Zaawansowany Python

Luciano Ramalho

przekład: Maria Chaniewska,
Jakub Niedźwiedź

APN Promise
Warszawa 2015

O'REILLY®

Zaawansowany Python

© 2015 APN PROMISE SA

Authorized translation of English edition of
Fluent Python
ISBN 978-1-491-94600-8
Copyright © 2015 Luciano Ramalho. All rights reserved.

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

APN PROMISE SA, biuro: ul. Kryniczna 2, 03-934 Warszawa
tel. +48 22 35 51 600, fax +48 22 35 51 699
e-mail: mspress@promise.pl

Wszystkie prawa zastrzeżone. Żadna część niniejszej książki nie może być powielana ani rozpowszechniana w jakiejkolwiek formie i w jakikolwiek sposób (elektroniczny, mechaniczny), włącznie z fotokopiowaniem, nagrywaniem na taśmy lub przy użyciu innych systemów bez pisemnej zgody wydawcy.

Logo O'Reilly jest zarejestrowanym znakiem towarowym O'Reilly Media, Inc. *Fluent Python*, ilustracja z okładki i powiązane elementy są znakami towarowymi O'Reilly Media, Inc.

Wszystkie inne nazwy handlowe i towarowe występujące w niniejszej publikacji mogą być znakami towarowymi zastrzeżonymi lub nazwami zastrzeżonymi odpowiednich firm odnośnych właścicieli.

Przykłady firm, produktów, osób i wydarzeń opisane w niniejszej książce są fikcyjne i nie odnoszą się do żadnych konkretnych firm, produktów, osób i wydarzeń. Ewentualne podobieństwo do jakiejkolwiek rzeczywistej firmy, organizacji, produktu, nazwy domeny, adresu poczty elektronicznej, logo, osoby, miejsca lub zdarzenia jest przypadkowe i niezamierzone.

APN PROMISE SA dołożyła wszelkich starań, aby zapewnić najwyższą jakość tej publikacji. Jednakże nikomu nie udziela się rękojmi ani gwarancji.
APN PROMISE SA nie jest w żadnym wypadku odpowiedzialna za jakiekolwiek szkody będące następstwem korzystania z informacji zawartych w niniejszej publikacji, nawet jeśli APN PROMISE została powiadomiona o możliwości wystąpienia szkód.

ISBN: 978-83-7541-171-3

Projekt okładki: Ellie Volkhausen
Ilustracje: Rebecca Demarest

Przekład: Maria Chaniewska, Jakub Niedźwiedź
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Para Marta, com todo o meu amor.

Spis treści

Przedmowa .... *xv*

Część I Prolog

1 Model danych Pythona 3

Pythoniczna talia kart 4
Sposoby używania metod specjalnych 8
  Emulacja typów liczbowych 9
  Reprezentacja tekstowa 11
  Operatory arytmetyczne 12
  Wartość Boolean typu niestandardowego 12
Przegląd metod specjalnych 13
Dlaczego len nie jest metodą 14
Podsumowanie rozdziału 15
Lektura uzupełniająca 16

Część II Struktury danych

2 Sekwencje i tablice 21

Przegląd wbudowanych sekwencji 22
Wyrażenia listowe i wyrażenia generatora 23
  Wyrażenia listowe a czytelność 23
  Wyrażenia listowe a funkcje map i filter 25
  Iloczyny kartezjańskie 26
  Wyrażenia generatora 27
Krotki nie są jedynie niezmiennymi listami 29
  Krotki jako rekordy 29
  Rozpakowywanie krotek 30
  Rozpakowywanie zagnieżdżonych krotek 32
  Krotki nazwane 33
  Krotki jako niezmienne listy 35
Wycinanie 36
  Dlaczego wycinki i zakresy wykluczają ostatni element 37
  Obiekty wycinków 37
  Wycinanie wielowymiarowe i wielokropki 39
  Przypisywanie do wycinków 40

Używanie + i * z sekwencjami 40
    Budowanie listy list 41
Przypisanie złożone w przypadku sekwencji 43
    Zagadkowe przypisywanie += 44
Metoda list.sort oraz wbudowana funkcja sorted 46
Zarządzanie sekwencjami uporządkowanymi przy użyciu bisect 48
    Wyszukiwanie za pomocą funkcji bisect 48
    Wstawianie za pomocą funkcji bisect.insort 51
Kiedy lista nie jest rozwiązaniem 52
    Tablice 52
    Widoki pamięci 56
    NumPy i SciPy 57
    Deques i inne kolejki 60
Podsumowanie rozdziału 64
Lektura uzupełniająca 65

3 Słowniki i zbiory 71

Ogólne typy odwzorowujące 72
Wyrażenia słownikowe 74
Przegląd powszechnych metod odwzorowań 75
    Obsługa brakujących kluczy za pomocą setdefault 77
Odwzorowania z elastycznym przeszukiwaniem kluczy 79
    defaultdict: inne podejście do brakujących kluczy 79
    Metoda __missing__ 81
Odmiany dict 84
Tworzenie klas podrzędnych klasy UserDict 85
Niezmienne odwzorowania 87
Teoria zbiorów 88
    Literały zbiorów 90
    Wyrażenia zbioru 91
    Operacje na zbiorach 92
Budowa wewnętrzna typów dict i set 96
    Eksperyment wydajnościowy 96
    Tablice mieszające w słownikach 98
    Praktyczne konsekwencje działania słownika dict 101
    Jak działają zbiory – konsekwencje praktyczne 104
Podsumowanie rozdziału 105
Lektura uzupełniająca 105

4 Tekst a bajty 109

Problemy ze znakami 110
Podstawy bajtów 111
    Struktury i widoki pamięci 114

Podstawowe kodery/dekodery....115
Zrozumienie problemów kodowania/dekodowania....117
Radzenie sobie z UnicodeEncodeError....117
Radzenie sobie z UnicodeDecodeError....119
Błąd SyntaxError podczas ładowania modułów z nieoczekiwanym kodowaniem....120
Jak wykryć kodowanie sekwencji bajtów....122
BOM: przydatny gremlin....122
Obsługa plików tekstowych....124
Domyślne kodowanie: dom wariatów....127
Normalizacje Unicode w celu rozsądniejszego porównywania....130
Sprowadzanie do jednego rejestru....133
Funkcje narzędziowe do dopasowywania normalizowanego tekstu....134
„Normalizcja ekstremalna”: usuwanie znaków diakrytycznych....135
Sortowanie tekstu Unicode....138
Sortowanie przy użyciu algorytmu porządku alfabetycznego Unicode....140
Baza danych Unicode....141
Dwutrybowe interfejsy API dla typów str i bytes....143
str a bytes w wyrażeniach regularnych....143
str a bytes w funkcjach modułu os....144
Podsumowanie rozdziału....147
Lektura uzupełniająca....148

Część III Funkcje jako obiekty

5 Funkcje pierwszej klasy....155
Traktowanie funkcji jako obiektu....156
Funkcje wyższego rzędu....157
Nowoczesne odpowiedniki funkcji map, filter i reduce....158
Funkcje anonimowe....160
Siedem odmian obiektów wywoływalnych....161
Definiowane przez użytkownika typy wywoływalne....162
Introspekcja funkcji....163
Od parametrów pozycyjnych do parametrów tylko słów kluczowych....165
Pozyskiwanie informacji o parametrach....167
Adnotacje do funkcji....172
Pakiety do programowania funkcyjnego....174
Moduł operator....174
Zamrażanie argumentów przy użyciu funkcji functools.partial....178
Podsumowanie rozdziału....180
Lektura uzupełniająca....181

