik timesqrt.py
import sys, mytimer
reps = 10000
repslist = range(reps)                                # Pobranie pełnego przedziału listy pomiarów dla Pythona 2.6

from math import sqrt
def mathMod():
    for i in repslist:
        res = sqrt(i)
    return res

def powCall():
    for i in repslist:
        res = pow(i, .5)
    return res

def powExpr():
```

```

for i in repslist:
    res = i ** .5
return res

print(sys.version)
for tester in (mytimer.timer, mytimer.best):
    print('<%s>' % tester.__name__)
    for test in (mathMod, powCall, powExpr):
        elapsed, result = tester(test)
        print ('-'*35)
        print ('%s: %.5f => %s' %(test.__name__, elapsed, result))

```

Poniżej znajdują się wyniki testów dla Pythona 2.6 oraz 3.0. W obu przypadkach wygląda, że moduł math jest szybszy od wyrażenia z operatorem `**`, które z kolei jest szybsze od wywołania funkcji `pow`. Należy jednak spróbować samodzielnie, na własnym komputerze i zainstalowanej na nim wersji Pythona. Warto również zwrócić uwagę na to, że Python 3.0 jest w tym teście prawie dwa razy wolniejszy od wersji 2.6. Python 3.1 i nowsze wersje mogą działać lepiej (warto zmierzyć to w przyszłości samodzielnie, by się o tym przekonać).

```

c:\misc> c:\python30\python timesqrt.py
3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)]
<timer>
-----
mathMod: 5.33906 => 99.994999875
-----
powCall: 7.29689 => 99.994999875
-----
powExpr: 5.95770 => 99.994999875
<best>
-----
mathMod: 0.00497 => 99.994999875
-----
powCall: 0.00671 => 99.994999875
-----
powExpr: 0.00540 => 99.994999875

c:\misc> c:\python26\python timesqrt.py
2.6.1 (r261:67517, Dec 4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)]
<timer>
-----
mathMod: 2.61226 => 99.994999875
-----
powCall: 4.33705 => 99.994999875
-----
powExpr: 3.12502 => 99.994999875
<best>
-----
mathMod: 0.00236 => 99.994999875
-----
powCall: 0.00402 => 99.994999875
-----
powExpr: 0.00287 => 99.994999875

```

W celu zmierzenia względnych szybkości *słowników składanych* Pythona 3.0 oraz ich odpowiedników w postaci pętli `for` w sesji interaktywnej należy wykonać sesję podobną do poniższej. Wydaje się, że oba rozwiązania są w Pythonie 3.0 mniej więcej porównywalne. W przeciwieństwie do list składanych pętle wykonywane ręcznie są dziś nieco szybsze od słowników składanych (choć różnica nie jest znacząca — możemy zaoszczędzić pół sekundy, tworząc pięćdziesiąt słowników, każdy z milionem elementów). I znów, zamiast brać wyniki tego porównania za prawdę objawioną, warto sprawdzić to na własnym komputerze i zainstalowanej na nim wersji Pythona.

```

c:\misc> c:\python30\python
>>>
>>> def dictcomp(I):
...     return {i: i for i in range(I)}
...
>>> def dictloop(I):
...     new = {}
...     for i in range(I): new[i] = i
...     return new
...
>>> dictcomp(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>> dictloop(10)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6, 7: 7, 8: 8, 9: 9}
>>>
>>> from mytimer import best, timer
>>> best(dictcomp, 10000)[0]                                # Słownik o 10 000 elementów
0.0013519874732672577
>>> best(dictloop, 10000)[0]
0.001132965223233029
>>>
>>> best(dictcomp, 100000)[0]                               # 100 000 elementów: 10 razy wolniejszy
0.01816089754424155
>>> best(dictloop, 100000)[0]
0.01643484018219965
>>>
>>> best(dictcomp, 1000000)[0]                             # 1 000 000 elementów: 10 razy tyle czasu
0.18685105229855026
>>> best(dictloop, 1000000)[0]                            # Czas tworzenia jednego słownika
0.1769041177020938
>>>
>>> timer(dictcomp, 1000000, _reps=50)[0]                # Słownik z 1 000 000 elementów
10.692516087938543
>>> timer(dictloop, 1000000, _reps=50)[0]                # Czas tworzenia 50
10.197276050447755

```

Część V Moduły

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części piątej” na końcu rozdziału 24.

1. Podstawy importowania. Kiedy skończymy, plik *mymod.py* oraz sesja interaktywna powinny wyglądać podobnie do poniższego kodu. Należy pamiętać, że Python może wczytać cały plik do listy łańcuchów znaków. Wbudowana funkcja `len` zwraca natomiast długość łańcuchów znaków oraz list.

```

def countLines(name):
    file = open(name)
    return len(file.readlines( ))

def countChars(name):
    return len(open(name).read( ))

def test(name):                                         # Lub przekazanie obiektu pliku
    return countLines(name), countChars(name)           # Lub zwrócenie słownika

% python
>>> import mymod
>>> mymod.test('mymod.py')
(10, 306)

```

Warto zwrócić uwagę na to, że funkcje ładują cały plik w pamięci na raz, dlatego nie będą działały dla patologicznie dużych plików przekraczających możliwości pamięci komputera. Do szerszego zastosowania możemy również wczytać plik wiersz po wierszu za pomocą iteratorów i zliczać je w miarę wczytywania.

```
def countLines(name):
    tot = 0
    for line in open(name): tot += 1
    return tot

def countChars(name):
    tot = 0
    for line in open(name): tot += len(line)
    return tot
```

Wyrażenie generatora może dawać ten sam rezultat — `sum(len(line) for line in open(name))`. W systemie Unix możemy zweryfikować dane wyjściowe za pomocą polecenia `wc`. W systemie Windows można kliknąć plik prawym przyciskiem myszy i sprawdzić jego właściwości. Warto jednak zauważać, że nasz skrypt może zgłaszać mniej znaków od systemu Windows — ze względu na przenośność Python przekształca znaczniki końca wiersza `\r\n` do `\n`, usuwając tym samym jeden bajt (znak) na wiersz. By dokładnie dopasować skrypt do wyniku z systemu Windows, należy otworzyć plik w trybie binarnym ('`rb`') lub dodać liczbę bajtów odpowiadającą liczbie wierszy.

Aby wykonać „ambitniejszą” część ćwiczenia (przekazanie pliku obiektu, tak byśmy plik otwierali tylko raz), będziemy musieli użyć metody `seek` wbudowanego obiektu pliku. Metoda ta działa tak samo jak wywołanie `fseek` z języka C (i tak naprawdę wewnętrznie je wywołuje) — przywraca bieżącą pozycję w pliku do przekazanej wartości przesunięcia. Po wywołaniu `seek` przyszłe operacje wejścia i wyjścia odbywają się względem tej nowej pozycji. By przewinąć plik do początku bez zamknięcia go i ponownego otwierania, należy wywołać `file.seek(0)`. Metoda pliku `read` wznowia działanie od bieżącej pozycji w pliku, dlatego w celu ponownego wczytywania go, musimy przewinąć do początku. Poniżej widać, jak mogłyby działać ta sztuczka.

```
def countLines(file):
    file.seek(0)                                     # Przewinięcie do początku pliku
    return len(file.readlines( ))

def countChars(file):
    file.seek(0)                                     # To samo (przewinięcie, jeśli jest konieczne)
    return len(file.read( ))

def test(name):
    file = open(name)                                # Przekazanie obiektu pliku
    return countLines(file), countChars(file) # Plik otwierany tylko raz

>>> import mymod2
>>> mymod2.test("mymod2.py")
(12, 466)
```

2. Instrukcje `from i from *`. Poniżej znajduje się część z `from *`. By wykonać resztę, należy zastąpić `*` za pomocą `countChars`.

```
% python
>>> from mymod import *
>>> countChars("mymod.py")
```

306

3. Wartość `__main__`. Po utworzeniu poprawnego kodu powinno to działać w obu trybach (wykonaniu programu oraz importowaniu modułu).

```
def countLines(name):
    file = open(name)
    return len(file.readlines( ))

def countChars(name):
    return len(open(name).read( ))

def test(name):                      # Lub przekazanie obiektu pliku
    return countLines(name), countChars(name) # Lub zwrócenie słownika

if __name__ == '__main__':
    print(test('mymod.py'))

% python mymod.py
(13, 360)
```

W tym miejscu najprawdopodobniej zacząłbym rozważyć użycie argumentów wiersza poleceń lub informacji od użytkownika w celu podania nazw plików, które mają być zliczane, zamiast zapisywać je na stałe w skrypcie.Więcej informacji na temat sys.argv można znaleźć w rozdziale 24., natomiast więcej o danych od użytkownika — w rozdziale 10.

```
if __name__ == '__main__':
    print(test(input('Wprowadź nazwę pliku:')))

if __name__ == '__main__':
    import sys
    print(test(sys.argv[1]))
```

4. Zagnieżdżone importowanie. Poniżej przedstawiam moje rozwiązańe (plik *myclient.py*).

```
from mymod import countLines, countChars
print(countLines('mymod.py'), countChars('mymod.py'))

% python myclient.py
13 360
```

Jeśli chodzi o resztę tego zadania, funkcje modułu mymod są dostępne (czyli można je importować) z najwyższego poziomu modułu myclient, ponieważ from po prostu przypisuje do zmiennych w pliku importującym (działa to tak samo, jakby instrukcje def modułu mymod znajdowały się w myclient). Przykładowo w kolejnym pliku może się znajdować taki kod:

```
import myclient
myclient.countLines(...)

from myclient import countChars
countChars(...)
```

Gdyby moduł myclient wykorzystał instrukcję import zamiast from, by dotrzeć do funkcji modułu mod za pośrednictwem myclient, musielibyśmy użyć ścieżki.

```
import myclient
myclient.mymod.countLines(...)

from myclient import mymod
mymod.countChars(...)
```

Zawsze możemy zdefiniować moduły *kolektorów* importujące wszystkie zmienne z innych modułów, tak by były one dostępne w jednym module. Wykorzystując poniższy kod,

otrzymujemy trzy różne kopie tej samej zmiennej somename (`mod1.somename`, `collector.somename` oraz `_main_.somename`). Wszystkie trzy współdzielą początkowo ten sam obiekt liczby całkowitej, a tylko zmienna somename istnieje jako taka w sesji interaktywnej.

```
# Plik mod1.py
somename = 42

# Plik collector.py
from mod1 import *
from mod2 import *
from mod3 import *

# Zbiera wiele zmiennych
# from przypisuje do moich zmiennych

>>> from collector import somename
```

1. *Importowanie pakietów.* W celu wykonania tego ćwiczenia umieściłem rozwiążanie `mymod.py` z ćwiczenia 3. w pakiecie katalogów. Na poniższym listingu widać, co zrobiłem w celu ustawienia katalogu oraz wymaganego pliku `__init__.py` w interfejsie konsoli systemu Windows. Kod ten należy odpowiednio zmodyfikować na potrzeby innej platformy (na przykład użyć `mv` oraz `vi` zamiast `move` i `edit`). Takie rozwiązanie działa dla dowolnego katalogu (ja akurat wykonywałem polecenia w katalogu instalacyjnym Pythona). Część tych działań można również wykonać za pomocą graficznego interfejsu użytkownika eksploratora plików.

Po skończeniu otrzymałem katalog `mypkg` zawierający pliki `__init__.py` oraz `mymod.py`. Plik `__init__.py` jest niezbędny w katalogu `mypkg`, jednak nie w jego katalogu nadzrzednym. Katalog `mypkg` umieszczony jest w katalogu głównym podanym w ścieżce wyszukiwania modułów. Warto zwrócić uwagę na to, że instrukcja `print` umieszczona w kodzie pliku inicjalizującego katalogu wykonywana jest tylko przy pierwszej operacji importowania, a przy drugiej już nie.

```
C:\python30> mkdir mypkg
C:\Python30> move mymod.py mypkg\mymod.py
C:\Python30> edit mypkg\__init__.py
...zapisanie instrukcji print...
C:\Python30> python
>>> import mypkg.mymod
inicjalizacja mypkg
>>> mypkg.mymod.countLines('mypkg\mymod.py')
13
>>> from mypkg.mymod import countChars
>>> countChars('mypkg\mymod.py')
360
```

6. *Przeladowywanie.* Ćwiczenie to wymaga jedynie poeksperymentowania z modyfikacją pliku `changer.py` przedstawionego w książce, dlatego nie ma tu nic do pokazania.
7. *Importowanie wzajemne.* Mówiąc w skrócie, importowanie najpierw `recur2` działa, ponieważ importowanie rekurencyjne z `recur1` do `recur2` pobiera `recur2` jako całość, zamiast pobierać wybrane zmienne. Moduł `recur2` jest niekompletny, kiedy importujemy go z `recur1`, jednak ponieważ wykorzystujemy instrukcję `import` zamiast `from`, jesteśmy bezpieczni — Python odnajduje i zwraca utworzony już obiekt modułu `recur2` i kontynuuje wykonywanie reszty `recur1` bez przeskódeł. Kiedy importowanie `recur2` jest wznowiane, druga instrukcja `from` odnajduje zmienną `Y` w module `recur1` (wykonanie było kompletne),

Rozszerzone objaśnienie będzie następujące. Zaimportowanie na początku `recur2` działa, ponieważ importowanie rekurencyjne z `recur1` do `recur2` pobiera `recur2` jako całość, zamiast pobierać wybrane zmienne. Moduł `recur2` jest niekompletny, kiedy importujemy go z `recur1`, jednak ponieważ wykorzystujemy instrukcję `import` zamiast `from`, jesteśmy bezpieczni — Python odnajduje i zwraca utworzony już obiekt modułu `recur2` i kontynuuje wykonywanie reszty `recur1` bez przeskódeł. Kiedy importowanie `recur2` jest wznowiane, druga instrukcja `from` odnajduje zmienną `Y` w module `recur1` (wykonanie było kompletne),

dlatego nie jest zgłoszany błąd. Wykonanie pliku jako skryptu nie jest tym samym co importowanie go jako modułu. Przypadki te są takie same jak wykonanie pierwszej instrukcji `import` lub `from` w sesji interaktywnej. Wykonanie `recur1` jako skryptu jest takie samo jak zainportowanie `recur2` w sesji interaktywnej, ponieważ `recur2` jest pierwszym modulem zainportowanym w `recur1`.

Część VI Klasy i programowanie zorientowane obiektowo

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części szóstej” na końcu rozdziału 31.

1. *Dziedziczenie.* Poniżej znajduje się kod rozwiązania dla tego ćwiczenia (plik `adder.py`) wraz z testami interaktywnymi. Metoda przeciążająca `__add__` musi się pojawić tylko raz, w klasie nadrzędnej, ponieważ wywołuje ona specyficzne dla typu metody `add` w klasach podrzędnych.

```
class Adder:  
    def add(self, x, y):  
        print('Nie zaimplementowano!')  
    def __init__(self, start=[]):  
        self.data = start  
    def __add__(self, other):  
        return self.add(self.data, other)      # Czy w klasach podrzędnych?  
                                              # Czy zwrócić typ?  
  
class ListAdder(Adder):  
    def add(self, x, y):  
        return x + y  
  
class DictAdder(Adder):  
    def add(self, x, y):  
        new = {}  
        for k in x.keys(): new[k] = x[k]  
        for k in y.keys(): new[k] = y[k]  
        return new  
  
% python  
>>> from adder import *  
>>> x = Adder()  
>>> x.add(1, 2)  
Nie zaimplementowano!  
>>> x = ListAdder()  
>>> x.add([1], [2])  
[1, 2]  
>>> x = DictAdder()  
>>> x.add({1:1}, {2:2})  
{1: 1, 2: 2}  
  
>>> x = Adder([1])  
>>> x + [2]  
Nie zaimplementowano!  
>>>  
>>> x = ListAdder([1])  
>>> x + [2]  
[1, 2]  
>>> [2] + x
```

```
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: __add__ nor __radd__ defined for these operands
```

Warto zauważyc, że w ostatnim teście otrzymujemy błąd dla wyrażeń, w których instancja klasy pojawia się po prawej stronie operatora +. Jeśli chcemy to naprawić, musimy użyć metod __radd__, zgodnie z opisem z podrozdziału „Przeciążanie operatorów” z rozdziału 29.

Jeśli w jakiś sposób zapisujemy wartość instancji, możemy równie dobrze przepisać metodę add w taki sposób, by przyjmowała tylko jeden argument, zgodnie z innymi przykładami z tej części książki.

```
class Adder:
    def __init__(self, start=[]):
        self.data = start
    def __add__(self, other):
        return self.add(other)
    def add(self, y):
        print('Nie zaimplementowano!')
```



```
class ListAdder(Adder):
    def add(self, y):
        return self.data + y
```



```
class DictAdder(Adder):
    def add(self, y):
        pass
```

Zmodyfikuj mnie, by użyć self.data zamiast x


```
x = ListAdder([1, 2, 3])
y = x + [4, 5, 6]
print(y)
```

Wyświetla [1, 2, 3, 4, 5, 6]

Ponieważ wartości są raczej dołączane do obiektów niż przekazywane, ta wersja jest bardziej zorientowana obiektywnie. A skoro już doszliśmy aż tutaj, najprawdopodobniej okaże się, że możemy się całkowicie pozbyć metody add i po prostu zdefiniować specyficzne metody __add__ w dwóch klasach podanych.

2. *Przeciążanie operatorów*. Kod rozwiązania (plik *mylist.py*) wykorzystuje kilka metod przeciążania operatorów, o których nie mówiliśmy zbyt wiele w tekście książki, powinny one jednak być łatwe do zrozumienia. Skopiowanie wartości początkowej w konstruktorze jest ważne, ponieważ może ona być zmienna. Nie chcemy modyfikować lub mieć referencji do obiektu, który może być współdzielony gdzieś poza klasą. Metoda __getattr__ przekierowuje wywołania do opakowanej listy. Wskazówki dotyczące łatwiejszego sposobu zapisania tego kodu w Pythonie 2.2 oraz nowszych wersjach można znaleźć w podrozdziale „Rozszerzanie typów za pomocą klas podanych” w rozdziale 31.