6 Wzorce projektowe z funkcjami pierwszej klasy . . . 185
Studium przypadku: refaktoryzacja wzorca Strategia . . . 186
Klasyczny wzorzec Strategia . . . 186
Strategia zorientowana funkcyjnie . . . 190
Wybieranie najlepszej strategii: proste podejście . . . 193
Znajdowanie Strategii w module . . . 194
Polecenie . . . 196
Podsumowanie rozdziału . . . 197
Lektura uzupełniająca . . . 198

7 Dekoratory funkcji i domknięcia . . . 201
Dekoratory 101 . . . 202
Kiedy Python wykonuje dekoratory . . . 203
Wzorzec Strategia wzbogacony dekoratorem . . . 205
Reguły zasięgów zmiennych . . . 207
Domknięcia . . . 210
Deklaracja nonlocal . . . 213
Implementacja prostego dekoratora . . . 215
Sposób działania . . . 216
Dekoratory w bibliotece standardowej . . . 218
Memoizacja dzięki functools.lru_cache . . . 219
Funkcje generyczne z pojedynczym rozsyłaniem . . . 221
Zagnieżdżanie dekoratorów . . . 224
Dekoratory parametryzowane . . . 225
Parametryzowany dekorator rejestrujący . . . 226
Parametryzowany dekorator Clock . . . 228
Podsumowanie rozdziału . . . 230
Lektura uzupełniająca . . . 231

Część IV Idiomy zorientowane obiektowo

8 Odwołania do obiektów, zmienność i odzyskiwanie pamięci . . . 237
Zmienne nie są pudełkami . . . 238
Tożsamość, równość i aliasy . . . 240
Wybór między == a is . . . 241
Względna niezmienność krotek . . . 242
Kopie są domyślnie płytkie . . . 243
Głębokie i płytkie kopie arbitralnych obiektów . . . 246
Parametry funkcji jako odwołania . . . 247
Typy zmienne jako domyślne parametry: zły pomysł . . . 249
Programowanie obronne ze zmiennymi parametrami . . . 251

del i odzyskiwanie pamięci 253
Słabe odwołania 255
Skecz WeakValueDictionary 256
Ograniczenia słabych odwołań 258
Trikowe gry Pythona z niezmiennymi obiektami 259
Podsumowanie rozdziału 261
Lektura uzupełniająca 262

9 Obiekt pythonowy 267

Reprezentacje obiektów 268
Przypomnienie klasy Vector 268
Alternatywny konstruktor 271
classmethod a staticmethod 272
Formatowane wyświetlanie 274
Haszowalny obiekt Vector2d 277
Prywatne i „chronione" atrybuty w Pythonie 283
Oszczędzanie miejsca dzięki atrybutowi klasy __slots__ 285
Problemy z atrybutem __slots__ 288
Przesłanianie atrybutów klasy 288
Podsumowanie rozdziału 291
Lektura uzupełniająca 292

10 Kodowanie, haszowanie i wycinanie sekwencji 297

Vector: definiowany przez użytkownika typ sekwencyjny 298
Vector podejście nr 1: zgodność z Vector2d 298
Protokoły i kacze typowanie 301
Vector podejście nr 2: sekwencja z możliwością wycinania 302
Działanie wycinania 303
Metoda __getitem__ świadoma wycinania 305
Vector podejście nr 3: dynamiczny dostęp do atrybutów 307
Vector podejście nr 4: haszowanie i szybsze == 311
Vector podejście nr 5: formatowanie 316
Podsumowanie rozdziału 324
Lektura uzupełniająca 325

11 Interfejsy: od protokołów do abstrakcyjnych klas bazowych 331

Interfejsy i protokoły w kulturze języka Python 332
Python lubi sekwencje 334
Małpie łatanie w celu zaimplementowania protokołu w trakcie działania programu 336
Wodne ptactwo Alexa Martelli 338
Tworzenie podklasy z abstrakcyjnej klasy bazowej 344
Abstrakcyjne klasy bazowe w bibliotece standardowej 346

Abstrakcyjne klasy bazowe w collections.abc .346
Wieża liczbowa klas ABC .348
Definiowanie i wykorzystywanie abstrakcyjnej klasy bazowej .349
Szczegóły składni abstrakcyjnych klas bazowych .354
Tworzenie podklas dla abstrakcyjnej klasy bazowej Tombola .355
Wirtualna podklasa klasy Tombola .357
Jak testowano podklasy klasy Tombola .360
Użycie metody register w praktyce .363
Gęsi mogą zachowywać się jak kaczki .364
Podsumowanie rozdziału .365
Lektura uzupełniająca .368

12 Dziedziczenie: na dobre czy na złe .375

Tworzenie klas podrzędnych z typów wbudowanych jest zawiłe .376
Wielokrotne dziedziczenie i kolejność ustalania metod .379
Wielokrotne dziedziczenie w świecie rzeczywistym .384
Radzenie sobie z wielokrotnym dziedziczeniem .387
1. Rozróżniać dziedziczenie interfejsów od dziedziczenia implementacji .387
2. Tworzyć jawne interfejsy przy pomocy klas ABC .387
3. Korzystać z domieszek w celu ponownego wykorzystania kodu .387
4. Jawnie deklarować domieszki dzięki nazewnictwu .388
5. Klasa ABC może być też domieszką, ale nie na odwrót .388
6. Nie tworzyć podklasy dziedziczącej z więcej niż jednej klasy konkretnej .388
7. Dostarczać użytkownikom klasy łączone .389
8. „Preferować komponowanie obiektów przed dziedziczeniem klas” .389
Tkinter: dobry, zły i brzydki .390
Nowoczesny przykład: domieszki w ogólnych widokach Django .391
Podsumowanie rozdziału .394
Lektura uzupełniająca .395

13 Przeciążanie operatorów: rób to poprawnie .399

Podstawy przeciążania operatorów .400
Operatory unarne .400
Przeciążanie operatora + w celu zaimplementowania dodawania wektorów 403
Przeciążanie operatora * dla mnożenia wektora przez wartość skalarną .409
Bogate operatory porównania .413
Operatory rozszerzonego przypisania .418
Podsumowanie rozdziału .423
Lektura uzupełniająca .424

Część V Przepływ sterowania

14 Iterowalność, iteratory i generatory 431

Klasa Sentence – podejście nr 1: sekwencja słów 432
Dlaczego sekwencje są iterowalne: funkcja iter 434
Obiekty iterowalne a iteratory 436
Klasa Sentence – podejście nr 2: klasyczne wnętrze 440
Klasa Sentence jako iterator: zły pomysł 441
Klasa Sentence – podejście nr 3: funkcja generatora 442
Jak działa funkcja generatora 443
Klasa Sentence – podejście nr 4: leniwa implementacja 447
Klasa Sentence – podejście nr 5: wyrażenie generatora 448
Wyrażenia generatora: kiedy ich używać 450
Inny przykład: generator ciągu arytmetycznego 451
Ciąg arytmetyczny wykorzystujący itertools 453
Funkcje generatora w bibliotece standardowej 455
Nowa składnia w wersji Python 3.3: yield from 467
Funkcje redukujące obiekty iterowalne 468
Bliższe przyjrzenie się funkcji iter 470
Studium przypadku: generatory w narzędziu do konwersji baz danych 471
Generatory jako współprogramy 473
Podsumowanie rozdziału 474
Lektura uzupełniająca 474

15 Zarządzanie kontekstem i bloki else 481

Zrób to, potem tamto: bloki else poza instrukcją if 482
Zarządzanie kontekstem i bloki with 484
Narzędzia contextlib 489
Korzystanie z @contextmanager 489
Podsumowanie rozdziału 493
Lektura uzupełniająca 494

16 Współprogramy 497

Jak współprogramy wyewoluowały z generatorów 498
Podstawowe zachowanie generatora zastosowane jako współprogram 499
Przykład: współprogram obliczający średnią kroczącą 503
Dekoratory przygotowujące współprogram 504
Kończenie współprogramów i obsługa wyjątków 506
Zwracanie wartości ze współprogramu 510
Korzystanie z yield from 512
Znaczenie konstrukcji yield from 519
Przypadek użycia: współprogramy dla dyskretnego symulowania zdarzeń 525