```
class MyList:
    def __init__(self, start):
        # self.wrapped = start[:]
        self.wrapped = []
        # Skopiowanie start — brak efektów ubocznych
        # Upewniamy się, że jest to lista
        for x in start: self.wrapped.append(x)
    def __add__(self, other):
        return MyList(self.wrapped + other)
    def __mul__(self, time):
        return MyList(self.wrapped * time)
    def __getitem__(self, offset):
        return self.wrapped[offset]
    def __len__(self):
        return len(self.wrapped)
```

```

def __getslice__(self, low, high):
    return MyList(self.wrapped[low:high])
def append(self, node):
    self.wrapped.append(node)
def __getattr__(self, name):           # Inne składowe: sort, reverse itd.
    return getattr(self.wrapped, name)
def __repr__(self):
    return repr(self.wrapped)

if __name__ == '__main__':
    x = MyList('mielonka')
    print(x)
    print(x[2])
    print(x[1:])
    print(x + ['jajka'])
    print(x * 3)
    x.append('a')
    x.sort()
    for c in x: print(c, end=' ')

```

```
% python mylist.py
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a']
e
['i', 'e', 'l', 'o', 'n', 'k', 'a']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'jajka']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'm', 'i', 'e', 'l', 'o', 'n', 'k', 'a',
 'm', 'i', 'e', 'l', 'o', 'n', 'k', 'a']
a a e i k l m n o

```

Warto zauważyć, że bardzo istotne jest skopiowane wartości początkowej za pomocą dodania do listy, a nie wycinka, ponieważ w innym przypadku wynik mógłby nie być prawdziwą listą i tym samym nie reagowałby na oczekiwane metody listy, takie jak `append` (wycinek łańcucha znaków zwraca kolejny łańcuch znaków, a nie listę). Bylibyśmy w stanie skopiować wartość początkową `MyList` za pomocą wycinka, ponieważ jej klasa przeciąża operację tworzenia wycinka i udostępnia oczekiwany interfejs listy. Musimy jednak uniakać kopiowania opartego na wycinkach dla obiektów takich, jak łańcuchy znaków. Warto również podkreślić, że zbiorы są obecnie w Pythonie typami wbudowanymi, dlatego jest to tak naprawdę tylko ćwiczenie programistyczne (więcej informacji na temat zbiorów można znaleźć w rozdziale 5.).

3. *Klasy podrzędne.* Moja propozycja rozwiązania (plik `mysub.py`) znajduje się poniżej. Rozwiązanie tego ćwiczenia powinno być podobne do poniższego.

```

from mylist import MyList

class MyListSub(MyList):
    calls = 0                                     # Współdzielona przez instancje

    def __init__(self, start):
        self.adds = 0                               # Różni się w każdej instancji
        MyList.__init__(self, start)

    def __add__(self, other):
        MyListSub.calls += 1                         # Licznik dla całej klasy
        self.adds += 1                             # Liczniki dla instancji
        return MyList.__add__(self, other)

    def stats(self):
        return self.calls, self.adds               # Wszystkie dodawania, moje dodawania

if __name__ == '__main__':

```

```

x = MyListSub('mielonka')
y = MyListSub('szynka')
print(x[2])
print(x[1:])
print(x + ['jajka'])
print(x + ['tost'])
print(y + ['bar'])
print(x.stats( ))

```

```
% python mysub.py
e
['i', 'e', 'l', 'o', 'n', 'k', 'a']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'jajka']
['m', 'i', 'e', 'l', 'o', 'n', 'k', 'a', 'tost']
['s', 'z', 'y', 'n', 'k', 'a', 'bar']
(3, 2)

```

1. *Metody atrybutów.* Ja rozwiązałem to ćwiczenie w następujący sposób. Warto zauważyć, że w Pythonie 2.6 operatory próbują również pobrać atrybuty za pomocą metody `__getattr__`. Musimy zwrócić uwagę, by działały. Uwaga: zgodnie z informacjami z rozdziału 30. metoda `__getattr__` nie jest w Pythonie 3.0 wywoływana dla operacji wbudowanych, dlatego poniższe wyrażenie nie będzie działało w taki sposób, jak pokazano poniżej. W wersji 3.0 taka klasa musi redefiniować metodę przeciążania operatora `__X__` w sposób jawnego.Więcej informacji na ten temat można znaleźć w rozdziałach 30., 37. oraz 38.

```

>>> class Meta:
...     def __getattr__(self, name):
...         print('pobierz', name)
...     def __setattr__(self, name, value):
...         print('ustaw', name, value)
...
>>> x = Meta()
>>> x.append
pobierz append
>>> x.spam = "wieprzowina"
ustaw spam wieprzowina
>>>
>>> x + 2
pobierz __coerce__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: 'NoneType' object is not callable
>>>
>>> x[1]
pobierz __getitem__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: 'NoneType' object is not callable

```

```

>>> x[1:5]
pobierz __len__
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: 'NoneType' object is not callable

```

1. *Obiekty zbiorów.* Poniżej znajduje się rodzaj sesji interaktywnej, jaką powinniśmy uzyskać. Komentarze wyjaśniają, która metoda jest wywoływana.

```

% python
>>> from setwrapper import Set
>>> x = Set([1, 2, 3, 4])                                # Wykonuje __init__

```

```

>>> y = Set([3, 4, 5])
>>> x & y                                # __and__ intersect, następnie __repr__
Zbiór:[3, 4]
>>> x | y                                # __or__ union, następnie __repr__
Zbiór:[1, 2, 3, 4, 5]

>>> z = Set("hallo")
>>> z[0], z[-1]                           # __init__ usuwa duplikaty
('h', 'o')
# __getitem__

>>> for c in z: print(c, end=' ')
...
h a l o
>>> len(z), z                            # __len__ __repr__
(4, Zbiór:['h', 'a', 'l', 'o'])

>>> z & "mallo", z | "mallo"
(Zbiór:['a', 'l', 'o'], Zbiór:['h', 'a', 'l', 'o', 'm'])

```

Moje rozwiązywanie klasy podrzędnej rozszerzenia z wieloma argumentami przypomina klasę zaprezentowaną niżej (plik *multiset.py*). Wystarczy jedynie zastąpić dwie metody w oryginalnym zbiorze. Łąncuch znaków dokumentacji klasy wyjaśnia, jak to działa.

```

from setwrapper import Set

class MultiSet(Set):
    """
        Dziedziczy wszystkie zmienne klasy Set, jednak
        rozszerza intersect oraz union w taki sposób, by
        obsługiwały większą liczbę argumentów. Warto
        zauważyc, że "self" nadal jest pierwszym
        argumentem (przechowywanym teraz w argumencie *args).
        Odziedziczone operatory & oraz | wywołują nowe
        metody z dwoma argumentami, jednak przetworzenie
        więcej niż dwóch wymaga wywołania metody, a nie
        wyrażenia.
    """

    def intersect(self, *others):
        res = []
        for x in self:
            for other in others:
                if x not in other: break
            else:
                res.append(x)
        return Set(res)

    def union(*args):
        res = []
        for seq in args:
            for x in seq:
                if not x in res:
                    res.append(x)
        return Set(res)

```

Przeszukanie pierwszej sekwencji
Dla wszystkich pozostałych argumentów
Element w każdym z nich?
Nie — wyjście z pętli
Tak — dodanie elementu na końcu

self to args[0]
Dla wszystkich argumentów
Dla wszystkich węzłów
Dodanie nowych elementów do wyniku

Nasza interakcja z tym rozszerzeniem będzie przypominała poniższą. Warto zauważyc, że możemy obliczać część wspólną zbiorów za pomocą & lub wywołania `intersect`, natomiast dla trzech lub większej liczby argumentów musimy już wywołać `intersect`. Operator & jest operatorem binarnym (dwustronnym). Moglibyśmy również po prostu `Multiset` nazwać `Set`, by zmiana ta była bardziej niewidoczna, gdybyśmy użyli `setwrapper.Set` do odniesienia się do oryginału wewnętrz klasy `multiset`.

```

>>> from multiset import * >>> x = MultiSet([1,2,3,4])
>>> y = MultiSet([3,4,5])
>>> z = MultiSet([0,1,2])

>>> x & y, x | y                                # Dwa argumenty (Zbiór:[3, 4], Zbiór:[1,
2, 3, 4, 5])

>>> x.intersect(y, z)                          # Trzy argumenty
Zbiór:[]
>>> x.union(y, z)
Zbiór:[1, 2, 3, 4, 5, 0]
>>> x.intersect([1,2,3], [2,3,4], [1,2,3])    # Cztery argumenty
Zbiór:[2, 3]
>>> x.union(range(10))                        # Działa również na innych zbiorach Zbiór:[1,
2, 3, 4, 0, 5, 6, 7, 8, 9]

```

6. Łącza drzew klas. Poniżej widać sposób, w jaki ja zmodyfikowałem klasę `Lister`, a także powtórzenie testu pokazującego jej format. W przypadku wersji opartej na funkcji `dir` należy zrobić to samo; podobnie będzie w przypadku formatowania obiektów klas w wariantce z przechodzeniem drzewa klas.

```

class ListInstance:
    def __str__(self):
        return '<Instancja klasy %s(%s), adres %s:\n%s>' % (
            self.__class___.name__,          # Nazwa mojej klasy
            self.__super__( ),              # Moje klasy nadrzędne
            id(self),                      # Mój adres
            self.__attrnames( ) )          # Lista nazwa=wartość
    def __attrnames(self):
        ...bez zmian...
    def __super__(self):
        names = []
        for super in self.__class___.bases__: # Jeden poziom w górę od klasy
            names.append(super.__name__)    # name, nie str(super)
        return ', '.join(names)

C:\misc> python testmixin.py
<Instancja klasy Sub(Super, ListInstance), adres 7841200:
zmienna data1=mielonka
zmienna data2=jajka
zmienna data3=42 >

```

7. Kompozycja. Moja propozycja rozwiązania (plik `lunch.py`) znajduje się poniżej, wraz z komentarzami z opisem umieszczonymi w kod. W tym przypadku wydaje się, że łatwiej jest problem opisać w Pythonie niż w języku polskim.

```

class Lunch:
    def __init__(self):                         # Utworzenie/osadzenie klas Customer oraz Employee
        self.cust = Customer( )
        self.empl = Employee( )
    def order(self, foodName):                  # Rozpoczęcie symulacji zamówienia klienta Customer
        self.cust.placeOrder(foodName, self.empl)
    def result(self):                          # Zapytanie klienta Customer o jego jedzenie Food
        self.cust.printFood( )

class Customer:
    def __init__(self):                         # Inicjalizacja mojego jedzenia na None
        self.food = None
    def placeOrder(self, foodName, employee): # Złożenie zamówienia pracownikowi Employee
        self.food = employee.takeOrder(foodName)

```

```

def printFood(self):
    print(self.food.name)                                # Wyświetlenie nazwy mojego jedzenia

class Employee:
    def takeOrder(self, foodName):                      # Zwrócenie jedzenia Food o żądanej nazwie
        return Food(foodName)

class Food:
    def __init__(self, name):                           # Przechowanie nazwy jedzenia
        self.name = name

if __name__ == '__main__':
    x = Lunch( )
    x.order('burrito')
    x.result( )
    x.order('pizza')
    x.result( )

% python lunch.py
burrito
pizza

```

8. *Hierarchia zwierząt w zoo.* Poniżej znajduje się sposób, w jaki ja utworzyłem taksonomię zwierząt w kodzie napisanym w Pythonie (plik *zoo.py*). Jest ona sztuczna, jednak ten ogólny wzorzec kodu ma zastosowanie do wielu prawdziwych struktur, od graficznych interfejsów użytkownika po bazy danych pracowników. Warto zauważyc, że referencja `self.speak` w `Animal` uruchamia niezależne wyszukiwanie dziedziczenia, które odnajduje metodę `speak` w klasie podrzędnej. Należy przetestować ten kod interaktywnie zgodnie z opisem z ćwiczenia. Można spróbować rozszerzyć tę hierarchię za pomocą nowych klas, a także utworzyć instancje poszczególnych klas drzewa.

```

class Animal:
    def reply(self): self.speak( )                         # Z powrotem do klasy podrzędnej
    def speak(self): print('mięlonka')                   # Własny komunikat

class Mammal(Animal):
    def speak(self): print('he?')

class Cat(Mammal):
    def speak(self): print('miau')

class Dog(Mammal):
    def speak(self): print('hau')

class Primate(Mammal):
    def speak(self): print('Witaj, świecie!')

class Hacker(Primate): pass                            # Dziedziczy po klasie Primate

```

9. *Skecz z martwą papugą.* Poniżej widać, jak ja zaimplementowałem to zadanie (plik *parrot.py*). Warto zwrócić uwagę na to, jak działa metoda `line` w klasie nadrzędnej `Actor`. Dzięki dwukrotnemu dostępowi do atrybutów `self` dwa razy odsyła ona Pythona z powrotem do instancji i tym samym uruchamia dwa wyszukiwania dziedziczenia — `self.name` oraz `self.says()` odnajdują informacje w określonych klasach podrzędnych.

```

class Actor:
    def line(self): print(self.name + ':', repr(self.says( )))

class Customer(Actor):
    name = 'klient'
    def says(self): return "To już ekspapuga!"

```

```

class Clerk(Actor):
    name = 'sprzedawca'
    def says(self): return "nie, wcale nie..."

class Parrot(Actor):
    name = 'papuga'
    def says(self): return None

class Scene:
    def __init__(self):
        self.clerk = Clerk()
        self.customer = Customer()
        self.subject = Parrot()

    def action(self):
        self.customer.line()                      # Osadzenie instancji
        self.clerk.line()                         # Scene to kompozyt
        self.subject.line()                       # Delegacja do osadzonych

```

Część VII Wyjątki oraz narzędzia

Ćwiczenia znajdują się w podrozdziale „Sprawdź swoją wiedzę — ćwiczenia do części siódmej” na końcu rozdziału 35.

1. *Instrukcja try/except.* Moja wersja funkcji oops (plik *oops.py*) znajduje się poniżej. Jeśli chodzi o pytania dodatkowe, modyfikacja oops w taki sposób, by funkcja ta zgłaszała wyjątek KeyError zamiast IndexError oznacza, że program obsługi nie będzie przechwytywał wyjątku (zostanie on przekazany na najwyższy poziom programu i wywoła domyślny komunikat o błędzie Pythona). Nazwy KeyError oraz IndexError pochodzą z najbardziej zewnętrznego zakresu wbudowanego. By to zobaczyć, należy zimportować moduł `__builtins__` (`__builtin__` w Pythonie 2.6) i przekazać go jako argument do funkcji `dir`.

```

def oops():
    raise IndexError

def doomed():
    try:
        oops()
    except IndexError:
        print('przechwycono błąd indeksu!')
    else:
        print('nie przechwycono żadnego błędu...')

if __name__ == '__main__':
    doomed()

% python oops.py
przechwycono błąd indeksu!

```

2. *Obiekty wyjątków oraz listy.* Poniżej widać mój sposób rozszerzenia tego modułu, tak by obsługiwał on mój własny wyjątek.

```

class MyError(Exception):
    pass

def oops():
    raise MyError('Mielonka!')

def doomed():
    try:
        oops()
    except IndexError:

```

```

        print('przechwycono błąd indeksu!')
    except MyError as data:
        print('przechwycono błąd:', MyError, data)
    else:
        print('nie przechwycono żadnego błędu...')

if __name__ == '__main__':
    doomed()

% python oops.py
przechwycono błąd: <class '__main__.MyError'> Mielonka!

```

Tak jak w przypadku wszystkich innych wyjątków opartych na klasach, instancja ta powraca jako dane dodatkowe. Komunikat o błędzie pokazuje teraz zarówno klasę (<...>), jak i jejinstancję (<Mielonka!>). Instancja ta musi dziedziczyć zarówno metodę `__init__`, jak i `__repr__` bądź `__str__` po klasie Pythona o nazwie `Exception` — w przeciwnym razie wyświetlna byłaby tak samo jak klasa. Więcej informacji na temat tego, jak działa to dla wbudowanych klas wyjątków, można znaleźć w rozdziale 34.

3. *Obsługa błędów.* Poniżej znajduje się jeden ze sposobów wykonania tego ćwiczenia (plik `safe2.py`). Swoje testy przeprowadzałem w pliku, a nie w sesji interaktywnej, ale wyniki powinny być podobne.