Symulacje zdarzeń dyskretnych 525
Symulacja floty taksówek 526
Podsumowanie rozdziału 535
Lektura uzupełniająca 536

17 Współbieżność z futures 543
Przykład: pobieranie stron WWW na trzy sposoby .544
Skrypt pobierania sekwencyjnego 546
Pobieranie przy pomocy concurrent.futures 548
Gdzie są obiekty future? 549
Blokowanie wejścia/wyjścia a GIL 553
Uruchamianie procesów przy pomocy concurrent.futures 554
Eksperymentowanie z Executor.map 556
Pobierania flag z wyświetlaniem postępów i obsługą błędów 559
Obsługa błędów w przykładach flags2 564
Korzystanie z futures.as_completed 566
Alternatywy dla przetwarzania wielowątkowego 569
Podsumowanie rozdziału 570
Lektura uzupełniająca 571

18 Współbieżność z asyncio 577
Wątek kontra współprogram: porównanie 579
Klasa asyncio.Future: nieblokująca z założenia 585
Instrukcja yield from a obiekty future, zadania i współprogramy 586
Pobieranie obrazów przy pomocy asyncio i aiohttp 588
Bieganie w kółko wokół wywołań blokujących 593
Ulepszanie skryptu pobierającego obrazy wykorzystującego asyncio 595
Wykorzystanie asyncio.as_completed 596
Korzystanie z obiektu wykonawczego w celu uniknięcia
zablokowania pętli zdarzeń 601
Od procedur zwrotnych do obiektów future i współprogramów 603
Wykonywanie wielu żądań dla każdego pobierania 605
Pisanie serwerów wykorzystujących asyncio 608
Serwer TCP wykorzystujący asyncio 609
Serwer WWW wykorzystujący aiohttp 614
Inteligentniejsi klienci a lepsza współbieżność 617
Podsumowanie rozdziału 618
Lektura uzupełniająca 619

Część VI Metaprogramowanie

19 Atrybuty i właściwości dynamiczne 627

Przekształcanie danych przy pomocy atrybutów dynamicznych 628
Badanie danych przypominających JSON przy pomocy atrybutów dynamicznych 631
Problem z nieprawidłowymi nazwami atrybutów...................634
Elastyczne tworzenie obiektów przy pomocy __new__ 635
Restrukturyzacja źródła danych OSCON przy pomocy shelve.........637
Pobieranie połączonych rekordów przy pomocy właściwości..........641
Użycie właściwości do sprawdzania poprawności atrybutów 647
LineItem – podejście nr 1: klasa dla elementu zamówienia............647
LineItem – podejście nr 2: właściwość sprawdzająca swoją poprawność 648
Właściwe spojrzenie na właściwości...................................650
Właściwości przesłaniają atrybuty instancji..........................651
Dokumentacja właściwości...654
Kodowanie fabryki właściwości.......................................655
Obsługiwanie usuwania atrybutów.....................................658
Podstawowe atrybuty i funkcje obsługujące atrybuty 659
Atrybuty specjalne, które wpływają na obsługę atrybutów 659
Funkcje wbudowane do obsługi atrybutów 660
Metody specjalne do obsługi atrybutów.............................661
Podsumowanie rozdziału 662
Lektura uzupełniająca..663

20 Deskryptory atrybutów 669

Przykład deskryptora: sprawdzanie poprawności atrybutu................669
LineItem podejście nr 3: prosty deskryptor 670
LineItem podejście nr 4: automatyczne nazwy atrybutów przechowywania 675
LineItem podejście nr 5: nowy typ deskryptora 681
Deskryptory przesłaniające a nieprzesłaniające 684
Deskryptor przesłaniający 686
Deskryptor przesłaniający bez __get__ 687
Deskryptor nieprzesłaniający.......................................688
Nadpisywanie deskryptora w klasie 689
Metody są deskryptorami...690
Wskazówki dotyczące użycia deskryptorów 693
Dokumentacja docstring deskryptora i przesłanianie usuwania 694
Podsumowanie rozdziału 695
Lektura uzupełniająca..696

21 Metaprogramowanie klas 699
Fabryka klas 700
Dekorator klasy służący do dostosowywania dekryptorów. 703
Co dzieje się kiedy: czas importu a czas działania 706
Ćwiczenia dotyczące czasu przetwarzania 707
Metaklasy 101. 710
Ćwiczenie dotyczące czasu przetwarzania metaklasy. 713
Metaklasa do dostosowywania deskryptorów. 717
Metoda specjalna __prepare__ metaklasy 719
Klasy jako obiekty 721
Podsumowanie rozdziału 722
Lektura uzupełniająca. 723

Posłowie. *727*

Dodatek: Skrypty pomocnicze *731*

Żargon społeczności Pythona. *759*

Indeks *769*

O autorze. *788*

Przedmowa

> Plan jest taki: gdy ktoś używa funkcjonalności, której nie rozumiesz, po prostu go zastrzel. Jest to łatwiejsze niż uczenie się czegoś nowego, a wkrótce jedyni żyjący programiści będą pisali w łatwym do zrozumienia, wąskim podzbiorze języka Python 0.9.6 ;-)
>
> *– Tim Peters*
> *Legendarny deweloper Pythona i autor The Zen of Python*

„Python jest łatwym do nauczenia, potężnym językiem programowania". To są pierwsze słowa w oficjalnym samouczku Python Tutorial (*https://docs.python.org/3/tutorial/*). To prawda, ale jest pewna pułapka: ponieważ ten język jest łatwy do nauczenia i zastosowania, wielu praktykujących programistów Pythona korzysta tylko z ułamka jego potężnych funkcjonalności.

Doświadczony programista może zacząć pisać użyteczny kod Pythona w ciągu paru godzin. W miarę jak pierwsze produktywne godziny zmieniają się w tygodnie i miesiące, wielu deweloperów nadal programuje w Pythonie z silnymi naleciałościami z języków, które znali wcześniej. Nawet osoby, dla których jest to pierwszy język programowania, często poznają go z materiałów szkoleniowych ostrożnie pomijających specyficzne funkcjonalności.

Jako nauczyciel przedstawiający Pythona programistom doświadczonym w innych językach dostrzegam inny problem, który ta książka próbuje rozwiązać: tęsknimy jedynie za tym, co już znamy. Kierując się doświadczeniem z innych języków, każdy może zgadnąć, że Python obsługuje wyrażenia regularne, i poszukać dokumentacji na ten temat. Ale jeśli ktoś nigdy nie widział wcześniej deskryptorów ani rozpakowywania krotek, prawdopodobnie nie będzie się zastanawiać nad ich użyciem. Zatem może pomijać korzystanie z tych funkcjonalności tylko dlatego, że są specyficzne dla Pythona.

Ta książka nie jest wyczerpującym kompendium od A do Z dotyczącym Pythona. Skupia się na funkcjonalnościach języka, które albo są unikalne dla Pythona, albo nie są obecne w wielu innych popularnych językach. Jej zakres obejmuje rdzeń języka i tylko

niektóre jego biblioteki. Rzadko będę pisać o pakietach, które nie są w bibliotece standardowej, chociaż indeks pakietów Pythona obejmuje obecnie ponad 60 000 bibliotek, a wiele z nich jest niewiarygodnie przydatnych.

Dla kogo jest ta książka

Ta książka została napisana dla praktykujących programistów Pythona, który chcą osiągnąć biegłą znajomość wersji Python 3. Jeśli znasz wersję Python 2, ale chcesz przejść do wersji Python 3.4 lub nowszej, to doskonale. Gdy to piszę, większość profesjonalnych programistów Pythona używa wersji Python 2, więc zatroszczyłem się specjalnie, aby podkreślić funkcjonalności wersji Python 3, które mogą być nowe dla tych odbiorców.