```

import sys, traceback

def safe(entry, *args):
    try:
        entry(*args)                                # Przechwycenie wszystkiego innego
    except:
        traceback.print_exc()
        print('Mam', sys.exc_info()[0], sys.exc_info()[1])

import oops
safe(oops.oops)

% python safe2.py
Traceback (most recent call last):
  File "safe2.py", line 6, in safe
    entry(*args)                                # Przechwycenie wszystkiego innego
  File "oops.py", line 5, in oops
    raise MyError, 'świecie'
witaj,: świecie
Mam witaj, świecie

```

4. Poniżej znajduje się kilka przykładów kodu, które można przestudiować w miarę posiadania chwili wolnego czasu. Więcej podobnych przykładów znajduje się w innych książkach oraz w Internecie.

Odnalezienie największego pliku źródłowego Pythona w jednym katalogu

```

import os, glob
dirname = r'C:\Python30\lib'

allsizes = []
allpy = glob.glob(dirname + os.sep + '*.py')
for filename in allpy:
    filesize = os.path.getsize(filename)
    allsizes.append((filesize, filename))

allsizes.sort()
print(allsizes[:2])
print(allsizes[-2:])

```

```
# Odnalezienie największego pliku źródłowego Pythona w całym drzewie katalogów
```

```
import sys, os, pprint
if sys.platform[:3] == 'win':
    dirname = r'C:\Python30\Lib'
else:
    dirname = '/usr/lib/python'

allsizes = []
for (thisDir, subsHere, filesHere) in os.walk(dirname):
    for filename in filesHere:
        if filename.endswith('.py'):
            fullname = os.path.join(thisDir, filename)
            fullsize = os.path.getsize(fullname)
            allsizes.append((fullsize, fullname))

allsizes.sort()
pprint pprint(allsizes[:2])
pprint pprint(allsizes[-2:])
```

```
# Odnalezienie największego pliku źródłowego Pythona w ścieżce wyszukiwania
# importowanych modułów
```

```
import sys, os, pprint
visited = {}
allsizes = []
for srcdir in sys.path:
    for (thisDir, subsHere, filesHere) in os.walk(srcdir):
        thisDir = os.path.normpath(thisDir)
        if thisDir.upper() in visited:
            continue
        else:
            visited[thisDir.upper()] = True
        for filename in filesHere:
            if filename.endswith('.py'):
                pypath = os.path.join(thisDir, filename)
                try:
                    pysize = os.path.getsize(pypath)
                except:
                    print('pomijam', pypath)
                allsizes.append((pysize, pypath))

allsizes.sort()
pprint pprint(allsizes[:3])
pprint pprint(allsizes[-3:])
```

```
# Sumowanie rozdzielonych przecinkami kolumn w pliku tekstowym
```

```
filename = 'data.txt'
sums = {}

for line in open(filename):
    cols = line.split(',')
    nums = [int(col) for col in cols]
    for (ix, num) in enumerate(nums):
        sums[ix] = sums.get(ix, 0) + num

for key in sorted(sums):
    print(key, '=', sums[key])
```

```

# Podobnie do poprzedniego przykładu, jednak dla sum wykorzystuje listy,
# a nie słowniki

import sys
filename = sys.argv[1]
numcols = int(sys.argv[2])
totals = [0] * numcols

for line in open(filename):
    cols = line.split(',')
    nums = [int(x) for x in cols]
    totals = [(x + y) for (x, y) in zip(totals, nums)] 

print(totals)

# Testowanie regresji w danych wyjściowych zbioru skryptów

import os
testscripts = [dict(script='test1.py', args=''),
               dict(script='test2.py', args='spam')]

for testcase in testscripts:
    commandline = '%(script)s %(args)s' % testcase
    output = os.popen(commandline).read()
    result = testcase['script'] + '.result'
    if not os.path.exists(result):
        open(result, 'w').write(output)
        print('Utworzony:', result)
    else:
        priorresult = open(result).read()
        if output != priorresult:
            print('NIEPOWODZENIE:', testcase['script'])
            print(output)
        else:
            print('Przeszedł:', testcase['script'])

# Utworzenie graficznego interfejsu użytkownika za pomocą tkinter
# (w Pythonie 2.6 Tkinter) z przyciskami zmieniającymi kolor oraz wielkość

from tkinter import *                                         # W Pythonie 2.6 należy użyć Tkinter
import random
fontsize = 25
colors = ['red', 'green', 'blue', 'yellow', 'orange', 'white', 'cyan', 'purple']

def reply(text):
    print(text)
    popup = Toplevel()
    color = random.choice(colors)
    Label(popup, text='Okno wyskakujące', bg='black', fg=color).pack()
    L.config(fg=color)

def timer():
    L.config(fg=random.choice(colors))
    win.after(250, timer)

def grow():
    global fontsize
    fontsize += 5
    L.config(font=('arial', fontsize, 'italic'))
    win.after(100, grow)

```

```

win = Tk( )
L = Label(win, text='Mielonka', font=('arial', fontsize, 'italic'), fg='yellow',
    bg='navy', relief=RAISED)
L.pack(side=TOP, expand=YES, fill=BOTH)
Button(win, text='Naciśnij', command=(lambda: reply('red'))).pack(side=BOTTOM,
    fill=X)
Button(win, text='Licznik', command=timer).pack(side=BOTTOM, fill=X)
Button(win, text='Powiększ', command=grow).pack(side=BOTTOM, fill=X)
win.mainloop( )

# Podobnie do poprzedniego przykładu, ale wykorzystuje klasy, tak by każde okno
# miało własne informacje o stanie

from tkinter import *
import random

class MyGui:
    """
    Graficzny interfejs użytkownika z przyciskami zmieniającymi kolor i powiększającymi napis
    """
    colors = ['blue', 'green', 'orange', 'red', 'brown', 'yellow']

    def __init__(self, parent, title='Okno wyskakujące'):
        parent.title(title)
        self.growing = False
        self.fontsize = 10
        self.lab = Label(parent, text='Gu11', fg='white', bg='navy')
        self.lab.pack(expand=YES, fill=BOTH)
        Button(parent, text='Mielonka', command=self.reply).pack(side=LEFT)
        Button(parent, text='Powiększ', command=self.grow).pack(side=LEFT)
        Button(parent, text='Zatrzymaj', command=self.stop).pack(side=LEFT)

    def reply(self):
        "losowa modyfikacja koloru przycisku po naciśnięciu przycisku Mielonka"
        self.fontsize += 5
        color = random.choice(self.colors)
        self.lab.config(bg=color, font=('courier', self.fontsize, 'bold italic'))

    def grow(self):
        "po naciśnięciu przycisku Powiększ podpis zaczyna rosnąć"
        self.growing = True
        self.grower()

    def grower(self):
        if self.growing:
            self.fontsize += 5
            self.lab.config(font=('courier', self.fontsize, 'bold'))
            self.lab.after(500, self.grower)

    def stop(self):
        "po naciśnięciu przycisku Zatrzymaj przycisk przestaje rosnąć"
        self.growing = False

    class MySubGui(MyGui):
        colors = ['black', 'purple']           # Można dostosować w celu modyfikacji kolorów do wyboru

MyGui(Tk( ), 'main')
MyGui(Toplevel( ))
MySubGui(Toplevel( ))
mainloop( )

```

```

# Narzędzie do przeglądania folderu poczty przychodzącej oraz utrzymywania jej
"""

przegląda skrzynkę e-mailową POP, pobierając tylko
nagłówki i pozwalając na usuwanie bez pobierania pełnej wiadomości

import poplib, getpass, sys

mailserver = 'nazwa serwera POP naszej poczty' # pop.rmi.net
mailuser = 'identyfikator użytkownika poczty' # brian
mailpasswd = getpass.getpass('Hasło dla %s?' % mailserver)

print('Łączenie...')
server = poplib.POP3(mailserver)
server.user(mailuser)
server.pass_(mailpasswd)

try:
    print(server.getwelcome( ))
    msgCount, mboxSize = server.stat( )
    print('W skrzynce jest', msgCount, 'wiadomości e-mail, rozmiar ', mboxSize)
    msginfo = server.list( )
    print(msginfo)
    for i in range(msgCount):
        msgnum = i+1
        msgsize = msginfo[1][i].split( )[1]
        resp, hdrlines, octets = server.top(msgnum, 0) # Pobranie tylko nagłówków
        print('*'*80)
        print('[%d: octets=%d, size=%s]' % (msgnum, octets, msgsize))
        for line in hdrlines: print(line)

        if input('Wyświetlić?') in ['y', 'Y']:
            for line in server.retr(msgnum)[1]: print(line) # Pobranie całej wiadomości
        if input('Usunąć?') in ['y', 'Y']:
            print('usuwanie')
            server.dele(msgnum) # Usunięcie z serwera
        else:
            print('pomijanie')
    finally:
        server.quit() # Należy odblokować skrzynkę mbox
input('Żegnaj.') # Zachowanie okna

```

Skrypt CGI po stronie klienta wchodzący w interakcję z przeglądarką

```

#!/usr/bin/python
import cgi
form = cgi.FieldStorage( ) # Przetworzenie danych formularza
print("Content-type: text/html\n") # Nagłówek i pusty wiersz
print("<HTML>")
print("<title>Strona odpowiedzi</title>") # Strona html z odpowiedzią
print("<BODY>")
if not 'user' in form:
    print("<h1>Kim jesteś?</h1>")
else:
    print("<h1>Witaj, <i>%s</i>!</h1>" % cgi.escape(form['user'].value))
print("</BODY></HTML>")

```

Skrypt bazy danych zapełniający bazę MySQL i wykonujący do niej zapytania

```

from MySQLdb import Connect
conn = Connect(host='localhost', user='root', passwd='darling')

```

```

curs = conn.cursor( )
try:
    curs.execute('drop database testpeopledb')
except:
    pass # Nie istniała

curs.execute('create database testpeopledb')
curs.execute('use testpeopledb')
curs.execute('create table people (name char(30), job char(10), pay int(4))')

curs.execute('insert people values (%s, %s, %s)', ('Robert', 'programista',
    50000))
curs.execute('insert people values (%s, %s, %s)', ('Zuzanna', 'programista',
    60000))
curs.execute('insert people values (%s, %s, %s)', ('Anna', 'menedżer', 40000))

curs.execute('select * from people')
for row in curs.fetchall():
    print(row)

curs.execute('select * from people where name = %s', ('Robert',))
print(curs.description)
colnames = [desc[0] for desc in curs.description]
while True:
    print('-' * 30)
    row = curs.fetchone()
    if not row: break
    for (name, value) in zip(colnames, row):
        print('%s => %s' % (name, value))

conn.commit() # Zapisanie wstawionych rekordów

```

Skrypt bazy danych wypełniający obiekt shelve obiektami Pythona

Zachęcam do zjrzenia również do przykładów z rozdziału 27. (shelve) oraz z rozdziału 30. (pickle)

```

rec1 = {'name': {'first': 'Robert', 'last': 'Kowalski'},
        'job': ['programista', 'menedżer'],
        'age': 40.5}

rec2 = {'name': {'first': 'Zuzanna', 'last': 'Zielona'},
        'job': ['menedżer'],
        'age': 35.0}

```

```

import shelve
db = shelve.open('dbfile')
db['bob'] = rec1
db['sue'] = rec2
db.close()

```

Skrypt bazy danych wyświetlający i aktualniający obiekt shelve utworzony # w poprzednim skrypcie

```

import shelve
db = shelve.open('dbfile')
for key in db:
    print(key, '=>', db[key])

bob = db['bob']
bob['age'] += 1
db['bob'] = bob
db.close()

```

Skorowidz

- , 153
-, 153
'', 198
!=, 152, 153
#!, 91
%, 152, 153, 221
&, 153
(...), 153, 154
*, 153, 472, 473, 475, 476, 477, 481, 482, 486
**, 153, 471, 472, 473, 475, 476, 481, 486, 1045
..., 362
.NET, 76
/, 153, 161
//, 153, 161

`_nonzero__`, 738, 753
`_oct__`, 732
`_or__`, 729
`_print__`, 781
`_radd__`, 728, 729, 745
`_repr__`, 729, 742, 743, 759, 781, 883
`_set__`, 729, 963
`_setattr__`, 496, 710, 729, 740, 809, 956, 971, 990
`_setitem__`, 729, 730, 731
`_setslice__`, 732
`_slots__`, 659, 687, 805, 807, 969, 1036
`_stdout__`, 340
`_str__`, 656, 674, 687, 689, 729, 742, 743, 778, 883, 982
`_sub__`, 728
`_X__`, 99, 128, 655
`_reps`, 540
`_thread`, 446
`_X`, 608
`{...}`, 153, 154, 178
`|`, 153
`~`, 153
`+`, 153
`<<`, 152, 153
`>>`, 152
`==`, 193, 286
`>>>`, 153
`>>>`, 82
`2to3`, 341

A

abs, 168, 169, 538
abstrakcyjne klasy nadzędne, 712
 Python 2.6, 713
 Python 3.0, 713
ActivePython, 108
add, 177
adnotacje funkcji, 499, 1053
agregacja, 684, 760
AI, 58
akcesory, 447
aliasy zmiennych, 466
all, 530
alternatywne implementacje Pythona, 75
analiza short-circuit, 354
analiza składniowa drzewa DOM, 949
analiza składniowa tekstu, 218
analiza składniowa XML, 948
and, 153, 354
any, 530
anydbm, 257
Apache, 56
API do tworzenia skryptów, 54

apostrofy, 200, 204
apostrofy odwrotne, 152
append, 130, 131, 242, 245, 268
apply, 478
architektura MVC, 56
architektura programu, 556, 557
argumenty, 427, 428, 430, 465, 467, 488, 497
 *, 473, 476
 **, 473, 476
 _reps, 540
 formy dopasowywania, 472
 metody deskryptorów, 962
 pad, 530
 przekazywanie, 465
 rozpakowywanie, 476
 słowa kluczowe, 472
 tryby dopasowania argumentów, 470
 uniwersalne stosowanie funkcji, 477
varargs, 471
wartości domyślne, 451, 472, 474
współdzielone referencje, 466
argumenty dekoratora, 1006, 1035, 1053
argumenty mogące być tylko słowami
 kluczowymi, 471, 479, 487, 542
 reguły dotyczące kolejności, 481
 zastosowanie, 482
argumenty wiersza poleceń, 211
 _name__, 611
argumenty ze słowami kluczowymi, 245, 335, 474, 476, 489, 668
argv, 211
ArithmeticalError, 881
arytmetyka liczb zmiennoprzecinkowych, 170
as, 302, 329, 614, 841, 853, 867
as_integer_ratio, 152
ASCII, 71, 129, 141, 199, 212, 514, 912, 913
assert, 302, 841, 865, 1044
 wyłapywanie ograniczeń, 866
AssertionError, 865, 866
associative array, 248
atrybuty, 557, 569, 955
 _bases__, 718, 721
 _cause__, 865
 _class__, 718, 720, 1069
 _context__, 865
 _dict__, 576, 718
 _doc__, 722
 _slots__, 805, 807
atrybuty danych, 704
atrybuty funkcji, 461, 499, 547, 1011
atrybuty instancji, 688, 1009
atrybuty introspektywne, 1062
atrybuty klas, 648, 688, 705

atrybuty metaklas, 1077
atrybuty modułu, 98
atrybuty odziedziczone, 779
atrybuty prywatne, 1033
atrybuty pseudoprывatne, 767, 768
AttrDisplay, 689
automatyczne przekierowanie strumienia, 340
automatyczne zarządzanie pamięcią, 61
automatyczne zwalnianie przestrzeni obiektu, 188
automatycznie wykonywany kod, 996

B

backtracking, 843
BaseException, 874, 880
baza danych, 57, 698
 SQL, 368
BDFL, 60
bezpośrednie zarządzanie funkcjami, 1031
bezpośrednie zarządzanie klasami, 1031
bezwzględne importowanie pakietów, 593, 600
białe znaki, 127
biblioteka standardowa, 50, 559
biblioteki, 98
biblioteki narzędzi, 61
bieżący katalog roboczy, 564
big-endian, 917
bin, 151, 166
binarne AND, 167
binarne OR, 167
binarne pliki z danymi, 202
bit, 167
bit_length, 152, 168
bitowe AND, 153
bitowe NOT, 153
bitowe OR, 153
bitowe XOR, 153
BitTorrent, 54
Blender, 58
bliskość kodu, 504
blok pętli else, 361
bloki, 348
 bloki zagnieżdżone, 349
blokowe łańcuchy znaków, 204
błędny, 111, 312, 848, 893
 błędny liczbowe, 881
 TypeError, 857
BOM, 917, 941
bool, 181, 289
Boolean, 143, 181, 357
brakująca deklaracja typu, 185
break, 302, 311, 360, 361, 363, 370
budowanie, 73

build, 73
builtins, 169, 442, 449
Button, 785
byte code, 53, 72
byte order mark, 917
bytearray, 199, 214, 236, 281, 912, 915, 917, 932
 metody, 933
 modyfikacja w miejscu, 933
bytes, 199, 275, 291, 912, 915, 917, 920, 929
 formatowanie łańcuchów znaków, 930
 mieszanie typów łańcuchów znaków, 931
 operacje na sekwencjach, 930
 sposoby tworzenia obiektów, 931
bytes.decode, 920

C

C++, 63
callable, 773
callback, 748
CGI, 56, 343, 697
cgi.FieldStorage, 257
chr, 212
ciało funkcji, 428
ciało wyrażenia lambda, 502
class, 121, 144, 145, 192, 302, 407, 408, 639, 648,
 649, 658, 703, 704
classmethod, 812, 815
classestools, 689
close, 140, 272, 273
closure, 450
cmath, 165
cohesion, 491
COM, 57
command, 506
compile, 292
complex, 151, 291
complex number, 150
concatenation, 125
context manager, 869
contextlib, 870
continue, 302, 361, 363
control flow, 348
copy, 192, 284
CORBA, 57
coroutines, 523
count, 140, 270
coupling, 491
cPickle, 279
CPython, 75
csh, 91
cudzysłowy, 198, 200
cykl tworzenia oprogramowania, 74

cykliczne struktury danych, 293
Cython, 57, 79
czas wykonywania, 428
część wspólna zbiorów, 153
czyszczenie pamięci, 135, 188, 193

D

Dabo, 56
dane binarne, 272, 911, 912, 929, 945
dane XML, 948
dbm, 691
DBM, 257
debuger, 111, 904
debugowanie, 106, 111
 zewnętrzne instrukcje try, 895
decimal, 171
Decimal, 170, 171
decimal.Decimal.from_float, 171
decode, 215, 920
def, 121, 144, 302, 407, 425, 427, 428, 429, 440, 648,
 649, 706
definicja funkcji, 429
definiowanie
 dekoratory, 997
 funkcje zwrotne, 506
 klasy, 144
deformatowanie obiektów, 692
deklaracja atrybutów instancji, 650
deklaracja metaklas, 714, 1071
deklaracja przestrzeni nazw, 443
deklaracja slotów, 807
deklaracja typu kodowania znaków pliku
 źródłowego, 928
deklaracja w czasie wykonywania, 998
deklaracja zmiennych, 427
dekodowanie, 914
dekoratory, 451, 820, 960, 995, 997
 adnotacje funkcji, 1053
 argumenty, 1006, 1035
 definiowanie, 997
 dodatkowe wywołania, 1029
 obiekty opakowujące, 996
 składnia, 997
 sprawdzanie poprawności
 argumentów funkcji, 1044
 sprawdzanie przedziałów dla argumentów
 pozycyjnych, 1045
 sprawdzanie typów, 1054
 uogólnianie kodu pod kątem słów
 kluczowych, 1047
 wady, 998
 zagnieżdżanie dekoratorów, 1004

zakresy funkcji zawierającej, 1035
zalety, 1030
zarządzanie funkcjami, 996
zarządzanie klasami, 996
zastosowanie, 997, 1029
dekoratory funkcji, 714, 749, 766, 811, 820, 821,
 995, 998, 1029, 1044, 1063
 atrybuty funkcji, 1011
 atrybuty instancji klasy, 1009
 dodawanie argumentów dekoratora, 1019
 implementacja, 999
 kod, 1007
 metody, 1012
 mierzenie czasu wywołania, 1017
 obsługa dekoracji metod, 1000
 składnia, 820
 śledzenie wywołań, 1007
 zachowanie informacji o stanie, 1009
zakresy funkcji zawierających, 1010
zakresy zawierające, 1010
zarządzanie obiektami funkcji, 996
zastosowanie, 998
zmienne globalne, 1010
zmienne nielokalne, 1010
dekoratory klas, 684, 742, 766, 822, 996, 1002, 1063,
 1066, 1088
 funkcje zarządzające, 1028
 implementacja, 1002
 implementacja atrybutów prywatnych, 1033
 klasy singletona, 1021
 kod, 1021
 metaklasy, 1081
 obsługa większej liczby instancji, 1003
 śledzenie interfejsów obiektów, 1023
 zachowanie większej liczby instancji, 1027
 zarządzanie obiektami klas, 996
 zastosowanie, 1002
dekoratory metod, 461
dekorowanie argumentów funkcji, 501
dekorowanie metod klas, 1012
 deskryptory, 1015
 zagnieżdżone funkcje, 1013
del, 248, 302, 303
delegacja, 683, 765, 997, 1035
delegacja metod przeciążania operatorów, 1039
deprecation, 331
design patterns, 643
deskryptory, 741, 961, 962, 1001
 __delete__, 964
 __get__, 962
 __set__, 963
 argumenty metod, 962
dekorowanie metod klas, 1015

deskryptory atrybutów klas, 1063
deskryptory klas, 806
deskryptory plików, 894
deskryptory tylko do odczytu, 963
informacje o stanie, 967
obliczanie atrybutów, 966
sprawdzanie poprawności atrybutów, 988
właściwości, 968
destrukcja obiektu, 752
desygnator typu, 187
dict, 257, 291
 fromkeys, 258
dictionary, 248
dictionary comprehension, 259
dir, 100, 127, 128, 177, 240, 406, 779
distutils, 563, 567, 905
Django, 56, 697
długość łańcucha znaków, 124, 206
docstring, 348
docstrings, 405, 407
doctest, 903
dodawanie metod do klas, 1078
dodawanie prawostronne, 745
dodawanie w miejscu, 746
dokumentacja, 405, 722
 __doc__, 407, 722
 dir, 406
 docstrings, 405, 407
 help, 410
komentarze ze znakami #, 406
lista atrybutów dostępnych wewnętrz
 obiektu, 406
łańcuchy znaków dokumentacji, 405, 407
łańcuchy znaków dokumentacji zdefiniowane
 przez użytkownika, 407
PyDoc, 405, 410, 723
 wbudowane łańcuchy znaków dokumentacji, 409
 zbiór standardowej dokumentacji, 415
DOM, 948, 949
domknięcie, 450
domyślne wartości argumentów, 451
 zmienne pętli, 453
domyślny program obsługi wyjątków, 843
dopasowywanie argumentów, 470, 472
 *, 472, 475
 **, 472, 475
łączenie słów kluczowych i wartości
 domyślnych, 475
rozpakowywanie argumentów, 476
składnia, 471
słówka kluczowe, 473, 474
wartości domyślne, 474
dopasowywanie wzorców, 129, 944, 949
DOS, 82, 94
dostęp do atrybutów, 741, 766, 962
dostęp do bazy danych, 519
dostęp do krotek, 267
dostęp do metod, 770
dostęp do plików DBM, 257
dostęp do przestrzeni nazw modułu, 576
dostęp do słownika przestrzeni nazw modułu, 577
dostęp do słowników, 248
dostęp do zmiennych globalnych, 447
dostosowanie większych systemów
 do własnych potrzeb, 583
drzewa przestrzeni nazw, 709
drzewo klas, 638
dwukropki, 417
działania końcowe, 842, 846, 857
działania matematyczne, 122
działania na zbiorach, 180
dziedziczenie, 441, 578, 633, 634, 637, 651, 652, 675,
 679, 684, 708, 757, 759, 762, 1035
 abstrakcyjne klasy nadzędne, 712
 atrybuty, 651
 klasy nadzędne, 651
 klasy podrzędne, 651, 675
 metaklasy, 1077
 modelowanie relacji ze świata rzeczywistego, 759
 przeciążanie, 652
 przesłanianie, 652
 rozszerzanie metod, 676
 specjalizacja odziedziczonych metod, 710
 techniki interfejsów klas, 711
 tworzenie drzewa atrybutów, 709
 wydobywanie atrybutów odziedziczonych, 779
wyszukiwanie dziedziczenia, 652
 zmiana logiki, 651
dziedziczenie diamentowe, 801
dziedziczenie wielokrotne, 639, 775, 778
 kolejność podawania klas nadzędnych, 826
 problemy, 826
dzielenie, 161
 dzielenie bez reszty, 153, 161, 162
 dzielenie klasyczne, 161
 dzielenie prawdziwe, 161
 dzielenie z odcinaniem, 162
dzielenie przestrzeni nazw systemu, 556
dziesiętne menedżery kontekstu, 172

E

Eclipse, 107
eggs, 63, 567
EIBTI, 51
eksperymentowanie, 83

elastyczność kodu, 143
elastyczność obiektów, 282
ElementTree, 948
elif, 139, 302, 312, 346
elipsy, 362
else, 153, 302, 313, 314, 361, 364, 851
else (pętle), 364
Emacs, 108
emulacja
 funkcja map, 527, 528
 funkcja print z Pythona 3.0, 486
 funkcja zip, 527
 pętla while z języka C, 366
 prawdliwość w atrybutach instancji, 741
encapsulation, 671
encode, 920
end, 360
endswith, 219
enkapsulacja, 757
enumerate, 379, 394
env, 92
EOFError, 893
escape sequence, 200
ESRI, 54
eval, 277, 484
EVE Online, 54, 55
event handler, 748
except, 302, 313, 841, 844, 851, 853
except as, 863
exception, 841
Exception, 845, 854, 876, 880
exceptions, 880
exec, 81, 101, 102, 329
executable scripts, 91
expression, 152
extend, 246

F

fabryka, 647, 669
fabryka obiektów, 668, 785
 zastosowanie, 787
factory, 785
factory function, 450
faktoryzacja, 426, 836
False, 143, 181, 288, 289, 354, 442, 444
falsz, 288
FIFO, 142
file, 281, 291, 372
filter, 133, 138, 394, 397, 398, 508, 515
finally, 302, 841, 842, 846, 857
find, 126, 216
flagi, 167

float, 174
floating-point number, 150
floor, 163, 169
floor division, 153
flush, 272, 280
for, 136, 137, 207, 242, 302, 365, 417, 518
 else, 365, 366
 przechodzenie równolegle, 376
 przetwarzanie sekwencji, 367
 przypisanie krotek, 368
 rozszerzona składnia rozpakowania
 sekwencji, 325
 rozszerzone przypisanie sekwencji
 w Pythonie 3.0, 369
 sortowanie kluczy słowników, 136
 techniki tworzenia pętli, 372
 zaginione pętle, 370
format, 127, 169, 221, 225
 atrybuty, 226
 automatyczne numerowanie argumentów, 233
 formatowanie specjalizowane, 227
 jawne odwołania do wartości, 232
 klucze, 226
 możliwości, 232
 parametry formatowania, 226
 przesunięcia, 226
 Python 3.1, 231
 uogólnione argumenty, 233
 wyrażenia formatujące, 229, 234
formatowanie łańcuchów znaków, 127, 221, 611, 930
formatowanie obiektów, 692
formats, 611
formatujące łańcuchy znaków, 166
formaty wyświetlania liczb, 158
formy dopasowywania argumentów funkcji, 472
Fraction, 172
framework, 643
freeze, 78, 905
from, 97, 98, 99, 100, 302, 555, 571, 573, 623
 pakiety, 589
 problemy z użyciem, 574
 przypisanie, 572
 rekurencyjne importowanie, 626
 testowanie interaktywne, 625
from *, 571, 588, 608, 624
from as, 614
from_float, 174
fromkeys, 258, 260
frozen binaries, 78
frozenset, 281
FTP, 56
func.__code__, 1050
func.func_code, 1050

function decorator, 749, 820
fundacja PSF, 58
funkcja fabryki, 450
funkcja porównująca, 246
funkcje, 332, 347, 425
 __import__, 566
 abs, 168, 169, 538
 adnotacje, 499
 all, 530
 any, 530
 append, 242, 245
 apply, 478
 argumenty, 427, 428, 430, 465
 argumenty mogące być tylko słowami kluczowymi, 471, 479
 argumenty opcjonalne, 474
 argumenty ze słowami kluczowymi, 474
 atrybuty, 461, 499, 547
 bin, 151, 166
 chr, 212
 ciało, 428
 classmethod, 812, 815
 complex, 151
 copy, 192
 def, 427, 428
 definicja, 429, 432
 dir, 100, 128, 240, 406, 779
 enumerate, 379, 394
 eval, 277, 484
 exec, 101
 filter, 133, 515
 floor, 163, 169
 format, 169
 formy dopasowywania argumentów, 472
 from, 555
 funkcje zagnieździone, 548
 generatory, 427, 520
 getattr, 615
 getrefcount, 194
 help, 128, 215, 219, 410
 hex, 151, 166, 732
 imp.reload, 555, 561, 647
 import, 555
 input, 310
 instrukcje, 425
 int, 151, 169, 312
 introspekcja, 498
 isdigit, 312
 isinstance, 182, 291
 iter, 386, 387
 keys, 388
 lambda, 427, 501
 len, 124, 201, 206, 251
 list, 217
 map, 133, 243, 377, 395, 507, 519, 527, 529
 max, 168
 min, 168
 nazwy, 428
 nazwy lokalne, 544
 next, 388
 niezależność od otoczenia, 492
 obiekty, 497
 obiekty funkcji, 427
 obiekty mutowalne, 546
 oct, 151, 166, 732
 odwzorowywanie na sekwencje, 507
 open, 142, 203, 280, 917
 ord, 212
 parametry, 428
 polimorfizm, 430, 433
 pośrednie wywołania funkcji, 497
 pow, 168, 169
 print, 334, 340, 670
 problemy, 544
 projektowanie, 491
 property, 741, 956, 957, 1063
 przechowywanie stanu, 427
 przekazywanie argumentów, 427
 przekazywanie dowolnej liczby argumentów, 471
 range, 242, 373, 536
 rekurencja, 439
 reload, 97, 580, 581, 588, 624, 625
 replace, 214
 repr, 152, 159, 182, 211
 return, 427, 428, 430
 reversed, 247
 round, 163, 169
 rozbijanie zadania na funkcje, 491
 rozmiar, 492
 set, 142, 176, 396
 sin, 168
 singleton, 1022
 sort, 245, 246
 sorted, 246, 247, 262, 269, 287, 394, 395
 spójność, 492
 sprzęganie, 491, 492
 sqrt, 168, 169
 staticmethod, 812, 815, 817, 820
 str, 152, 159, 171, 182, 211
 sum, 168
 sygnalizowanie warunków, 893
 sys.exc_info, 896
 time.clock, 539
 time.time, 539
 trace, 540
 trunc, 163, 169

funkcje
tryby dopasowywania argumentów, 470
tworzenie, 426, 428
type, 143, 772