Jednak *Zaawansowany Python* dotyczy jak najlepszego wykorzystania wersji Python 3.4 i nie omawiałem poprawek koniecznych do zastosowania tego kodu w poprzednich wersjach. Większość przykładów powinna działać w wersji Python 2.7 z niewielkimi poprawkami lub od razu, ale w niektórych przypadkach przeniesienie na starszą wersję wymagałoby znaczących zmian.

Powiedziawszy to, wierzę, że ta książka może być przydatna, nawet jeśli musisz nadal korzystać z wersji Python 2.7, ponieważ podstawowe koncepcje są nadal takie same. Python 3 nie jest nowym językiem, a większość różnic można poznać w jedno popołudnie. *What's New in Python 3.0* (*https://docs.python.org/3.0/whatsnew/3.0.html*) jest dobrym punktem wyjścia. Oczywiście było wiele zmian od czasu wydania wersji Python 3.0 w roku 2009, ale żadne z nich nie były tak ważne, jak te w wersji 3.0.

Jeśli nie wiesz, czy znasz Pythona wystarczająco, aby skorzystać z tej książki, przejrzyj tematy oficjalnego samouczka *Python Tutorial*. Tematy opisane w samouczku nie zostaną tu wyjaśnione, poza pewnymi funkcjonalnościami, które są nowością w wersji Python 3.

Dla kogo nie jest ta książka

Jeśli po prostu uczysz się Pythona, ta książka będzie zbyt trudna. Powiem więcej, jeśli przeczytasz ją za wcześnie podczas swojej przygody z Pythonem, możesz mieć wrażenie, że każdy skrypt Pythona powinien wykorzystywać metody specjalne i triki metaprogramowania. Przedwczesna abstrakcja jest równie zła, jak przedwczesna optymalizacja.

Organizacja książki

Docelowi odbiorcy tej książki nie powinni mieć problemu z przeskoczeniem bezpośrednio do dowolnego rozdziału w tej książce. Jednak każda z sześciu części tworzy samodzielną książkę w ramach tej książki. Założyłem, że rozdziały w każdej części będą czytane kolejno.

Próbowałem podkreślić używanie dostępnych rozwiązań przed omawianiem, jak zbudować własne. Na przykład w rozdział 3 w części II dotyczy typów sekwencji, które są gotowe do użycia, łącznie z tymi, którym nie poświęca się zbyt wiele uwagi, takim jak `collections.deque`. Budowanie definiowanych przez użytkownika sekwencji jest opisane dopiero w części IV, gdzie widzimy także, jak wykorzystać abstrakcyjne klasy bazowe (ABC) z modułu `collections.abc`. Tworzenie własnych klas ABC jest omówione jeszcze dalej w części IV, ponieważ uważam, że jest ważne, aby swobodnie korzystać z klas ABC, zanim będzie się pisać własne.

To podejście ma parę zalet. Po pierwsze znajomość tego, co jest gotowe do użycia pozwala uchronić nas przed ponownym wynajdowaniem koła. Używamy istniejących klas kolekcji częściej niż implementujemy własne i możemy poświęcić więcej uwagi zaawansowanemu użyciu dostępnych narzędzi dzięki odroczeniu omawiania sposobów tworzenia własnych. Również jest bardziej prawdopodobne, że będziemy dziedziczyć z istniejących klas ABC, niż tworzyć własne od zera. W końcu uważam, że łatwiej jest zrozumieć abstrakcje po zobaczeniu ich w akcji.

Wadą tej strategii są dalsze odwołania rozsiane po rozdziałach. Mam nadzieję, że będzie Ci łatwiej je tolerować teraz, gdy wiesz, dlaczego zdecydowałem się na taki układ książki.

Oto parę głównych tematów w każdej części tej książki:

Część I

Pojedynczy rozdział na temat modelu danych Pythona wyjaśniający, że metody specjalne (np. `__repr__`) są kluczowe dla spójnego działania obiektów wszystkich typów – w języku, który jest ceniony za swoją spójność. Zrozumienie różnych aspektów modelu danych jest przeważającym tematem dalszej treści tej książki, ale rozdział 1 zapewnia ogólny przegląd na wysokim poziomie.

Część II

Rozdziały w tej części dotyczą użycia typów kolekcji: sekwencji, odwzorowań i zbiorów, a także rozdziału między `str` a `bytes` – przyczyny radości dla użytkowników wersji Python 3 i dużego cierpienia dla użytkowników wersji Python 2, którzy nie przenieśli jeszcze swoich baz kodu. Głównymi celami jest przypomnienie dostępnych rozwiązań i wyjaśnienie ich działania, które jest czasami zaskakujące, jak niespostrzegalna zmiana kolejności kluczy `dict` lub zastrzeżenia dotyczące zależności sortowania łańcuchów Unicode od ustawień lokalnych. Opisy są czasami rozległe i na wysokim poziomie (np. podczas prezentacji wielu odmian sekwencji i odwzorowań), a czasami głębokie (np. podczas rozważania tablic mieszających leżących u podstaw typów `dict` i `set`).

Część III

Zawiera omówienie funkcji jako obiektów pierwszej klasy w języku: co to oznacza, jak wpływa na niektóre popularne wzorce projektowe i jak implementować dekoratory funkcji przy wykorzystaniu domknięć. Opisana jest tutaj także ogólna

koncepcja obiektów wywoływalnych w Pythonie, atrybutów funkcji, introspekcji, adnotacji parametrów oraz nowa deklaracja `nonlocal` w wersji Python 3.

Część IV
: Teraz skupimy się na budowaniu klas. W części II deklaracja `class` pojawiła się w kilku przykładach. Część IV prezentuje wiele klas. Podobnie jak dowolny język zorientowany obiektowo (OO), Python ma szczególny zestaw funkcjonalności, które mogą, ale nie muszą być obecne w języku, w którym uczyliśmy się programowania opartego na klasach. Kolejne rozdziały wyjaśniają, jak działają odwołania, co oznacza naprawdę zmienność, jaki jest cykl życia instancji, jak budować własne kolekcje i klasy ABC, jak radzić sobie z wielokrotnym dziedziczeniem i jak implementować przeciążanie operatorów – kiedy to ma sens.

Część V
: W tej części opisane są konstrukcje językowe i biblioteki, które wykraczają poza sekwencyjny przepływ sterowania za pomocą instrukcji warunków, pętli i podprogramów. Zaczynamy od generatorów, następnie zajmujemy się menedżerami kontekstu i współprogramami, w tym trudną, ale potężną nową składnią `yield from`. Część V kończy się wprowadzeniem na wysokim poziomie do nowoczesnej współbieżności w Pythonie przy użyciu `collections.futures` (z wewnętrznym wykorzystaniem wątków i procesów wspomaganych przez obiekty future) i wykonywanie zorientowanych na zdarzenia operacji I/O za pomocą `asyncio` (wykorzystujące obiekty future na szczycie współprogramów i `yield from`).

Część VI
: Ta część zaczyna się od przeglądu technik do budowania klas z atrybutami tworzonymi dynamicznie do obsługi danych semistrukturalnych, takich jak zestawy danych JSON. Dalej zajęliśmy się znajomym mechanizmem właściwości, przed zagłębieniem się w to, jak działa dostęp do obiektów atrybutów na niższym poziomie w Pythonie przy użyciu deskryptorów. Wyjaśniam także związek między funkcjami, metodami i deskryptorami. W całej części VI implementacja krok po kroku biblioteki walidacji pól odkrywa subtelne problemy, które prowadzą do użycia w ostatnim rozdziale zaawansowanych narzędzi: dekoratorów klas i metaklas.

Podejście praktyczne

Często będziemy używać interaktywnej konsoli Pythona do badania języka i bibliotek. Uważam, że jest ważne, aby podkreślić siłę tego narzędzia do nauki, szczególnie dla Czytelników, którzy mieli więcej doświadczenia ze statycznymi, kompilowanymi językami, które nie dostarczają mechanizmu REPL (read-eval-print#loop).

Jeden ze standardowych pakietów testowych Pythona, `doctest`, działa symulując sesje konsoli i weryfikując, że wyrażenia są przetwarzane na pokazane odpowiedzi. Używałem modułu `doctest` do testowania większości kodu w tej książce, w tym listingów konsoli.

Nie musisz używać modułu `doctest`, ani nawet o nim wiedzieć, aby być na bieżąco: główną funkcjonalnością testów doctest jest to, że wyglądają jak transkrypcje interaktywnych sesji konsoli, więc z łatwością możesz wypróbować demonstrację samodzielnie.

Czasami będę wyjaśniać, co chcemy osiągnąć, pokazując test doctest przed kodem, który pozwala na jego działanie. Ustalenie z góry, co ma być zrobione, przed zastanowieniem się, jak to zrobić, pomaga skoncentrować się podczas kodowania. Zaczynanie od pisania testów jest podstawą techniki programowania opartego na testach, czyli TDD (test driven development). Uważam to również za pomocne podczas nauczania. Jeśli nie znasz modułu `doctest`, spójrz na jego dokumentację (*https://docs.python.org/3/library/doctest.html*) oraz repozytorium kodu źródłowego tej książki (*https://github.com/fluentpython/example-code*). Zobaczysz, że możesz zweryfikować poprawność większości kodu w tej książce, wpisując `python3 -m doctest example_script.py` w powłoce poleceń swojego systemu operacyjnego.

Sprzęt używany do pomiarów czasu

Ta książka zawiera parę prostych benchmarków i pomiarów czasu. Te testy zostały wykonane na jednym z dwóch laptopów używanych do pisania tej książki: 2011 MacBook Pro 13" z procesorem 2.7 GHz Intel Core i7 CPU, 8GB pamięci RAM oraz tradycyjnym dyskiem twardym, a także 2014 MacBook Air 13" z procesorem 1.4 GHz Intel Core i5 CPU, 4GB pamięci RAM i dyskiem SSD. MacBook Air ma wolniejszy procesor i mniej pamięci RAM, ale jego pamięć RAM jest szybsza (1600 zamiast 1333 MHz), a dysk SSD jest znacznie szybszy niż dysk HD. W codziennym użyciu nie mogę stwierdzić, który komputer jest szybszy.

Pogadanki: moja osobista perspektywa

Używam i nauczam Pythona oraz dyskutuję na jego temat od roku 1998 i cieszy mnie badanie i porównywanie języków programowania, ich projektów i teorii, która za nimi stoi. Na końcu pewnych rozdziałów dodałem ramki „Pogadanka" z moimi własnymi spostrzeżeniami dotyczącymi Pythona i innych języków. Możesz swobodnie pominąć te ramki, jeśli Cię nie interesują. Ich zawartość jest całkowicie opcjonalna.

Żargon społeczności Pythona

Chciałem, aby była to książka nie tylko o Pythonie, ale także o kulturze wokół niego. Przez ponad 20 lat społeczność Pythona wytworzyła własny szczególny dialekt i akronimy. Zamieszony na końcu tej książki rozdział „Żargon społeczności Pythona" zawiera listę terminów, które mają specjalne znaczenie wśród Pythonistów.

Użyte wersje Pythona

Testowałem cały kod w tej książce przy użyciu Python 3.4 – czyli CPython 3.4, najbardziej popularnej implementacji Pythona napisanej w języku C. Jest tylko jeden wyjątek: ramka „Nowy operator infiksowy @ w wersji Python 3.5" przedstawia operator `@`, który jest obsługiwany tylko w wersji Python 3.5.

Prawie cały kod w tej książce powinien działać z dowolnym interpreterem kompatybilnym z wersją Python 3.x, w tym PyPy3 2.4.0, który jest kompatybilny z wersją Python 3.2.5. Wartymi zauważenia wyjątkami są przykłady korzystające z `yield from` i `asyncio`, które są dostępne tylko w wersji Python 3.3 lub nowszych.

Większość kodu powinna działać także w wersji Python 2.7 z ewentualnymi drobnymi zmianami. Nie będą działać w niej przykłady związane z Unicode zawarte w rozdziale 4. Ponadto wcześniej wymienione wyjątki nie będą działać w wersjach Python 3 wcześniejszych niż 3.3.

Konwencje użyte w tej książce

W tej książce używane są następujące konwencje typograficzne:

Kursywa
: Wskazuje nowe terminy, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

`Stała szerokość`
: Służy do wydruków programów, a także wewnątrz akapitów do odwołań do elementów programu, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

Zauważ, że gdy podział wiersza występuje w terminie o `stałej_szerokości` nie jest dodawany dywiz – mógłby zostać źle zrozumiany jako część terminu.

`Stała szerokość i pogrubienie`
: Pokazuje polecenia lub inny tekst, który powinien być wpisany dokładnie tak przez użytkownika.

`Stała szerokość i kursywa`
: Pokazuje tekst, który powinien być zastąpiony wartościami podanymi przez użytkownika lub wyznaczonymi przez kontekst.

Ten element oznacza wskazówkę lub sugestię.

Ten element oznacza uwagę ogólną.

Ten element wskazuje ostrzeżenie lub przestrogę.

Korzystanie z przykładów kodu

Wszystkie skrypty i większość fragmentów kodu, które występują w tej książce, są dostępne w repozytorium kodu *tej książki* (*https://github.com/fluentpython/example-code*) na GitHub.

Cenimy sobie, ale nie wymagamy, umieszczenia następujących informacji: tytułu, autora, wydawcy i ISBN. Na przykład: „*Fluent Python* by Luciano Ramalho (O'Reilly). Copyright 2015 Luciano Ramalho, 978-1-491-94600-8."

Jak się z nami kontaktować

Istnieje strona internetowa dotycząca tej książki, gdzie znajduje się errata, przykłady i inne dodatkowe informacje. Jej adres to *http://bit.ly/fluent-python*.

Komentarze i pytania techniczne dotyczące książki można wysyłać na adres bookquestions@oreilly.com.

Więcej informacji o naszych książkach, kursach, konferencjach i wiadomościach, zobacz na naszej stronie pod adresem *http://www.oreilly.com*.

Znajdź nas na Facebooku: *http://facebook.com/oreilly*
Śledź nas na Twitterze: *http://twitter.com/oreillymedia*
Oglądaj nas na YouTube: *http://www.youtube.com/oreillymedia*

Podziękowania

Josef Hartwig zaprojektował zestaw szachów Bauhaus, który jest przykładem wspaniałego projektu: piękny, prosty i czysty. Guido van Rossum, syn architekta i brat mistrza projektowania czcionek, zaprojektował cudowny język. Uwielbiam uczyć Pythona, ponieważ jest piękny, prosty i czysty.

Alex Martelli i Anna Ravenscroft byli pierwszymi osobami, które zobaczyły konspekt tej książki i zachęciły mnie do wysłania do wydawnictwa O'Reilly w celu publikacji. Ich książki nauczyły mnie idiomatycznego Pythona i są modelem przejrzystości, dokładności

i głębokości w pisaniu technicznym. Ponad 5 000 wpisów Alexa na Stack Overflow (*http://stackoverflow.com/users/95810/alex-martelli*) jest źródłem spojrzeń na język i jego właściwe użycie.