wartości domyślne, 546
while, 373
wyrażenia, 425
wywołanie, 418, 429, 430, 432
xrange, 398
zakres, 438
zastosowanie, 426
zip, 259, 394, 527, 529
zmienne, 427
zmienne lokalne, 433
zmienne modyfikowane w pętli, 548
zwracanie wyniku, 427, 428
funkcje akcesorów, 447
funkcje anonimowe, 152, 501
funkcje fabryczne, 450
funkcje generatorów, 520, 525, 529
next, 523
protokół iteratorów, 521
rozszerzony protokół funkcji generatorów, 523
send, 523
stosowanie, 521
współbieżność, 523
yield, 520, 522
zawieszanie stanu, 520
funkcje iteratorów, 525
funkcje matematyczne, 152
funkcje niezwracające wyników, 548
funkcje pomocnicze, 1064
funkcje rekurencyjne, 493
implementacje alternatywne, 494
obsługa dowolnych struktur, 496
pętle, 495
sumowanie, 493
funkcje zagnieżdzone, 448
funkcje zarządzające, 1064, 1065

G

garbage collection, 135, 188, 752
generalne kategorie typów, 235
generator powtarzający, 525
GeneratorExit, 880
generatory, 133, 137, 427, 520, 521, 530
generowanie wyników we wbudowanych typach, 531
jednorazowe iteratory, 526
map, 530
zawieszanie stanu, 520
zip, 530
generatory obiektów iterowanych, 402

generowanie
dane wejściowe z konsoli, 310
liczby losowe, 123
obiekty, 647
wartości elementów, 379
wartości przesunięcia, 379
wyniki we wbudowanych typach, 531
get, 139
getattr, 615
getrefcount, 194
GIL, 79
global, 302, 425, 427, 440, 443, 444, 445, 447, 459, 1010
Global Interpreter Lock, 79
gniazda, 142, 280
Google, 54
Google's App Engine, 697
graficzny interfejs użytkownika, 55, 697
grafika, 58
granice bloków, 348
graphical user interface, 55
group, 944
groups, 129, 944
grupowanie podwyrażeń, 155
gry, 58
GTK, 56
GUI, 53, 55, 634

H

has_key, 139, 251, 263
has-a, 760
hash, 248
hash bang, 91
hashing, 251
help, 128, 215, 219, 410
hermetyzacja, 671, 757, 836, 998
hex, 151, 166, 732
hierarchia dziedziczenia, 633
hierarchia klas, 651
hierarchia typów, 290
hierarchia wyjątków, 877, 879
HTML, 129, 205
HTMLGen, 56

I

IDE, 71, 102, 903
identyczność obiektów, 153, 286
IDLE, 70, 81, 101, 102, 103
 Debug, 106
 Debugger, 106
debugowanie, 106

File, 103
kolorowanie kodu, 103
komunikaty o błędach, 104
New Window, 103
opcje zaawansowane, 106
problemy z połączeniem, 106
programy z wątkami, 105
Run, 103
Run Module, 103, 105
stosowanie, 105
tkinter, 105
tryb z jednym procesem, 106
uruchamianie skryptów, 105
zapisywanie plików, 105

if, 138, 139, 153, 302, 303, 345, 1044
elif, 139, 312, 346
else, 313
nawiasy, 304
ograniczniki bloków, 349
rozgałęzienia kodu, 346
testy prawdziwości, 353
wyrażenie trójargumentowe, 355

immutable, 126
imp, 97
imp.reload, 555, 561, 647
implementacja atrybutów prywatnych, 1033
implementacja współdzielonych usług, 556
implementacje Pythona, 75
import, 96, 98, 99, 100, 121, 302, 555, 570, 573, 575
import as, 614
import hook, 566
importowanie modułów, 96, 557, 559, 571
łańcuch znaków nazwy, 617
rekurencyjne importowanie, 626
zakres, 578
importowanie pakietów, 560, 583, 585, 588, 589, 590
__init__.py, 586
bezwzględne importowanie, 593, 594, 600
bezwzględne ścieżki pakietów, 596
from, 589
import, 589
import spoza pakietów, 598
import wewnątrz pakietów, 599
katalog roboczy, 600, 601
Python 3.0, 593
reguły wyszukiwania modułów, 598
sys.path, 594
ścieżka wyszukiwania, 586
względne importowanie, 593, 594, 598
względne importowanie w Python 3.0, 596
zakres importów względnych, 597

in, 139, 153, 206, 219, 242, 251, 263, 365
indeksowanie, 153, 207, 208, 730
Python 2.6, 732

indeksowanie listy, 243
indentacja, 305, 309, 349, 417
index, 140, 247, 270
IndexError, 843, 855, 863
informacje o stanie, 144, 666
informacje o stanie wyjątku, 875
Informix, 57
inicjalizacja
 instancje, 640
 obiekty, 640
 pakiety, 587
 przestrzeń nazw modułu, 587
inkrementalne prototypowanie, 667
input, 94, 310
insert, 247
inspekcja stosu wywołań, 1042
instancje, 636, 637, 647, 648, 658, 665
instrukcje, 301
 as, 302, 867
 assert, 302, 865
 break, 302, 311, 360, 361, 363
 class, 145, 192, 302, 408, 639, 648, 703
 continue, 302, 361, 363
 def, 302, 407, 427, 428, 706
 del, 302
 elif, 139, 302, 312
 else, 302, 313, 361, 364
 except, 302, 313, 844
 finally, 302, 842, 846, 857
 for, 302, 365
 from, 97, 302, 571, 589, 623
 global, 302, 427, 443
 if, 139, 302, 303, 345
 import, 100, 302, 570, 589
 input, 96
 instrukcje sterujące przebiegiem programu, 348
 instrukcje wielowierszowe, 86
 instrukcje wykonywalne, 648
 instrukcje złożone, 85, 304, 348
 instrukcje złożone w sesji interaktywnej, 86
 koniec wcięcia, 305
 koniec wiersza, 305
 lambda, 501
 modyfikacje w miejscu, 333
 nonlocal, 302, 427, 455
 ograniczniki, 351
 pad, 530
 pass, 302, 361
 print, 82, 334
 przypisanie, 83, 317
 raise, 302, 845, 863
 reload, 97, 100
 return, 302, 427, 428, 430, 469, 520, 548

- instrukcje
składnia indentacji, 306
try, 139, 302, 313, 842
while, 302, 310, 311, 359
with, 172, 302, 867
wyrażenia, 332
wywołanie, 332
yield, 302, 402, 427, 520
- int, 151, 169, 291, 312
integer, 150
integracja komponentów, 50, 56
integrated development environment, 102
interaktywne pętle, 310
interaktywne wykonywanie kodu, 82
interaktywny wiersz poleceń, 81, 439
 >>>, 82
 interaktywne wykonywanie kodu, 82
- interfejs użytkownika IDLE, 102
interfejsy, 144
interfejsy do słowników, 257
interpreter, 69, 72
introspekcja, 615, 684, 686, 1050
 funkcje, 498
- iRobot, 54
IronPython, 56, 75, 76
is, 152, 153, 193, 286
is not, 153
is_integer, 152
is-a, 649, 759
isalpha, 127, 219
isdigit, 127, 312
isinstance, 182, 291
items, 252, 260, 287, 397
iter, 386, 387
iterable object, 384
iteracja, 137, 383, 513
 iteracja po indeksie, 731
 iteracja po listach, 242
- iteratory, 251, 260, 383, 393, 402, 520
 filter, 398
 iter, 386
 iteratory plików, 272, 384, 519
 iteratory widoku słownika, 400
 iteratory zdefiniowane przez użytkownika, 734
 kilka iteratorów na tym samym obiekcie, 399
 kontrola iteracji, 386
 map, 398
 next, 386
 pomiary wydajności implementacji, 535
 protokół iteracyjny, 384
 przetwarzanie kluczy, 388
 range, 373, 397
 słowniki, 388
- typy wbudowane, 388
wiele iteracji po jednym obiekcie, 735
zip, 398
złożenia, 524

J

- jakość oprogramowania, 49, 50
Java, 63
Java Virtual Machine, 75
jadro języka Python, 29
jednorazowe iteracje, 532
jednorazowe iteratory, 526
jednostki programów, 121
język C++, 63
język Java, 63
język Lisp, 64
język Perl, 63
język PHP, 64
język Python, 49
język Ruby, 64
język skryptowy, 51, 52, 72
język SmallTalk, 64
język Tcl, 63
język Visual Basic, 64
język zarządzania, 52
język zorientowany obiektowo, 59
JIT, 76
join, 217
JPype, 56
JPython, 75
jump table, 347, 503
just-in-time, 76, 77
JVM, 75
Jython, 56, 75, 603

K

- kapsułkowanie, 671
katalog główny programu, 562
katalogi biblioteki standardowej, 562
katalogi ścieżek plików .pth, 562
kategorie typów, 235, 281
kategorie wyjątków, 875
 wbudowane wyjątki, 881
KeyboardInterrupt, 880
keys, 138, 251, 260, 261, 388, 397, 401
keyword-only argument, 479
klasy, 144, 408, 460, 633, 636, 637, 647, 658, 665, 690
 __bases__, 659, 688
 __class__, 659, 686
 __dict__, 659, 686
 __slots__, 659, 687

`__str__`, 687
abstrakcyjne klasy nadrzędne, 712
atrybuty, 648, 705
atrybuty danych, 704
atrybuty pseudoprывatne, 767
atrybuty specjalne, 686
BaseException, 874
class, 703
definicja, 144
delegacja, 683
dodawanie metod, 1078
dosłosowanie zachowania, 675, 679
dosłosowywanie do własnych potrzeb, 635
dziedziczenie, 634, 635, 637, 651, 675, 679, 708
dziedziczenie diamentowe, 801
dziedziczenie wielokrotne, 775, 826
Exception, 845
hierarchia klas, 651
inicjalizacja instancji, 640
instancje, 636, 637, 648, 658
introspekcja, 686
klasy abstrakcyjne, 762
klasy nadrzędne, 636, 651
klasy opakowujące, 765
klasy podrzędne, 636, 651, 675, 793
klasy pośredniczące, 765
klasy zdefiniowane przez użytkownika, 144
kod, 703
kompozycja, 634, 684
kompozyt, 683
konflikty nazw zmiennych, 689
konstruktory, 640, 655, 666
metody, 144, 638, 640, 648, 649, 671, 706
metody niezwiązane, 772
moduły, 653
modyfikacja atrybutów, 824
modyfikacja mutowalnych atrybutów, 825
nadklasy, 636
object, 800
operacje na typach wbudowanych, 655
operatory, 654, 655
platforma, 643
podklasy, 636
problemy, 824
przechwytywanie wbudowanych metod, 727
przeciążanie operatorów, 635, 654, 673, 727
przesadne opakowywanie, 829
przestrzenie nazw, 704, 714
rozszerszanie metod, 676
self, 639
singleton, 1021
składowe, 650
słowniki, 660

specjalizacja odziedziczonych metod, 710
tworzenie, 638, 647
tworzenie instancji, 666
type, 1070
wydobywanie atrybutów odziedziczonych, 779
wyświetlanie składowych, 687
wywołanie konstruktorów klas nadrzędnych, 708
zakresy zagnieżdżone, 827
zastosowanie, 634
klasy mieszane, 775
tworzenie, 776
klasy w nowym stylu, 795
`__getattribute__`, 797, 811
`__slots__`, 805
deskryptory, 811
dziedziczenie diamentowe, 801
jawne rozwiązywanie konfliktów, 802
 kolejność wyszukiwania nazw
 w drzewie dziedziczenia, 796
kontrola typów, 798
metaklasy, 811
model typów, 797
nowości, 796
object, 800
przeciążanie nazw, 811
sloty, 805
właściwości, 809
wyszukiwanie atrybutów dla funkcji
 wbudowanych, 796
zakres zmian kolejności wyszukiwania, 804
klasy wyjątków, 873
 ArithmetricError, 881
 BaseException, 880
 domyślne wyświetlanie, 882
 Exception, 876, 880
 hierarchie wyjątków, 877, 879
 kategorie wbudowanych wyjątków, 881
 OverflowError, 881
 przechwytywanie kategorii wyjątków, 876
 stan, 882
 szczegóły wyjątku, 877
 tworzenie, 875
 udostępnianie metod wyjątków, 885
 udostępnianie szczegółów wyjątku, 884
 wbudowane klasy wyjątków, 880
 zgłaszanie instancji, 876
klasyfikacja typów obiektów, 282
klucz, 134, 248
kod bajtowy, 53, 72
kod generyczny, 806
kod języka C++, 77
kod klas, 703
kodłańcuchów znaków Unicode, 920

kod samosprawdzający, 609, 616
kod wykonywany automatycznie, 1063
kod źródłowy, 72
kodowanie znaków, 128, 913
 UTF-8, 914
kolejki FIFO, 142
kolekcje
 krotki, 267
 zbiory, 176
komentarze, 88, 157, 348
komentarze ze znakami #, 406
Komodo, 107
kompilacja kodu bajtowego, 72
kompilacja modułów, 560
kompilator C++, 77
kompilator JIT, 76, 77
 Psyco, 76
kompilowane rozszerzenia, 53
komponenty oprogramowania, 675
kompozycja, 634, 684, 760, 762
kompozyt, 683
komunikaty o błędach, 104, 843
koncepcje projektowania funkcji, 491
konfiguracja ścieżki wyszukiwania, 563
konflikt nazw, 100
koniec bloku, 305
koniec wcięcia, 305
koniec wiersza, 305
 przypadki specjalne, 308
konkatenacja, 125
 listy, 241
konstruktory, 640, 655, 666, 728
 dostosowanie, 680
kontekst iteracyjny, 393, 477
kontrola iteracji, 386
kontynuacja wiersza, 351, 352
konwencje dotyczące nazewnictwa, 330
konwersja
 kod znaków, 212
 krotki, 270
 łańcuchy znaków, 210
 typy danych, 155
 typy mieszane, 157
kopiowanie obiektów, 192, 283
 kopie najwyższego poziomu, 285
krok, 209
krotki, 99, 139, 157, 236, 255, 267, 268, 271
 append, 268
 count, 140, 270
 dostęp do krotek, 267
 indeksowanie, 269
 index, 140, 270
 konkatenacja, 269

konwersje, 269
listy składane, 270
literałы, 268
metody, 269
nawiasy, 269
niezmiennaś, 139, 269, 270
operacje, 268
powtórzenie, 269
przecinki, 269
przypisanie w pętli for, 368
składnia, 269
wycinek, 269
zagnieżdżanie, 268
zastosowanie, 140
krotki zmiennych, 519
ksh, 91
kwalifikacja, 98
kwalifikacja nazw atrybutów, 440, 577
kwalifikacja zmiennych, 577

L

lambda, 121, 152, 153, 425, 427, 440, 452, 501
last-in-first-out, 247
LEGB, 440, 441, 442
len, 124, 127, 201, 206, 251
LGB, 440
liczby, 122, 149, 235, 236
 działania matematyczne, 122
 dzielenie, 161
 formatowanie za pomocą łańcucha znaków, 159
 formaty wyświetlania, 158
 funkcje matematyczne, 152
 liczby losowe, 123, 170
 liczby wymierne, 172
 liczby zmiennoprzecinkowe, 123, 150
math, 168
metody specyficzne dla poszczególnych
 typów, 152
moduły liczbowe, 123
narzędzia, 151
operacje poziomu bitowego, 167
operatory wyrażeń, 152
pierwiastek kwadratowy, 169
porównania, 160
potęgowanie, 168
przetwarzanie, 168
rozszerzenia numeryczne, 182
systemy liczbowe, 165
wartość bezwzględna, 168
liczby całkowite, 123, 150
precyzja, 164
Python 2.6, 150
Python 3.0, 151

liczby dziesiętne, 142, 170
dziesiętne menedżery kontekstu, 172
globalne ustawienie precyzyji, 171
liczby ułamkowe, 142, 172
dokładność liczb, 173
Fraction, 172
from_float, 174
konwersje, 174
typy mieszane, 174
liczby zespolone, 151, 165
cmath, 165
licznik referencji, 187, 189, 194
LIFO, 247
Lisp, 64
list, 217, 291
list comprehension, 132, 390
lista atrybutów obiektu, 777
lista sys.path, 564
ListInstance, 786
ListTree, 786
listy, 130, 153, 236, 239, 271
append, 130, 131, 242, 245
długość, 239
dodawanie elementu, 245
dodawanie obiektu, 130
dostęp do elementów, 239
extend, 246
indeksowanie, 131, 243
iteracje, 242
konkatenacja, 241
macierze, 243
map, 243
modyfikacja, 375
modyfikacja w miejscu, 244
odwracanie kolejności elementów, 131, 246
operacje, 241, 247
operacje specyficzne dla typu, 130
pop, 130, 131, 247
powtórzenie, 241
przypisywanie do indeksu, 244
przypisywanie do wycinków, 244
reverse, 131, 246, 247
sekwencje, 130
składanie, 242
sort, 131, 245
sorted, 246
sortowanie, 131, 245
sprawdzanie granic, 131
stos, 247
usuwanie części listy, 247
usuwanie elementu, 247
usuwanie obiektu, 130
właściwości, 239
wycinki, 243
wywołania metod list, 245
zagnieżdżanie list, 131, 243
zwrócenie ostatniego elementu, 247
listy składane, 132, 270, 378, 383, 390, 513, 518, 528
filter, 133, 515
generatory, 133
macierze, 517
map, 133, 514
narzędzia funkcyjne, 513
next, 133
pętle zagnieżdżone, 515
pliki, 391
rozszerzona składnia, 392
składnia, 514
tworzenie, 390
warunki, 515
zastosowanie, 519
literały, 121
literały dwójkowe, 150, 151
literały krotek, 268
literały liczb całkowitych oraz zmiennoprzecinkowych, 150
literały liczb zespolonych, 150, 151
literały liczbowe, 150
literały list, 241
literały łańcuchów znaków, 199, 351
literały ósemkowe, 150, 151
literały słowników, 250
literały szesnastkowe, 150, 151
literały zbiorów, 178
little-endian, 917
logerror, 885
logika operacji programu, 301
logika samosprawdzająca, 616
lokalizacja przypisania do nazwy, 437

Ł

łańcuchy bajtowe, 911
łańcuchy wyjątków, 865
łańcuchy znaków, 124, 197, 205, 236, 913
analiza składniowa tekstu, 218
apostrofy, 200, 204
ASCII, 129, 199
bajt zerowy, 202
białe znaki, 127
blokowe łańcuchy znaków, 204
bytearray, 214, 915, 932
bytes, 915, 929
bytes.decode, 920
cudzysłowy, 200
decode, 215

łańcuchy znaków
dekodowanie, 914
dekodowanie tekstu spoza zakresu ASCII, 922
długość, 124, 206
dopasowywanie wzorców, 129, 944
endswith, 219
find, 126, 216
format, 127, 221, 225
formatowanie, 127, 221
formatowanie specjalizowane, 227
in, 219
indeksowanie, 207
indeksy, 124
isalpha, 127, 219
join, 217
kodowanie, 128
kodowanie łańcuchów Unicode, 923
kodowanie tekstu spoza zakresu ASCII, 922
kodowanie tekstu z zakresu ASCII, 921
kodowanie znaków, 912, 913
kody typów w wyrażeniach formatujących, 223
kolumny, 125
koniec łańcucha, 202
konkatenacja, 125, 217
konwersja, 210, 919
konwersja kodowania, 925
konwersja kodu znaków, 212
konwersja na wielkie litery, 127
len, 127
liczba znaków, 201
list, 217
lista z elementów, 217
literał, 198, 199, 918
łączenie, 125
metody, 214, 215
metody specyficzne dla typu, 126
modyfikacja, 213
narzędzia do konwersji, 210
niezmienność, 126, 213
null, 202
operacje, 198, 206
potrójne cudzysłowy, 204
powtórzenie, 125
przeszukiwanie, 126
przypisanie do zmiennej, 124
Python 3.0, 912, 915, 918, 944
re, 944
replace, 126, 216, 217
rstrip, 127, 219
sekwencje, 124
sekwencje ucieczki, 200, 201
split, 127, 218
sprawdzanie obecności podłańcucha, 219
sprawdzanie zawartości, 127, 219
startswith, 219
str, 199, 915
str.encode, 920
surowe łańcuchy znaków, 203
typy łańcuchów znaków, 915
unicode, 915
Unicode, 129, 199, 911, 925
upper, 127, 219
usuwanie białych znaków na końcu, 127, 219
UTF-8, 914
własności, 918
wycinki, 125, 207, 209, 211
wyrażenia formatujące, 221, 222
wyrażenia formatujące z użyciem słownika, 224
wyszukiwanie, 206, 216
zastępowanie, 126
zastępowanie podłańcucha, 216
zmiana wielkości liter, 219
znaki międzynarodowe, 129
znaki niedrukowalne, 202
łańcuchy znaków dokumentacji, 348, 405, 407, 613, 722, 902
łańcuchy znaków dokumentacji zdefiniowane przez użytkownika, 407
standardy, 408
wbudowane łańcuchy znaków dokumentacji, 409
łącznia przestrzeni nazw, 720
łączenie klas, 683
łączenie łańcuchów znaków, 125, 218

M

macierze, 243
listy składane, 517
maksimum, 484
maksymalizacja ponownego wykorzystania kodu, 426
manpage, 410
map, 133, 138, 243, 376, 377, 395, 397, 398, 507, 514, 519, 527, 529
emulacja z użyciem funkcji, 528
generatory, 530
Python 2.6, 377
mapping, 133
maszyna wirtualna Pythona, 73
match, 129, 944
math, 123, 152, 168
 floor, 163
 sqrt, 169
 trunc, 163
max, 168
Maya, 54, 58

mechanizm dekoratorów, 820
mechanizm slotów, 786
member, 650
menedżery kontekstu, 273, 867, 869
 __enter__, 869
 __exit__, 869
menedżery kontekstu plików, 280
protokół zarządzania kontekstem, 868
 with, 867
menedżery oparte na delegacji, 983
metaclasses, 1061
metafunkcje, 820, 998
metaklasy, 714, 798, 811, 820, 822, 996, 1030,
 1061, 1063
 __new__, 1074
 atrybuty, 1077
 deklaracja, 1071
 dekoratory klas, 1066, 1081
 dodawanie metod do klas, 1078
 dosłosowywanie tworzenia do własnych
 potrzeb, 1074
 dziedziczenie, 1077
 funkcje fabryczne, 1075
 inicjalizacja, 1074
 instancje, 1077
 protokół instrukcji class, 1071
 przeciążenie wywołań tworzących klasę, 1075
 sposoby tworzenia metaklas, 1074
 tworzenie, 1073
 type, 1068, 1070
 zarządzanie instancjami, 1082
 zastosowanie, 1061, 1066
metaprogramy, 615
metody, 144, 215, 332, 408, 638, 640, 648, 649, 671,
 706, 707, 770
 __add__, 655, 656, 728, 729
 __bool__, 729, 738, 751, 753
 __call__, 747
 __call__, 729, 774
 __cmp__, 738, 750
 __contains__, 729, 737, 738
 __del__, 729, 752
 __delattr__, 729
 __delete__, 729, 964
 __delitem__, 729
 __dict__, 777
 __enter__, 729, 869
 __eq__, 729, 750
 __exit__, 729, 869
 __ge__, 729
 __get__, 729, 962
 __getattr__, 729, 740, 765, 809, 829, 956, 970, 975
 __getattribute__, 729, 741, 797, 811, 956, 970,
 975, 991, 1042
 __getitem__, 729, 730, 731, 737, 739
 __getslice__, 732
 __gt__, 729, 750
 __iadd__, 729, 746
 __index__, 729, 732
 __init__, 640, 655, 666, 681, 708, 728, 729
 __iter__, 729, 733, 734, 737
 __le__, 729
 __len__, 729, 751
 __lt__, 729, 738, 750
 __ne__, 729, 750
 __new__, 729
 __next__, 729, 733
 __nonzero__, 738, 753
 __or__, 729
 __radd__, 729, 745
 __repr__, 729, 742
 __set__, 729, 963
 __setattr__, 729, 740, 809, 956
 __setitem__, 729, 730, 731
 __setslice__, 732
 __str__, 656, 674, 687, 729, 742
 __sub__, 728
argument self, 650
dekoratory funkcji, 1012
konstruktorzy, 655
obiekty, 770
obiekty metod instancji z wiązaniem, 770
obiekty metod klas bez wiązania, 770
polimorfizm, 758
self, 660, 706
tworzenie, 669
wywołanie, 215, 638, 706, 708
wywołanie konstruktorów klas nadzędnych, 708
 zakresy zagnieżdżone, 827
metody dostępu do zmiennych globalnych, 447
metody instancji, 812, 816
metody klas, 811
 stosowanie, 815
 zliczanie instancji, 818
 zliczanie instancji dla każdej z klas, 819
metody niezwiązane, 772
metody pomiaru czasu, 1021
metody przechwytywania atrybutów, 1063
metody przeciążania operatorów, 640, 728, 729, 1063
metody statyczne, 708, 811, 812, 816
 alternatywy, 814
 Python 2.6, 812
 Python 3.0, 812
 stosowanie, 815
zliczanie instancji, 817

metody z wiązaniem, 775
metody związane, 773
MFC, 56
mierzenie czasu wywołania, 1017
mieszanie, 251
mieszanie tabulatorów i spacji, 350
min, 168
minimalizacja modyfikacji dokonywanych pomiędzy plikami, 446
minimalizacja powtarzalności kodu, 426
minimalizacja stosowania zmiennych globalnych, 445
minimum, 482
mix-in class, 776
mod_python, 56
model metaklas, 1068
model programowania, 50
model typów dynamicznych, 186
model wykonywania kodu Pythona, 73
modelowanie relacji ze świata rzeczywistego, 759
Model-View-Controller, 56
model-widok-kontroler, 56
Module Docs, 70
modulo, 515
moduły, 87, 123, 555, 556, 607, 723
 __all__, 608
 __dict__, 577, 619
 __file__, 577
 __future__, 608
 __main__, 439, 609, 621
 __metaclass__, 1072
 __name__, 577, 609
 __thread__, 446
 _X, 608
 atrybuty, 98, 557, 569, 653
 biblioteka standardowa, 558
 builtins, 169, 442
 clastools, 689
 cmath, 165
 cPickle, 279
 dbm, 691
 decimal, 171
 distutils, 567
 dostęp do przestrzeni nazw modułu, 576
 formats, 611
 from, 571, 623
 from *, 571, 608, 624
 from as, 614
 import as, 614
 importowanie, 96, 555, 557, 559, 570, 571
 importowanie rekurencyjne, 626
 importowanie za pomocą łańcucha znaków nazwy, 617
 klasy, 653
kod samosprawdzający, 609, 616
 kolejność instrukcji, 622
 kompilowanie, 560
 konfiguracja ścieżki wyszukiwania, 563
 kwalifikacja, 98
 kwalifikowanie nazw atrybutów, 577
 maksymalizacja spójności modułów, 621
 math, 123, 168
 metaprogramy, 615
 mierzenie wydajności, 539
 mieszane tryby użycia, 609
 modyfikacja ścieżki wyszukiwania modułów, 613
 modyfikacja zmiennych pomiędzy plikami, 573
 mytimer, 536
 nazwa pliku modułu, 577
 nazwa widoczna dla kodu importującego, 577
 nazwy modułów, 569
 NumPy, 182
 odnalezienie modułu, 560
 opcje trybu użycia, 609
 pakiety, 585, 603
 pickle, 278, 662, 691, 692, 947
 pliki modułów, 99
 problemy, 622
 profile, 543, 904
 projektowanie modułów, 621
 przechodnie przeładowywanie modułów, 618
 przeładowywanie, 96, 555, 580
 przestrzeń nazw, 98, 100, 555, 575
 punkty zaczepienia operacji importowania, 566
 queue, 446
 random, 170
 re, 129, 292, 944
 reguły wyszukiwania modułów, 598
 reload, 624
 shelve, 278, 691, 692, 693
 string, 220
 struct, 945
 sys, 194, 280
 ścieżka wyszukiwania modułów, 101, 560, 561
 testowanie interaktywne, 625
 testy jednostkowe, 610
 threading, 446
 time, 536
 timeit, 543
 tkinter, 697
 Tkinter, 697
 tworzenie modułów, 569
 ukrywanie danych, 607
 użycie modułów, 570
 włączanie opcji z przyszłych wersji Pythona, 608
 wybór pliku modułu, 565
 wykonanie kodu, 101, 561

zagnieździanie przestrzeni nazw, 579
zakres, 576, 578
zakres globalny, 438
zastosowanie, 555
moduły rozszerzeń, 570
modyfikacja
 argumenty, 468
 atrybuty klas, 824
 łańcuchy znaków, 213
 mutowalne atrybuty klas, 825
 obiekty, 294
 ścieżka wyszukiwania modułów, 613
modyfikacje dokonywane pomiędzy plikami, 446, 573
modyfikacje w miejscu, 191, 333, 745
 listy, 244
 obiekty, 439
 słowniki, 251
Mono, 76
Monty Python's Flying Circus, 63
multiple inheritance, 639, 775
multiprocessing, 55
mutable, 130
MVC, 56
myconfig.pth, 563
MySQL, 57, 698
mytimer, 536

N

nadklasy, 636, 651
namespace, 98
narzędzia, 61
 narzędzia do analizy składniowej XML, 948
 narzędzia do konwersji łańcuchów znaków, 210
 narzędzia do optymalizacji wykonywania, 76
 narzędzia introspekcji, 686
 narzędzia liczbowe, 151, 168
 narzędzia powłoki, 52, 55
 narzędzia programistyczne, 902
 narzędzia udostępniane przez programistów, 62
nawiasy, 155, 269, 352
nawracanie, 843
nazwy, 100, 437, 715
 __name__, 330
 atrybuty, 440, 715
 funkcje, 428
 klasy narzędzi, 689
 konwencje, 330
 moduły, 569
 nazwy globalne, 441
 nazwy lokalne, 441
 nazwy pseudoprzywatne, 778, 1038

nazwy wbudowane, 330
obiekty, 331
pliki, 71
pliki modułów, 330
self, 330
wielkość liter, 329
zmienne, 329, 440
NET, 56
NetBeans, 108
next, 133, 137, 386, 388, 521, 523
nierówność, 152
niezmienność, 126
nieznana liczba argumentów, 471
niezwykły przebieg sterowania, 843
NLTK, 58
None, 143, 288, 428, 442, 444
nonlocal, 302, 329, 425, 427, 439, 440, 455, 456
not, 153
not in, 153
notacja dwójkowa, 165
notacja ósemkowa, 165
notacja szesnastkowa, 165
NotImplementedError, 712
NULL, 288
Numeric Python, 182
NumPy, 50, 53, 57, 182

O

obiekt typu, 143
obiekt wycinka, 730
obiekty, 120, 144, 145, 186, 187, 430
 automatyczne zwalnianie przestrzeni obiektu, 188
 czyszczenie pamięci, 188
 desygnator typu, 187
 identyczność, 286
 inicjalizacja, 640
 konstruktory, 640
 kopie, 283
 licznik referencji, 187, 189, 194
 nazwy, 331
 None, 288
 pamięć podręczna, 194
 pickle, 691
 referencje, 187, 189, 283
 równość, 286
 serializacja, 278, 691
 shelve, 691
 stan, 460, 648
 trwałość obiektów, 691
 type, 1068
 wywołanie metody, 215
obiekty cykliczne, 293

obiekty funkcji, 427, 497
 __annotations__, 499, 500
 __code__, 498
 __name__, 498
adnotacje funkcji, 499
argumenty, 497
atrybuty funkcji, 499
introspekcja funkcji, 498
 wyrażenia lambda, 502
obiekty instancji, 647, 648
obiekty iteratorów, 733
 __iter__, 733
 iteratory zdefiniowane przez użytkownika, 734
 wiele iteracji po jednym obiekcie, 735
obiekty iterowane, 384
 Python 3.0, 397
obiekty klas, 647
 zachowania domyślne, 648
obiekty kodu, 498
obiekty metod instancji z wiązaniem, 770
obiekty metod klas bez wiązania, 770
obiekty mutowalne, 546
obiekty opakowujące, 765, 996
obiekty plików, 140
obiekty przestrzeni nazw, 668
obiekty składane, 533
obiekty typów, 291
obiekty wbudowane, 120
obiekty widoków, 260
obiekty wycinków, 209, 210
obiekty wyjątków, 873
obiekty wywoływane, 773
object, 800
object persistence, 57, 691
object-oriented, 633
object-oriented programming, 49
object-relational mappers, 698
obliczanie atrybutów, 959, 966, 974
obliczenia, 311
obsługa bibliotek, 50
obsługa błędów, 312, 842, 848
 try, 313
obsługa BOM, 941
obsługa programowania dużych systemów, 61
obsługa przypadków specjalnych, 842
obsługa różnych wersji Pythona, 162
obsługa slotów, 786
obsługa tekstu Unicode, 911
obsługa wyjątków, 313, 842
oct, 151, 166, 732
ODBC, 57
odcinanie, 162, 164
odczytywanie

lista atrybutów obiektu, 777
pliki Unicode, 939
odnalezienie modułu, 560
odśmiecanie, 752
odwołanie do atrybutu, 153
odwracanie sekwencji, 247
odwzorowania, 133, 235
odwzorowanie funkcji na sekwencje, 507
odwzorowanie obiektowo-relacyjne, 698, 1062
ograniczniki bloków, 349
ograniczniki instrukcji, 351
OLPC, 54
OO, 633
OOP, 49, 633
opcje wykonywania kodu, 78, 108
open, 140, 141, 142, 203, 272, 280, 292, 917
operacje
 na bitach, 167
 na krotkach, 268
 na listach, 241, 247
 na łańcuchach znaków, 206
 na odwzorowaniach, 134
 na plikach, 272
 na sekwencjach, 124, 130
 na słownikach, 250
operator precedence, 154
operatory, 152, 153
 ==, 286
 and, 153, 354
 Boolean, 354, 357
 dzielenie, 161
 dzielenie bez reszty, 153
 in, 153, 206
 is, 152, 153, 193, 286
 is not, 153
 lambda, 153
 not, 153
 or, 153, 354
 porównania, 153, 154, 160
 priorytety, 154
 przeciążanie operatorów, 156, 282, 654, 657,
 673, 727
 reguły precedencji, 154
 yield, 153, 154
opróżnienie bufora wyjściowego na dysk, 272
optymalizacja, 76, 137, 905
or, 153, 354
Oracle, 57, 698
ord, 212
ORM, 698, 1062
os, 280
os.popen, 281, 389, 896
os.system, 85, 896

osadzanie kodu HTML, 129
osadzanie obiektów, 683
osadzanie wywołań, 108
osadzony tryb wykonywania, 109
OSCON, 59
otwieranie pliku, 141, 272
OverflowError, 881
override, 652

P

p2p, 54
package import, 585
pad, 530
pakietы, 585, 603, 662
 __init__.py, 586
 importowanie, 585, 588
 importowanie bezwzględne, 593, 600
 importowanie względne, 593, 594, 595, 598
 importowanie względne w Python 3.0, 596
inicjalizacja, 587
inicjalizacja przestrzeni nazw modułu, 587
pliki pakietów, 586
Python 3.0, 593
 ustawienia ścieżki wyszukiwania, 586
 zakres importów względnych, 597
parametry, 428
parametry wyjścia, 469
Parrot, 79
pass, 302, 361
PATH, 88
pdb, 112, 904
PEP, 58
Perl, 63
pexpect, 281
pętle, 359
 blok pętli else, 361, 364
 break, 361, 363
 continue, 361, 363
 domyślne wartości argumentów
 w zmiennych pętli, 453
 elipsy, 362
 emulacja pętli while z języka C, 366
 for, 136, 137, 365
 generowanie wartości przesunięcia, 379
 modyfikacja list, 375
 pass, 361
 pętle interaktywne, 310
 pętle nieskończone, 360
 przechodzenie niewyczerpujące, 374
 przechodzenie równoległe, 376
rekurencja, 495
techniki tworzenia pętli, 372

while, 137, 310, 311, 359
wyjście z pętli, 311
pętle liczników, 373
 range, 373
 while, 373
PHP, 64
pickle, 57, 277, 278, 279, 662, 691, 692, 764, 947
pierwiastek kwadratowy, 169
PIL, 58
platforma, 643
 .NET, 76
pliki, 87, 90, 140, 271
 __init__.py, 586
 buforowanie, 273
 close, 272, 273
 file, 281
 flush, 272, 280
 iteratory, 272
 listy składane, 391
 menedżery kontekstu, 280
 metody, 141
 narzędzia, 280
 open, 272
 operacje, 140, 272
 opróżnienie bufora wyjściowego, 272
 otwieranie, 141, 272
przechowywanie obiektów Pythona
 w plikach, 276
przechowywanie obiektów Pythona
 za pomocą pickle, 277
przestrzeń nazw, 576
przetwarzanie, 141, 274
przetwarzanie spakowanych danych
 binarnych, 278
pth, 562
py, 71, 72
pyc, 72, 560, 566
pyo, 566
Python 3.0, 275
read, 141, 272
readline, 272, 274, 276
rozmiar, 141
seek, 141, 272, 280
serializacja, 278
spakowane dane binarne, 278
tryb binarny, 917
tryb przetwarzania, 917
tryb tekstowy, 917
tworzenie, 272
wczytywanie, 272, 273
wczytywanie wiersza, 272
write, 272, 274
writeln, 272

pliki
wykorzystywanie plików, 273
XML, 949
zamykanie, 272, 894
zapisywanie łańcucha, 272
zawartość pliku, 273
zmiana pozycji w pliku, 272
pliki binarne, 141, 202, 272, 275, 917, 929, 935
Python 3.0, 936
pliki deskryptorów, 142, 280
pliki dostępne po kluczu, 142, 281
pliki FIFO, 280
pliki modułów, 87, 99
pliki oparte na deskryptorze, 142
pliki shelve, 694
 uaktualnianie obiektów, 695
pliki tekstowe, 141, 272, 275, 917, 935
 BOM, 917
 Python 3.0, 936
 UTF-16, 917
 UTF-32, 917
 UTF-8, 917
 znacznik kolejności bajtów, 917
pliki Unicode, 939
 BOM, 941
dekodowanie błędnych dopasowań, 940
dekodowanie danych wejściowych pliku, 940
kodowanie danych wyjściowych pliku, 939
kodowanie ręczne, 939
odczytywanie, 939
 Python 2.6, 943
 zapisywanie, 939
pliki źródłowe, 928
PMW, 56, 78
pobieranie atrybutu, 215
pobieranie Pythona, 70
podklasy, 636, 651
podprocedury, 426
polimorficzny model programowania, 431
polimorfizm, 125, 194, 343, 430, 433, 678, 757,
 758, 836
pomiary czasu, 537
pomiary wydajności implementacji iteratorów, 535
 argumenty mogące być tylko słowami
 kluczowymi, 542
moduły mierzące wydajność, 539
mytimer, 536
pomiary czasu, 537
profile, 543
skrypt mierzący wydajność, 536
timeit, 543