Martelli i Ravenscroft, a także Lennart Regebro i Leonardo Rochael byli ponadto recenzentami technicznymi tej książki. Każdy z tego wyróżniającego się zespołu recenzentów technicznych ma przynajmniej 15 lat doświadczenia w Pythonie, z ogromnym wkładem w wiele ważnych projektów Pythona w bliskim kontakcie z innymi deweloperami ze społeczności. Razem wysłali mi setki poprawek, sugestii, pytań i opinii, dodając dużo wartości do książki. Victor Stinner uprzejmie zrecenzował rozdział 18, wnosząc swoją wiedzę jako zarządcę `asyncio` do zespołu recenzentów technicznych. Był to duży przywilej i przyjemność współpracować z nimi przez te ostatnie miesiące.

Redaktorka Meghan Blanchette była wyróżniającym się mentorem, pomagając mi poprawić organizację i przepływ pracy nad książką, pokazując mi, co było nudne i powstrzymując mnie przed dalszymi opóźnieniami. Brian MacDonald edytował rozdziały w części III , gdy Meghan była niedostępna. Cieszyłem się pracą z nimi oraz ze wszystkimi, z którymi kontaktowałem się w wydawnictwie O'Reilly, w tym z zespołem twórców i pomocy technicznej Atlas (Atlas to platforma do publikowania książek wydawnictwa O'Reilly, której używałem szczęśliwie do pisania tej książki).

Mario Domenech Goulart dostarczył wielu szczegółowych sugestii zaczynając od pierwszego wydania Early Release. Otrzymałem także wartościowe opinie od następujących osób: Dave Pawson, Elias Dorneles, Leonardo Alexandre Ferreira Leite, Bruce Eckel, J. S. Bueno, Rafael Gonçalves, Alex Chiaranda, Guto Maia, Lucas Vido i Lucas Brunialti.

Przez lata wiele osób nakłaniało mnie, abym został autorem, a najbardziej przekonującymi byli Rubens Prates, Aurelio Jargas, Rudá Moura i Rubens Altimari. Mauricio Bussab otworzył dla mnie wiele drzwi, umożliwiając moją pierwszą prawdziwą próbę pisania książki. Renzo Nuccitelli wspierał ten projekt pisarski przez cały czas, chociaż to oznaczało opóźnienie naszego partnerstwa w *python.pro.br*.

Cudowna brazylijska społeczność Pythona jest pełna wiedzy, życzliwości i humoru. Grupa Python Brasil (*https://groups.google.com/group/python-brasil*) liczy tysiące osób, a nasze krajowe konferencje przyciągają ich setki, ale najbardziej wpływowymi Pythonistami na mojej drodze byli Leonardo Rochael, Adriano Petrich, Daniel Vainsencher, Rodrigo RBP Pimentel, Bruno Gola, Leonardo Santagada, Jean Ferri, Rodrigo Senra, J. S. Bueno, David Kwast, Luiz Irber, Osvaldo Santana, Fernando Masanori, Henrique Bastos, Gustavo Niemayer, Pedro Werneck, Gustavo Barbieri, Lalo Martins, Danilo Bellini i Pedro Kroger.

Dorneles Tremea był wspaniałym przyjacielem (niewiarygodnie życzliwie dzielącym się czasem i wiedzą), niesamowitym hakerem oraz najbardziej inspirującym liderem stowarzyszenia Brazilian Python Association. Odszedł zbyt wcześnie.

Przez lata moi studenci nauczyli mnie wiele przez swoje pytania, spostrzeżenia, opinie i kreatywne rozwiązania problemów. Érico Andrei i Simples Consultoria sprawili, że po raz pierwszy mogłem skupić się na byciu nauczycielem Pythona.

Martijn Faassen był moim mentorem grokowania i podzielił się ze mną bezcennymi spojrzeniami na temat Pythona i neandertalczyków. Jego praca oraz praca następujących osób: Paul Everitt, Chris McDonough, Tres Seaver, Jim Fulton, Shane Hathaway, Lennart Regebro, Alan Runyan, Alexander Limi, Martijn Pieters, Godefroid Chapelle, a także innych z planet Zope, Plone i Pyramid była decydująca dla mojej kariery. Dzięki Zope i surfowaniu na pierwszej fali webowej, byłem w stanie zacząć zarabiać na życie za pomocą Pythona w roku 1998. José Octavio Castro Neves był moim partnerem w pierwszej skupionej na Pythonie firmie programistycznej w Brazylii.

Mam zbyt wiele guru w szerokiej społeczności Pythona, aby wymienić ich wszystkich, ale poza tymi wcześniej wymienionymi, jestem wdzięczy następującym osobom: Steve Holden, Raymond Hettinger, A.M. Kuchling, David Beazley, Fredrik Lundh, Doug Hellmann, Nick Coghlan, Mark Pilgrim, Martijn Pieters, Bruce Eckel, Michele Simionato, Wesley Chun, Brandon Craig Rhodes, Philip Guo, Daniel Greenfeld, Audrey Roy i Brett Slatkin za nauczenie mnie nowych i lepszych sposobów uczenia Pythona.

Większość z tych stron została napisana w moim biurze domowym i w dwóch laboratoriach: CoffeeLab i Garoa Hacker Clube. *CoffeeLab* (*http://coffeelab.com.br/*) to siedziba kawiarnianych geeków w Vila Madalena, São Paulo, Brazil. *Garoa Hacker Clube* (*https://garoa.net.br/*) to klub hackerspace otwarty dla wszystkich: laboratorium społecznościowe, gdzie każdy może swobodnie wypróbować nowe pomysły.

Społeczność Garoa dostarczyła inspiracji, infrastruktury i luzu. Myślę, że Aleph cieszyłby się z tej książki.

Moja matka, Maria Lucia, i mój ojciec, Jairo, zawsze wspierali mnie na każdej drodze. Chciałbym, aby ojciec był tutaj i zobaczył tę książkę. Cieszę się, że mogę ją pokazać matce.

Moja żona, Marta Mello, trwała przy mnie przez 15 miesięcy, kiedy nieustannie pracowałem, ale nadal wspierała i podtrzymywała mnie w tych krytycznych momentach projektu, gdy chciałem uciec z tego maratonu.

Dziękuję Wam wszystkim za wszystko.

Część I

Prolog

Rozdział 1

Model danych Pythona

> Poczucie estetyki Guido dotyczące projektu języka jest zdumiewające. Spotkałem wielu dobrych projektantów umiejących tworzyć teoretycznie piękne języki programowania, z których jednak nikt nie chciał korzystać. Natomiast Guido jest jedną z tych rzadkich osób potrafiących zbudować język może odrobinę mniej piękny teoretycznie, ale dzięki temu sprawiający radość osobom, które w nim programują.[1]
>
> *– Jim Hugunin,*
> *twórca Jython, współtwórca AspectJ, architekt .Net DLR*

Jedną z najlepszych zalet Pythona jest jego spójność. Po pewnym czasie programowania w Pythonie możemy zacząć poprawnie zgadywać działanie nowych dla nas funkcjonalności.

Jednak osoby, które wcześniej uczyły się innego języka zorientowanego obiektowo niż Python, mogą uważać za dziwne używanie funkcji `len(collection)` zamiast metody `collection.len()`. Ta pozorna niezwykłość jest tylko czubkiem góry lodowej, której właściwe zrozumienie jest kluczem do wszystkiego, co nazywamy *pythonicznym*. Góra lodowa nazywa się modelem danych Pythona i opisuje interfejs API, którego możemy używać do tworzenia własnych obiektów działających dobrze z najbardziej idiomatycznymi funkcjonalnościami tego języka.

Model danych możemy uważać za opis Pythona jako platformy. Jego zadaniem jest formalizacja interfejsu bloków konstrukcyjnych samego języka, takich jak sekwencje, iteratory, funkcje, klasy, menedżery kontekstu itp.