pomoc, 127
ponowne wykorzystanie kodu, 425, 426, 556, 558,
 641, 836
pop, 130, 131, 247, 253
porównania, 153, 154, 160, 285, 353, 750
 Python 3.0, 246
rozmiar słowników, 263
słowniki, 287
porty szeregowe, 58
POSIX, 55
PostgreSQL, 57, 698
pośrednie wywołania funkcji, 497
potęgowanie, 168
potoki, 142, 280
potrójne cudzysłowy, 198, 204
pow, 168, 169
powiadomienia o zdarzeniach, 842
powtarzalność kodu, 425
powtórzenia, 125, 241, 293
prawda, 285, 288
Precyza liczby całkowitych, 164
print, 71, 82, 83, 85, 90, 95, 111, 141, 158, 205, 302,
 303, 329, 334, 336, 339, 340, 343, 670
 argumenty ze słowami kluczowymi, 335
 emulacja funkcji z Pythona 3.0, 486
 end, 335, 360
 file, 335
 format wywołania, 335
 formy instrukcji, 337
 Python 2.6, 337
 Python 3.0, 334
 sep, 335
 wyświetlanie niezależne od wersji, 341
printf, 223
priorytety operatorów, 154
Private, 1037
problemy programistyczne, 417
proceduralne podzielenie na części, 426
procedury, 332, 426
procesor strumienia danych, 762
profile, 138, 543, 904
program, 87, 665
 struktura, 556
 „witaj świecie”, 338
program obsługi wyjątków, 842, 843
program obsługi zdarzeń, 748
program profilujący, 903
programowanie, 74
 programowanie bazodanowe, 57
 programowanie naukowe, 57
 programowanie numeryczne, 53, 57
 programowanie systemowe, 55
 programowanie webowe, 56
 programowanie zorientowane aspektowo, 1062

programowanie funkcyjne, 508
filter, 508
map, 507
reduce, 508, 509
programowanie zorientowane obiektowo, 49, 144, 633, 635, 680, 682, 757, 836
agregacja, 760
delegacja, 765
dziedziczenie, 651, 757
dziedziczenie wielokrotne, 775
fabryka obiektów, 785
hermetyzacja, 757, 836
klasy, 647, 703
kompozycja, 760
polimorfizm, 757, 758, 836
ponowne wykorzystanie kodu, 641, 836
przeciążanie, 758
przeciążanie operatorów, 727
spójność, 836
struktura, 836
utrzymywanie kodu, 836
związek „jest”, 759
związek „ma”, 760
projektowanie, 757, 788
funkcje, 491
moduły, 621
projektowanie wyjątków, 889
problemy, 897
zagnieżdżanie programów obsługi wyjątków, 889
property, 741, 956, 957, 1063
Propozycja ulepszenia Pythona, 58
protokół deskryptora, 956
protokół instrukcji class, 1071
protokół iteracyjny, 137, 383, 384, 393, 402
funkcje generatorów, 521
generatory, 521
listy składane, 390
wyrażenia generatorów, 524
protokół przedawnienia, 331
protokół właściwości, 957
protokół zarządzania kontekstem, 868
prototypowanie, 57
prototypowanie inkrementalne, 667
proxy, 997
proxy class, 765
prywatność zmiennych, 742
przebieg sterowania, 348, 856
przechodnie przeładowywanie modułów, 618
przechodzenie niewyczerpujące, 374
przechodzenie równoległe, 376
przechowywanie obiektów
 baza danych, 691, 692
 pickle, 277
 pliki, 276
przechwytywanie atrybutów, 955
przechwytywanie atrybutów wbudowanych
 operacji, 979
przechwytywanie dostępu do atrybutów, 961
przechwytywanie wbudowanych atrybutów, 685
przechwytywanie wbudowanych metod klas, 727
przechwytywanie wyjątków, 841, 844
 kategorie wyjątków, 852
 wbudowane wyjątki, 857
przeciążanie, 652
przeciążanie wywołań tworzących klasę, 1075, 1076
przeciążanie za pomocą sygnatur wywołań, 758
przeciążanie operatorów, 156, 282, 635, 654, 655, 657, 673, 674, 727, 970
 __bool__, 751
 __call__, 747
 __cmp__, 750
 __contains__, 737
 __del__, 752
 __eq__, 750
 __getattr__, 740
 __getitem__, 730, 731, 737
 __gt__, 750
 __iadd__, 746
 __iter__, 733, 734, 737
 __len__, 751
 __lt__, 750
 __ne__, 750
 __next__, 733
 __radd__, 745
 __repr__, 742
 __setattr__, 740
 __setitem__, 730, 731
 __str__, 742
destrukcja obiektu, 752
dodawanie w miejscu, 746
indeksowanie, 730
iteracja po indeksie, 731
iteratory zdefiniowane przez użytkownika, 734
metody, 728, 729
obiekty iteratorów, 733
operatory przemienne, 745
porównania, 750
test przynależności, 737
testy logiczne, 751
wycinanie, 730
przecinające się sekwencje, 431
przecinki, 269
przedawnienie, 331
przeglądanie kluczy słownika, 388
przekazywanie argumentów, 427, 465
 argumenty niezmienne, 466

- przekazywanie argumentów
argumenty zmienne, 466
przekazywanie dowolnej liczby argumentów, 471
przekazywanie przez wartość, 466
przekazywanie przez wskaźnik, 466
tryby dopasowania argumentów, 470
przekazywanie wyjątków, 864
przekierowanie dostępu do uszczegółowionych atrybutów, 956
przekierowanie strumieni, 89
strumień wyjściowy, 338
przeładowywianie kodu modułu, 96, 555, 580
dostosowanie większych systemów do własnych potrzeb, 583
przechodnie przeładowywianie modułów, 618
reload, 580, 581
przenośne API dla baz danych, 57
przenośność kodu, 50, 60
przesłanianie, 652
przestrzenie nazw, 98, 100, 437, 555, 576, 637, 704, 714, 715
__dict__, 718
deklaracja, 443
łącza, 720
moduły, 575
nazwy atrybutów, 715
pojedyncze nazwy, 715
przypisania zmiennej, 715
słowniki przestrzeni nazw, 718
zakres, 437
przesunięcie bitowe, 152
przetwarzanie
liczby, 168
pliki, 274
sekwencje, 367
spakowane dane binarne w plikach, 278
przypadki specjalne dla reguły o końcu wiersza, 308
przypisanie, 83, 292, 302, 317, 418, 649, 715
formy instrukcji przypisania, 318
krotki, 318
listy, 318
referencje, 317
rozszerzone rozpakowanie sekwencji, 318
zmienne, 317
przypisanie rozszerzone, 319, 326
współdzielone referencje, 328
przypisanie sekwencji, 318, 319
wzorce zaawansowane, 320
przypisanie z wieloma celami, 319, 325
współdzielone referencje, 326
pseudoprzywatne atrybuty klas, 767
PSF, 58
Psyco, 76
pth, 562
Public, 1037
punkty zaczepienia operacji importowania, 566
puste wiersze, 348, 417
PVM, 73
py2exe, 78, 905
pyc, 560, 566
PyChecker, 444, 902
PyDev, 107
PyDoc, 405, 410, 723, 902
graficzny interfejs użytkownika, 413
raporty HTML, 412
strony dokumentacji, 414
uruchamianie, 412
pygame, 58
PyGTK, 56, 78
PyInstaller, 78, 905
PyLint, 902
Pylons, 56, 697
pyo, 566
PyOpenGL, 58
PyPI, 58
PyPy, 77, 79
PyQt, 56, 697
Pyrex, 79
PyRo, 58
PySerial, 58, 281
PySol, 58
python, 72, 81
Python, 49
język zorientowany obiektowo, 59
licencja, 59
wsparcie techniczne, 58
zastosowanie, 53
Python 2.6, 30
Python 3.0, 30, 33
usunięte elementy języka, 33
Python Enhancement Proposal, 58
Python Manuals, 70
Python Server Pages, 56
Python Software Foundation, 58
Python Virtual Machine, 73
PythonCard, 56, 108
PYTHONPATH, 561, 562, 563, 586, 613
PythonWin, 108
PyUnit, 903

Q

- Qt, 56
queue, 446

R

raise, 302, 841, 845, 863, 884
from, 865
przekazywanie wyjątków, 864
sygnalizowanie warunków przez funkcje, 893
random, 123, 152, 170
random(), 170
range, 242, 373, 397, 417, 536
modyfikacja list, 375
wycinki, 374
re, 129, 198, 204, 292, 944
re.compile, 292
re.match, 944
read, 141, 272
readline, 272, 274, 276, 384
readlines, 372, 391, 519
reduce, 508, 509
refaktoryzacja, 672
referencje, 186, 187, 189, 283, 292, 317, 466, 467
referencje cykliczne, 189, 754
referencje współdzielone, 190, 326
is, 193
modyfikacje w miejscu, 191
przypisanie rozszerzone, 328
przypisanie z wieloma celami, 326
równość, 193
regular expressions, 198
reguła LEGB, 440
reguła LGB, 440
reguły indentacji, 349
reguły precedencji, 154
reguły składni Pythona, 348
reguły wyszukiwania modułów, 598
rekordy, 256, 665
rekurencja, 439, 493
reload, 97, 98, 100, 580, 581, 588, 624
testowanie interaktywne, 625
remove, 177, 247
repetition, 125
replace, 126, 214, 216, 217
repr, 152, 159, 182, 211
formaty wyświetlania, 159
reprezentacje łańcuchów znaków, 742
RESTART, 104
return, 302, 425, 427, 428, 430, 432, 469, 520, 548
reverse, 131, 246, 247
reversed, 247
ręczne przekierowanie strumienia wyjścia, 339
roboty, 58
round, 163, 169
rozbiжение zadania na funkcje, 491
rozgałęzienia kodu, 346

rozpakowywanie argumentów, 470, 476
rozszerszanie, 711
rozszerszanie metod, 676
rozszerszanie typów wbudowanych, 791
klasy podrzędne, 793
osadzanie, 792
rozszerszenia numeryczne, 182
rozszerszenia plików, 90
rozszerszona składnia list składanych, 392
rozszerszona składnia rozpakowania sekwencji, 322
for, 325
przypadki brzegowe, 324
rozszerszona składnia słowników składanych, 534
rozszerszona składnia zbiorów składanych, 534
rozszerszone przypisanie sekwencji w pętlach for
w Pythonie 3.0, 369
rozszerszone rozpakowanie sekwencji, 318
rozszerszone wycinki, 209
rozszerszony protokół funkcji generatorów, 523
rozwiązywanie konfliktów w zakresie nazw, 440
równość, 193, 285
równoważność instrukcji import oraz from, 573
rstrip, 127, 211, 219, 277, 391
Ruby, 64
Run Module, 105
runtime engine, 73
rzadkie struktury danych, 255

S

SAX, 948
ScientificPython, 58
SciPy, 58, 183
screen scraping, 698
seek, 141, 272, 280
sekwencje, 124, 197, 235
krotki, 139
listy, 130
operacje, 124
przypisanie, 319
rozszerszona składnia rozpakowania
sekwiencji, 322
sekwiencje niezmienne, 140
sekwiencje ucieczki, 200, 201, 203
surowe łańcuchy znaków, 203
Unicode, 928
sekwiencje wirtualne, 384
sekwiencje znaków specjalnych, 198
self, 144, 145, 330, 468, 492, 638, 639, 640, 648, 650,
660, 666, 672, 682, 706, 770
send, 523
serializacja, 278, 691, 693, 764, 947

sesja interaktywna, 83
eksperymentowanie, 83
instrukcje wielowierszowe, 86
testowanie, 84
wykorzystywanie sesji, 85
set, 142, 176, 178, 281, 291, 396
set comprehension, 179
shared references, 190
shell tools, 55
shelve, 278, 281, 691, 692, 693, 947
interaktywne badanie obiektów shelve, 694
przechowywanie obiektów w bazie danych, 692
uaktualnianie obiektów, 695
short-circuit, 354
silnik bazy danych SQL, 57
silnik wykonawczy, 73
sin, 168
singleton, 1021, 1022
SIP, 57
site-packages, 563
skaner plików, 371
składanie list, 242, 383
składnia dekoratora funkcji, 811
składnia dekoratorów, 820, 969
składnia indentacji, 306
składnia Pythona, 348
składowe, 650
skompilowane pliki źródłowe, 72
skrypty, 87
tworzenie, 87
skrypty CGI, 56, 697
skrypty internetowe, 56
skrypty wykonywalne Uniksa, 91
slice, 125, 207
sloty, 805
kod generyczny, 806
wiele slotów w klasach nadrzędnych, 807
słowa zarezerwowane, 329
słowniki, 133, 235, 236, 248, 533
błędy brakujących kluczy, 255
dict, 257
długość, 249
dostęp do elementów, 136, 248
for, 136
fromkeys, 258
get, 139
has_key, 139, 251, 263
implementacja, 249
in, 251, 263
indeksowanie, 134
interfejsy, 257
items, 252, 260
iteratory, 251, 388, 401
keys, 251, 260, 261
klasy, 660
klucz, 134, 248, 254
 kolejność elementów, 251
 kolejność kluczy, 251
len, 251
liczba elementów, 251
lista kluczy, 251
literały, 250
mieszanie, 251
modyfikacja w miejscu, 251
operacje, 134, 250, 254
pop, 253
porównywanie, 287
porównywanie rozmiarów słowników, 263
przypisanie nowego klucza, 252
Python 3.0, 258
referencja do nieistniejącego klucza, 138
rekordy, 256
rzadkie struktury danych, 255
słowniki bez wartości, 178
słowniki rozwijane, 396
sorted, 262
sortowanie kluczy, 136, 262
sprawdzanie istnienia klucza, 251
stosowanie, 254
symulowanie elastycznych list, 254
tabela języków programowania, 253
tablice asocjacyjne, 249
tablice wielowymiarowe, 255
testowanie, 138
tworzenie, 257
update, 252
usuwanie ostatniego elementu, 253
usuwanie wartości klucza, 253
values, 252, 260
wartość, 134
widoki słowników, 260
zagnieżdżanie, 134, 249
zip, 259, 378
słowniki przestrzeni nazw, 718
słowniki składane, 259, 378, 534
składnia rozszerzona, 534
SmallTalk, 64
SOAP, 56, 698
sort, 131, 136, 245, 246, 483
argumenty ze słowami kluczowymi, 245
sorted, 138, 246, 247, 262, 269, 287, 394, 395
sortowanie, 131
klucze słowników, 136, 262
listy, 245
Python 3.0, 246
spacje, 348

spakowane dane binarne, 278
spam, 63
sparse data structure, 255
specjalizacja, 652
 odziedziczone metody, 710
specjalne atrybuty klas, 686
split, 127, 141, 218, 277
spójność, 491, 492
sprawdzanie błędów, 848
sprawdzanie danych wejściowych, 312
sprawdzanie identyczności obiektów, 286
sprawdzanie poprawności argumentów funkcji, 1044
sprawdzanie poprawności atrybutów, 986
 __getattr__, 990
 __getattribute__, 991
deskryptory, 988
 właściwości, 986
sprawdzanie równości, 193, 286
sprawdzanie typów, 144, 1054
sprawdzanie zawartości łańcuchów znaków, 219
sprintf, 221
sprzęganie, 491, 492
SQL, 57, 368, 698
 SQLAlchemy, 57, 698
SQLite, 57, 698
SQLObject, 57, 698
sqrt, 168, 169
stack trace, 844
Stackless Python, 55, 75, 79
stałe, 121
stan, 460, 967
 atributy funkcji, 461
 klasy, 460
stan deskryptora, 967
stan instancji, 967
stan obiektu, 648
standardowa składnia wywołania funkcji, 335
standardowy strumień wyjścia błędów, 341
standardowy strumień wyjściowy, 334
standardy kodowania, 332
startswith, 219
state information, 144
staticmethod, 812, 815, 816, 817, 820
stdout, 334, 339, 343
step, 209
sterowanie przebiegiem wysokiego poziomu,
 841, 847
StopIteration, 521
stos, 247
stosowanie dowolnego dekoratora do metod, 1086
stosowanie zmiennych globalnych, 445
str, 152, 159, 171, 182, 199, 211, 275, 291, 912, 915,
 917, 920
 find(), 206
 formaty wyświetlania, 159
str.encode, 920
stream redirection, 89
stride, 209
string, 197, 220
strings, 124
strony internetowe, 697
struct, 912, 945
struktura programu, 301, 556
strukturny, 256
strukturny danych, 120
strumienie poleceń powłoki, 281
strumienie standardowe, 280
styl kodowania z języka C, 417
subprocess.Popen, 281
sum, 168
suma zbiorów, 153, 177
sumowanie, 493
surowe łańcuchy znaków, 198, 203
SWIG, 57
Swing, 56
Sybase, 57
sygnalizowanie warunków przez funkcje, 893
sygnatury wywołań, 758
symetryczna różnica zbiorów, 153
symulowanie elastycznych list, 254
symulowanie parametrów wyjścia, 469
SyntaxError, 85
sys, 194, 211, 280, 334
sys.argv, 211, 611
sys.exc_info, 877, 885, 896
sys.setdefaultencoding, 912
sys.getrefcount, 194
sys.modules, 447, 615
sys.path, 412, 564, 565, 594
sys.platform, 88
sys.stderr, 341
sys.stdin, 893
sys.stdout, 280, 334, 339, 340, 343
system HTMLGen, 56
SystemExit, 880
systemowy wiersz poleceń, 87
sztuczna inteligencja, 58
szybkość programowania, 51

Ś

ścieżka kwalifikacji, 578
ścieżka wyszukiwania modułów, 101, 560, 561, 567
 katalog główny programu, 562
 katalogi biblioteki standardowej, 562
 katalogi ścieżek plików .pth, 562

ścieżka wyszukiwania modułów
konfiguracja ścieżki wyszukiwania, 563
modyfikacja, 613
PYTHONPATH, 562, 563
sys.path, 564
wariacje ścieżki wyszukiwania modułów, 564
wybór pliku modułu, 565
ścieżki do katalogów, 90
ślad stosu, 844
śledzenie dekoratorów, 1085
śledzenie interfejsów obiektów, 1023
śledzenie metaklas, 1085
śledzenie wywołań, 1007

T

tabela języków programowania, 253
tablica skoków, 347, 503
tablice, 130
tablice asocjacyjne, 248, 249
tablice mieszające, 248
tablice wielowymiarowe, 255
Tcl, 63
techniki interfejsów klas, 711
techniki kodowania łańcuchów Unicode, 923
techniki tworzenia pętli, 372
tekst, 124
 ASCII, 199
 Unicode, 199, 911, 912
test przynależności, 737
testowanie, 667
 testowanie interaktywne, 625
 testowanie kodu wewnętrz tego samego
 procesu, 895
 testowanie programów w locie, 83, 84
testy jednostkowe, 609, 610
testy logiczne, 738, 751
testy prawdziwości, 353
threading, 446
time, 138, 536, 904
time.clock, 539
time.time, 539
timeit, 138, 543, 904
timer, 1021
timsort, 483
Tk GUI API, 55
tkinter, 55, 78, 103, 105, 603, 697, 785
 funkcje zwrotne, 506
Tkinter, 103, 506, 697