Podczas kodowania z wykorzystaniem dowolnej platformy dużo czasu spędzamy, implementując metody wywoływane przez tę platformę. To samo dzieje się, gdy polegamy na modelu danych Pythona. Interpreter Pythona wywołuje metody specjalne, aby

1 *Story of Jython* [Historia Jythona] (*http://hugunin.net/story_of_jython.html*), napisana jako przedmowa do książki *Jython Essentials* (O'Reilly, 2002), której autorami są Samuele Pedroni i Noel Rappin.

wykonywać podstawowe operacje na obiektach, często wyzwalane przez specjalną składnię. Nazwy metod specjalnych są zawsze zapisywane z dwoma podkreśleniami z przodu i z tyłu (tj. `__getitem__`). Na przykład składnia `obj[key]` jest obsługiwana przez metodę specjalną `__getitem__`. W celu przetworzenia kodu `my_collection[key]` interpreter wywołuje metodę `my_collection.__getitem__(key)`.

Nazwy metod specjalnych pozwalają naszym obiektom na implementację i obsługę podstawowych konstrukcji języka oraz interakcję z nimi. Przykładami podstawowych konstrukcji języka są:

- iteracja
- kolekcje
- dostęp do atrybutów
- przeciążanie operatorów
- wywoływanie funkcji i metod
- tworzenie i niszczenie obiektów
- reprezentacja i formatowanie łańcuchów
- konteksty zarządzane (tj. bloki `with`)

Magiczne i dunder

Termin *metoda magiczna* to slangowe określenie metody specjalnej, ale mówiąc o konkretnej metodzie, np. `__getitem__`, niektórzy programiści Pythona skracają jej nazwę do „under-under-getitem" (podkreślenie-potkreślenie-getitem). Jest to jednak dwuznaczne, ponieważ składnia __x ma inne znaczenie specjalne[2]. Precyzyjne wymawianie „under-under-getitem-under-under" jest męczące, wiec skorzystam z rady autora i nauczyciela o nazwisku Steve Holden i powiem „dunder-getitem." Wszyscy doświadczeni Pythoniści rozumieją ten skrót. W efekcie metody specjalne są nazywane również *metodami dunder*[3].

Pythoniczna talia kart

Oto bardzo prosty przykład, który demonstruje siłę implementacji zaledwie dwóch metod specjalnych, `__getitem__` i `__len__`.

2 Zobacz „Prywatne i ‹chronione› atrybuty w Pythonie" w rozdziale 9.

3 Osobiście po raz pierwszy usłyszałem „dunder" od Steva Holdena. Wikipedia pierwsze pisemne użycie „dunder" przypisuje Markowi Johnsonowi i Timowi Hochbergowi w odpowiedziach na pytanie „How do you pronounce __ (double underscore)?" [Jak wymawiać „podwójne podkreślenie"] z listy dyskusyjnej python-list z 26 września 2002: (*https://mail.python.org/pipermail/python-list/2002-September/157561.html*).

Przykład 1-1 zawiera kod klasy reprezentującej talię kart do gry.

Przykład 1-1 *Talia jako sekwencja kart*

```
import collections
Card = collections.namedtuple('Card', ['rank', 'suit'])
class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()
    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]
    def __len__(self):
        return len(self._cards)
    def __getitem__(self, position):
        return self._cards[position]
```

Na początek warto zawrócić uwagę na użycie `collections.namedtuple` do konstrukcji prostej klasy reprezentującej poszczególne karty. Od wersji Python 2.6 `namedtuple` może służyć do budowania klas obiektów, które są po prostu wiązkami atrybutów bez żadnych własnych metod, przypominającymi rekordy bazy danych. W tym przykładzie użyliśmy przyjemnej reprezentacji kart w talii, jak widać w sesji konsoli:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

Jednak istotą tego przykładu jest klasa `FrenchDeck` (francuska talia kart). Jest krótka, ale mocna. Po pierwsze, jak wszystkie kolekcje Pythona, talia reaguje na funkcję `len()`, zwracając liczbę zawartych w niej kart:

```
>>> deck = FrenchDeck()
>>> len(deck)
5
```

Odczytanie konkretnych kart z talii – powiedzmy, pierwszej i ostatniej – powinno być proste, jak `deck[0]` lub `deck[-1]`, a to właśnie zapewnia metoda `__getitem__`:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Czy powinniśmy utworzyć metodę służącą do wyboru losowej karty? Nie ma potrzeby. Python ma już funkcję służącą do pobierania losowego elementu z sekwencji: `random.choice`. Możemy jej użyć po prostu na wystąpieniu talii:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

Zobaczyliśmy właśnie dwie zalety używania metod specjalnych wspierających model danych Pythona:

- Użytkownicy naszych klas nie muszą zapamiętywać różnych nazw metod dla operacji standardowych („Jak pobrać liczbę elementów? Czy było to `.size()`, `.length()`, czy coś innego?”).
- Łatwiej jest skorzystać z bogatej biblioteki standardowej Pythona i unikać ponownego wynajdowania koła, jak w przypadku funkcji `random.choice`.

Ale będzie jeszcze lepiej.

Ponieważ nasza metoda `__getitem__` odwołuje się do operatora `[]` atrybutu `self._cards`, nasza talia automatycznie obsługuje wycinanie. Oto jak możemy zobaczyć trzy karty z wierzchu nowej talii, a następnie wybrać tylko asy, zaczynając od indeksu 12 i pomijając 13 kart za każdym razem:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3', suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Dzięki implementacji metody specjalnej `__getitem__` nasza talia umożliwia iterowanie:

```
>>> for card in deck:  # doctest: +ELLIPSIS
...   print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

Możemy iterować po tali również w przeciwną stronę:

```
>>> for card in reversed(deck):  # doctest: +ELLIPSIS
...   print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
```

```
Card(rank='Q', suit='hearts')
...
```

Wielokropki w testach doctest

Kiedy to tylko możliwe, listingi z konsoli Pythona w tej książce są wyodrębniane z testów doctest, aby zapewnić ich dokładność. Jeśli wyniki są zbyt długie, pominięta część jest oznaczana wielokropkiem (`...`), jak w ostatnim wierszu poprzedniego kodu. W takich przypadkach używamy dyrektywy `# doctest: +ELLIPSIS`, aby test doctest przeszedł pomyślnie. W przypadku stosowania tych przykładów w konsoli interaktywnej możemy całkowicie pominąć dyrektywy doctest.

Iteracja jest często niejawna. Jeśli kolekcja nie ma metody `__contains__`, operator `in` przeprowadza skanowanie sekwencyjne. W tym przypadku: `in` działa z klasą `FrenchDeck` ponieważ jest ona iterowalna. Sprawdźmy:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

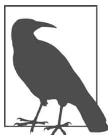
A sortowanie? Częstym systemem określania rankingu kart jest ich wartość (gdzie asy są najwyższe), a następnie kolor w kolejności od najwyższych do najniższych: spades (piki), hearts (kiery), diamonds (karo) i clubs (trefle). Oto funkcja, która ustawia karty według tej zasady, zwracając `0` dla 2 trefl, a `51` dla asa pik:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)
def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) + suit_values[card.suit]
```