trace, 540
Translator Shedskin C++, 77
True, 143, 181, 193, 288, 289, 354, 442, 444
trunc, 163, 169

trwałość obiektów, 57, 142, 691, 764
try, 139, 302, 303, 313, 842, 851, 860, 897
 debugowanie, 895
try/except, 841
try/except/else, 851
try/except/finally, 859
 except, 861
 finally, 861
 zagnieżdżanie, 861
try/finally, 841, 846, 857
tryb przetwarzania pliku, 917
tryby dopasowania argumentów, 470
 *, 471, 476
 **, 471, 476
 apply, 478
 argumenty mogące być tylko słowami
 kluczowymi, 471
dopasowanie po nazwie argumentu, 471
łączenie słów kluczowych i wartości
 domyślnych, 475
nagłówek funkcji, 471
nieznana liczba argumentów, 471
reguły dotyczące kolejności, 472
rozpakowywanie argumentów, 476
składnia dopasowania, 471
słowa kluczowe, 471, 472, 474
uniwersalne stosowanie funkcji, 477
uogólnione funkcje działające na zbiorach, 485
wartości domyślne, 471, 472, 474
wywołanie funkcji, 471
zbieranie argumentów, 476
tuple, 139, 267, 291
TurboGears, 56, 697
tworzenie
 drzewo klas, 638
 drzewo przestrzeni nazw, 709
 funkcje, 426
 generator powtarzający, 525
 hierarchia klas, 651
 instancje, 666
 klasy, 647
 klasy mieszane, 776
 klasy podzielone, 675
 klasy wyjątków, 875
 konstruktory, 666
 listy składane, 390
 metaklasy, 1073
 metody, 669
 moduły, 569
 obiekty funkcji, 427, 428
 obiekty klas, 703
 pliki, 272
 skrypty, 87

słowniki, 257, 378
tablica skoków, 503
właściwości, 957
zbiory, 142, 176, 178
zbiory składane, 180
zmienne, 186
type, 143, 291, 772, 1068, 1070
TypeError, 857
typy danych, 121, 431
 Boolean, 143
 bytearray, 214, 912, 915, 932
 bytes, 915, 929
 Decimal, 170
 Fraction, 172
 hierarchie typów, 290
 krotki, 139, 267
 liczby, 122, 149
 listy, 130, 239
 listy składane, 132
 łańcuchy znaków, 124, 197
 obiekty typów, 291
 słowniki, 133, 248
 sprawdzanie typów, 144
 str, 915
 unicode, 915
 zagnieżdżanie, 131
 zbiory, 142
typy dynamiczne, 61, 122, 185, 194
typy niezmienne, 236
typy podstawowe, 121
typy powiązane, 187
typy powiązane z implementacją, 121
typy wbudowane, 120, 292
typy zmienne, 236

U

uaktualnianie obiektów w pliku shelve, 695
uchwyty do plików, 280
udostępnianie kodu, 905
udostępnianie metod wyjątków, 885
udostępnianie szczegółów wyjątku, 884
ukośniki prawe, 204
ukrywanie danych w modułach, 607
ułamki, 172
umieszczańanie obiektów w pamięci podręcznej, 194
unicode, 199, 912, 915
Unicode, 129, 141, 199, 275, 911
 dekodowanie tekstu spoza zakresu ASCII, 922
 kodowanie tekstu spoza zakresu ASCII, 921, 922
 kodowanie tekstu z zakresu ASCII, 921
konwersja kodowania, 925
Python 2.6, 925

sekwencje ucieczki, 928
techniki kodowania, 923
unikanie mieszania tabulatorów i spacji, 350
unikanie pętli w metodach przechwytyujących
 atrybuty, 972
unit testing, 609
unittest, 903
uniwersalne stosowanie funkcji, 477
Unladen Swallow, 79
uogólnianie kodu pod kątem słów kluczowych, 1047
uogólnienie kodu pod kątem deklaracji atrybutów
 jako publicznych, 1036
uogólnione funkcje działające na zbiorach, 485
update, 177, 252
upper, 127, 219
uruchamianie programu, 81
 >>>, 82
edytor tekstowy, 110
interaktywny wiersz poleceń, 81
kliknięcie ikony w systemie Windows, 93
ograniczenia klikania ikon, 95
skrypty wykonywalne Uniksa, 91
 wiersz poleceń, 88, 90
usługi sieciowe, 698
UTF-16, 917, 918
UTF-32, 917
UTF-8, 71, 914
utrzymywanie kodu, 677
użycie modułu, 570
użycie zmiennej, 186

V

values, 252, 260, 397
varargs, 471
Vim, 108
Visual Basic, 64
void *, 187

W

warianty modeli wykonywania, 74
wartości Boolean, 143, 181
 Python 2.6, 753
wartości domyślne, 1047
wartości zmiennoprzecinkowe o stałej precyzji, 170
wartość bezwzględna liczby, 168
wbudowane klasy wyjątków, 880
 domyślne wyświetlanie, 882
 kategorie wyjątków, 881
 stan, 882
wbudowane łańcuchy znaków dokumentacji, 409
wbudowane typy obiektów, 61

wczytywanie pliku, 272
web2py, 56, 697
WebWare, 56
while, 137, 302, 310, 311, 359, 373, 417
 break, 361, 363
 continue, 361, 363
 elipsy, 362
 else, 364
 natychmiastowe przejście na góre pętli, 363
ogólny format pętli, 361
pass, 361
pętle nieskończone, 360
techniki tworzenia pętli, 372
 wyjście z pętli, 363
whitespace, 127
widoki słowników, 260, 401
 operacje zbiorowe, 261
wiele iteracji po jednym obiekcie, 735
wiele slotów w klasach nadzędnych, 807
wielkość liter, 329
wieloprzetwarzanie, 55
wielowątkowość, 446
wiersz poleceń, 82, 90
 wykonywanie plików, 88
wiersze, 351
Windows, 89, 90
Wing IDE, 108
with, 172, 302, 303, 329, 841, 867, 869
właściwości, 956, 957
 dekoratory, 960
 deskryptory, 968
 składnia, 960
 sprawdzanie poprawności atrybutów, 986
 tworzenie, 957
właściwości klas, 809, 1063
włączanie opcji z przyszłych wersji Pythona, 608
wpisywanie poleceń, 85
 instrukcje wielowierszowe, 86
wrapper, 765
write, 140, 272, 274, 343
writelnes, 272
wskaźniki, 187
 NULL, 288
współbieżność, 523
współdzielone referencje, 326, 466
współdzielony stan, 459
współprogramy, 523
wstawianie automatycznie wykonywanego kodu, 996
wstawianie kodu wykonywanego w momencie dostępu do atrybutów, 956
wxPython, 56, 78, 697
wybór pliku modułu, 565
 punkty zaczepienia operacji importowania, 566
wycinanie, 730
wycinki, 125, 153, 207, 209, 374
 krok, 209
 krotki, 269
 listy, 243
 obiekty wycinków, 209
 przeciążanie operatorów, 730
 Python 2.6, 732
 range, 374
 trzeci limit, 209
 zastosowanie, 211
wydajność, 73
 iteratory, 535
wydajność programistów, 49, 51
wydobywanie atrybutów odziedziczonych, 779
wyjątki, 139, 312, 841, 843
 __del__, 754
 ArithmetricError, 881
 as, 841, 853, 864, 867
 assert, 841, 865
 AssertionError, 865
 BaseException, 874, 880
 części instrukcji try, 853
 dane, 884
 debugowanie, 895
 domyślne wyświetlanie, 882
 domyślny program obsługi wyjątków, 843
 działania końcowe, 842, 846, 857
 else, 854
 EOFError, 893
 except, 313, 853
 Exception, 845, 854, 876, 880
 finally, 846, 859
 hierarchie wyjątków, 877
 IndexError, 843, 855, 863
 informacje o stanie wyjątku, 875
 kategorie wyjątków, 875, 881
 klasy, 845
 klasy wyjątków, 873
 łańcuchy wyjątków, 865
 menedżery kontekstu, 867
 niezwykły przebieg sterowania, 843
 NotImplementedError, 712
 obiekty wyjątków, 873
 obsługa, 313
 obsługa błędów, 842
 obsługa przypadków specjalnych, 842
 OverflowError, 881
 powiadomienia o zdarzeniach, 842
 problemy projektowania wyjątków, 897
 program obsługi wyjątków, 842
 przechwytywanie, 841, 844

przechwytywanie kategorii wyjątków, 852, 876
przechwytywanie wbudowanych wyjątków, 857
przekazywanie wyjątków, 864
puste except, 854, 898
Python 3.0, 855
raise, 841, 845, 863
raise from, 865
StopIteration, 521
sygnalizowanie warunków przez funkcje, 893
sys.exc_info, 896
szczegóły wyjątku, 877
ślad stosu, 844
testowanie kodu wewnętrz tego samego
 procesu, 895
try, 313, 842, 897
try/else, 855
try/except, 841, 844
try/except/else, 851
try/except/finally, 859
try/finally, 841, 842, 846, 857
tworzenie klas wyjątków, 875
udostępnianie metod wyjątków, 885
udostępnianie szczególnów wyjątku, 884
wbudowane klasy wyjątków, 880
with, 841, 867
własne sposoby wyświetlania, 883
wyjątki oparte na klasach, 875
wyjątki oparte na łańcuchach znaków, 874
wyjątki zdefiniowane przez użytkownika, 845
wyspecjalizowane programy obsługi
 wyjątków, 852
zachowania, 884
zachowanie domyślne, 856
zagnieżdżanie programów obsługi wyjątków, 889
zagnieżdżanie przebiegu sterowania, 891
zagnieżdżanie składniowe, 891
zagnieżdżone instrukcje try/finally, 890
zamykanie plików, 894
zamykanie połączeń z serwerem, 894
zastosowanie, 841, 893
 zgłaszanie, 843, 845, 863
wyjście z pętli, 311, 363
wykonywanie interfejsów klas nadzędnych, 711
wykonywanie kodu bajtowego modułu, 561
wykonywanie obliczeń na danych użytkownika, 311
wykonywanie plików modułów, 101
wykonywanie programu, 69, 71, 74
 wiersz poleceń, 88
wyłapywanie ograniczeń, 866
wyniki, 418
wypisywanie atrybutów dla każdego obiektu
 w drzewie klas, 781
wypisywanie danych, 83

wyrażenia, 152, 157, 301, 332
wyrażenia formatujące, 221, 222, 229, 234
wyrażenia generatorów, 520, 524, 525, 529, 782
wyrażenia indeksujące, 124
wyrażenia lambda, 427, 440, 501
 bliskość kodu, 504
 ciało, 502
 domyślne wartości argumentów, 502
 funkcje zwrotne, 506
 tablica skoków, 503
 zaciemnianie kodu, 504
 zagnieżdżone wyrażenia lambda, 505
 zakres, 505
 zakres zagnieżdżony, 453
 zastosowanie, 503
wyrażenia regularne, 944
wyrażenia z wycinkiem, 374
wyrażenie trójargumentowe if/else, 355
WYSIWYG, 350
wyspecjalizowane programy obsługi wyjątków, 852
wyspecjalizowany kompilator JIT, 77
wyszukiwanie, 635
 łańcuchy znaków, 206
 sekwencje, 738
wyszukiwanie dziedziczenia atrybutów, 635
wyświetlanie wartości w sesji interaktywnej, 332
wyświetlanie zawartości pliku, 274
wywołania zwrotne, 506, 748, 775
wywołanie, 153, 215, 302, 332, 418
 funkcje, 334, 429, 430, 432, 439
 konstruktory klas nadzędnych, 708
 metody, 144, 215, 638, 706
wywołanie tworzące klasę, 1075
wywoływanie wyjątków, 863
względne importowanie pakietów, 593, 598
 bezwzględne ścieżki pakietów, 596
 Python 3.0, 596
 zastosowanie, 595
wzorce, 129
wzorce projektowe, 643

X

XML, 56, 58, 129, 205, 948
XML-RPC, 56, 698
xmlrpclib, 58
XPath, 949
Xquery, 949
xrange, 398
xreadlines, 372
XSLT, 949
xterm, 82

Y

yield, 153, 154, 302, 303, 329, 331, 402, 425, 427, 520, 521, 522
YouTube, 54

Z

zachowania domyślne, 648
zachowanie informacji o stanie, 1009
zachowanie stanu zakresu zawierającego, 451
zaciemnianie kodu, 504
zagnieżdżanie dekoratorów, 1004
zagnieżdżanie kodu, 314
zagnieżdżanie programów obsługi wyjątków, 889
 zagnieżdżanie przebiegu sterowania, 891
 zagnieżdżanie składniowe, 891
zagnieżdżanie przebiegu sterowania, 891
zagnieżdżanie przestrzeni nazw, 579
zagnieżdżanie składniowe, 891
zagnieżdżanie typów danych, 131
zagnieżdżanie zakresów, 454
zagnieżdżone bloki kodu, 304, 349
zagnieżdżone instrukcje try/finally, 890
zagnieżdżone pętle for, 370
zagnieżdżone wyrażenia lambda, 505
zakres, 331, 437, 441, 462, 576, 578
 zakres dynamiczny, 579
 zakres funkcji zawierającej, 1010, 1035
 zakres globalny, 438, 459
 zakres importów względnych, 597
 zakres leksykalny, 438, 579
 zakres lokalny, 438, 439
 zakres wbudowany, 442
 zakresy zawierające, 1010
zakres zagnieżdżony, 449, 827
 domyślne wartości argumentów, 451
 domyślne wartości argumentów w zmiennych pętli, 453
 dowolne zagnieżdżanie zakresów, 454
 funkcje fabryczne, 450
 wyrażenia lambda, 452
zachowanie stanu zakresu zawierającego, 451
zakres zagnieżdżony statycznie, 448
zamrożone pliki binarne, 78, 109, 560
zamrożone zbior, 179, 236
zamykanie plików, 272, 894
zamykanie połączeń z serwerem, 894
zapętlenia w metodach przechwytyujących atrybuty, 972
zapętlenie referencji, 754
zapisywanie plików, 140, 272
 pliki Unicode, 939
zapisywanie właściwości w kodzie, 960
zarządzane atrybuty, 955
zarządzanie atrybutami, 976
zarządzanie dostępem do atrybutów, 741, 955
zarządzanie funkcjami, 996
zarządzanieinstancjami, 996
zarządzanie klasami, 996
zarządzanie kontekstem, 868
zarządzanie wywołaniami, 996
zawieszanie stanu, 520
zbieranie argumentów, 476
zbior, 132, 133, 142, 152, 176, 236, 533
 add, 177
 dir, 177
 działania matematyczne, 176
 działania na zbiorach, 180
 iteracja, 177
 metody, 177
 obiekty widoku, 178
 ograniczenia niezmienności, 179
 Python 2.6, 176
 Python 3.0, 178
 remove, 177
 set, 142, 176, 178
 suma zbiorów, 177
 tworzenie, 142, 176, 178
 uogólnione funkcje działające na zbiorach, 485
 update, 177
 usuwanie elementów, 177
 wstawianie elementów, 177
 wyrażenia z operatorami, 176
 zamrożone zbior, 179
zbior rozwijane, 396
zbior składane, 179, 534
 składnia rozszerzona, 534
zbiór dokumentacji standardowej, 415
zbiór narzędzi Pythona, 901
zdarzenia, 506
ZeroDivisionError, 881
zgłaszaniewyjątków, 843, 845, 863
zgodność z wersjami, 670
zintegrowane środowisko programistyczne, 102, 107, 903
zip, 259, 376, 394, 397, 398, 527, 529
 generatory, 530
 tworzenie słowników, 378
zliczanie instancji
 metody klas, 818
 metody statyczne, 817
złamanie prywatności, 1039
złożenia, 133
złożenia list, 513
zmiana pozycji w pliku, 272

- zmienne, 98, 124, 157, 186, 187, 317, 427
kwalifikacja, 577
modyfikacja zmiennych pomiędzy plikami, 573
nazwy, 329
przestrzenie nazw, 715
tworzenie, 186
typ zmiennej, 186
użycie zmiennej, 186
zakres, 331
zmienne lokalne, 433, 438
zmienne pseudoprыватне, 767
znieksztalcanie nazw, 767
zmienne globalne, 438, 443, 459, 1010
metody dostępu, 447
minimalizacja stosowania, 445
zmienne modyfikowane w pętli, 548
zmienne nielokalne, 438, 456, 457, 1010
przypadki graniczne, 458
zastosowanie, 458
- zmienne odwzorowania, 249
znacznik kolejności bajtów, 917, 941
znieksztalcanie nazw zmiennych, 767
ZODB, 57, 698
Zope, 56, 697
zorientowanie obiektowe, 145
zorientowany obiektowo skryptowy język programowania, 51
związek „jest”, 759
związek „ma”, 760
zwracanie wyniku częściowego, 332
zwracanie wyniku z funkcji, 427

Ž

źródła dokumentacji Pythona, 405

O autorze

Mark Lutz jest znanym na całym świecie instruktorem Pythona, autorem najnowszych i najlepiej się sprzedających tekstów poświęconych temu językowi oraz jedną z najważniejszych postaci w środowisku Pythona.

Mark jest również autorem popularnych książek: *Programming Python* (O'Reilly) oraz *Python. Leksykon kieszonkowy* (wydanie polskie: Helion) — obie doczekały się już trzeciego bądź czwartego wydania. Używa i promuje Pythona od 1992 roku, zaczął pisać książki na temat tego języka w 1995 roku, szkolenia z Pythona prowadzi od 1997 roku i do początku roku 2009 prowadził ponad dwadzieścia pięć poświęconych temu językowi sesji treningowych dla ponad trzech i pół tysiąca osób. Jego książki poświęcone Pythonowi sprzedają się w nakładzie około dwustu pięćdziesięciu tysięcy egzemplarzy i zostały przetłumaczone na kilkanaście języków.

Lutz jest absolwentem informatyki na University of Wisconsin i w ciągu ostatnich dwudziestu pięciu lat jako programista pracował nad kompilatorami, narzędziami programistycznymi, aplikacjami skryptowymi oraz wybranymi systemami klient-serwer. Można się z nim skontaktować za pośrednictwem strony internetowej <http://www.rmi.net/~lutz>.

Kolofon

Zwierzę na okładce książki *Python. Wprowadzenie. Wydanie IV* to amerykański szczur drzewny z rodzaju *Neotoma*. Mieszka w różnych warunkach środowiskowych (przede wszystkim na obszarach skalistych, pustynnych lub w buszu) na większości terytorium Ameryki Północnej oraz Środkowej, zazwyczaj w pewnym oddaleniu od ludzi. Szczury te dobrze się wspinają, a gniazda zakładają w drzewach lub zaroślach na wysokości do sześciu metrów. Niektóre gatunki kopią podziemne nory lub mieszkają w skalnych szczelinach bądź zajmują nory opuszczone przez inne zwierzęta.

Te szaro-beżowe gryzonie średniej wielkości są prawdziwymi „chomikami” — znoszą do swoich siedzib dokładnie wszystko, obojętnie, czy jest im potrzebne, czy nie, i są szczególnymi wielbieliami świecących przedmiotów, takich jak puszki, szkło czy srebro.

Rysunek na okładce to dziewiętnastowieczny sztych z *Cuvier's Animals*.