Korzystając z funkcji `spades_high`, możemy teraz wyświetlić talię w kolejności rosnącej:

```
>>> for card in sorted(deck, key=spades_high):  # doctest: +ELLIPSIS
...      print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards ommitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Chociaż klasa `FrenchDeck` niejawnie dziedziczy z klasy `object`[4], jej funkcjonalność nie jest dziedziczona, ale pochodzi z podległego modelu danych i kompozycji. Dzięki implementacji metod specjalnych, `__len__` i `__getitem__`, klasa `FrenchDeck` zachowuje się jak standardowa sekwencja Pythona, pozwalając na korzystanie z podstawowych funkcjonalności języka (np. iteracji i wycinania) oraz z biblioteki standardowej, jak widać na przykładach korzystających z funkcji `random.choice`, `reversed` i `sorted`. Dzięki kompozycji implementacje metod `__len__` i `__getitem__` mogą delegować całą pracę do obiektu `list` o nazwie `self._cards`.



A tasowanie?

Przy dotychczasowej implementacji talii `FrenchDeck` nie da się tasować, ponieważ jest *niezmienna*: karty i ich pozycje nie mogą być zmieniane bez naruszenia hermetyzacji i bezpośredniej obsługi atrybutu `_cards`. W rozdziale 11 zostanie to naprawione przez dodanie jednowierszowej metody `__setitem__`.

Sposoby używania metod specjalnych

Najważniejszą cechą metod specjalnych jest to, że mają być wywoływane przez interpreter Pythona, a nie przez programistów. Nie piszemy `my_object.__len__()`. Piszemy `len(my_object)`, a jeśli `my_object` jest wystąpieniem klasy zdefiniowanej przez użytkownika, wtedy Python wywołuje zaimplementowaną metodę `__len__`.

Jednak w przypadku typów wbudowanych, takich jak `list`, `str`, `bytearray` itd. interpreter używa skrótu: implementacja CPython funkcji `len()` rzeczywiście zwraca wartość pola `ob_size` w strukturze `PyVarObject` języka C, która reprezentuje dowolny wbudowany obiekt o zmiennym rozmiarze umieszczony w pamięci. Jest to znacznie szybsze od wywołania metody.

Najczęściej wywoływanie metod specjalnych odbywa się niejawnie. Na przykład instrukcja `for i in x:` rzeczywiście powoduje wywołanie funkcji `iter(x)`, która z kolei może wywołać metodę `x.__iter__()`, jeśli jest ona dostępna.

Zwykle kod nie powinien zawierać wielu bezpośrednich wywołań metod specjalnych. O ile nie zajmujemy się metaprogramowaniem, powinniśmy częściej implementować metody specjalne niż wywoływać je jawnie. Jedyną metodą specjalną, która jest często wywoływana bezpośrednio w kodzie użytkownika, jest metoda `__init__`. Służy ona do wywołania inicjalizatora klasy nadrzędnej we własnej implementacji metody `__init__`.

Jeśli potrzebujemy wywołać metodę specjalną, zwykle lepiej jest wywołać związaną z nią funkcję wbudowaną (np. `len`, `iter`, `str`, itd.). Te wbudowane funkcje wywołują odpowiednią metodę specjalną, ale często dostarczają także inne usługi, a ponadto

4 W wersji Python 2 konieczny był jawny zapis `FrenchDeck(object)`, ale w wersji Python 3 jest to domyślne.

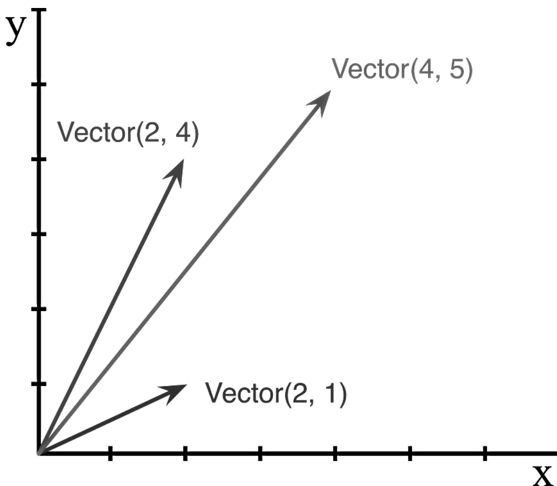
– w przypadku typów wbudowanych – są szybsze od wywołań metod. Zobacz na przykład „Bliższe przyjrzenie się funkcji iter” w rozdziale 14.

Należy unikać tworzenia dowolnych, niestandardowych atrybutów o składni `__foo__`, ponieważ te nazwy mogą nabrać specjalnego znaczenia w przyszłości, nawet jeśli nie są obecnie używane.

Emulacja typów liczbowych

Wiele metod specjalnych pozwala obiektom użytkownika reagować na operatory, takie jak `+`. Zajmiemy się tym bardziej szczegółowo w rozdziale 13. Tutaj naszym celem jest zilustrowanie użycia metod specjalnych kolejnym prostym przykładem.

Zaimplementujemy klasę reprezentującą wektory dwuwymiarowe – czyli wektory euklidesowe, takie jak używane w matematyce i fizyce (patrz rysunek 1-1).



Rysunek 1-1 *Przykład dodawania dwuwymiarowych wektorów. Vector(2, 4) + Vector(2, 1) daje w wyniku Vector(4, 5).*

Do reprezentacji wektorów dwuwymiarowych wystarczyłby wbudowany typ `complex`, ale naszą klasę da się rozszerzyć, aby reprezentowała wektory *n*-wymiarowe. Zrobimy to w rozdziale 14.

Zaczniemy od zaprojektowania interfejsu API dla takiej klasy. W tym celu napiszemy symulowaną wersję sesji konsoli, której użyjemy później jako testu doctest. Następujący fragment służy do testowania dodawania wektorów zilustrowanego na rysunku 1-1:

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
```

```
>>> v1 + v
Vector(4, 5)
```

Zauważ, jak operator + tworzy wynik Vector, który jest wyświetlany w przyjazny sposób w konsoli.

Wbudowana funkcja abs zwraca wartość bezwzględną liczb całkowitych i zmiennoprzecinkowych oraz moduł liczb zespolonych (complex). Zatem w naszym interfejsie API dla spójności również użyjemy funkcji abs do obliczenia modułu wektora:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.
```

Możemy także zaimplementować operator *, aby można było mnożyć przez skalar (tj. mnożyć wektor przez liczbę, aby wytworzyć nowy wektor o tym samym zwrocie i przemnożonym module):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.
```

Przykład 1-2 to klasa Vector implementująca właśnie opisane operacje dzięki użyciu metod specjalnych __repr__, __abs__, __add__ i __mul__.

Przykład 1-2 *Prosta klasa wektora dwuwymiarowego*

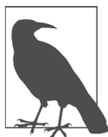
```
from math import hypot
class Vector:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Vector(%r, %r)' % (self.x, self.y)
    def __abs__(self):
        return hypot(self.x, self.y)
    def __bool__(self):
        return bool(abs(self))
    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)
    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

Zauważ, że chociaż zaimplementowaliśmy cztery metody specjalne (poza `__init__`), żadna z nich nie jest bezpośrednio wywoływana wewnątrz klasy ani w typowym użyciu klasy ilustrowanym przez listingi konsoli. Jak wspomniałem wcześniej, przeważnie są one wywoływane tylko przez interpreter Pythona. W kolejnych podrozdziałach omówimy kod poszczególnych metod specjalnych.

Reprezentacja tekstowa

Metoda specjalna `__repr__` jest wywoływana przez wbudowaną funkcję `repr`, aby otrzymać reprezentację tekstową obiektu do inspekcji. Jeśli nie zaimplementowalibyśmy metody `__repr__`, wystąpienia wektorów byłyby pokazane w konsoli w taki sposób: `<Vector object at 0x10e100070>`.

Konsola interaktywna i debuger wywołują funkcję `repr` na wynikach przetwarzanych wyrażeń, tak jak robi to symbol zastępczy `%r` w klasycznym formatowaniu z operatorem `%` i pole konwersji !r w nowej składni *Format String Syntax* (*http://bit.ly/1Vm7gD1*) używanej w metodzie `str.format`.



Jeśli chodzi o operator `%` i metodę `str.format`, obie są używane zarówno przeze mnie w tej książce, jak i przez większość społeczności Pythona. Coraz bardziej preferuję potężniejszą metodę `str.format`, ale zdaję sobie sprawę, że wielu Pythonistów woli prostszy operator `%`, więc w najbliższej przyszłości prawdopodobnie będziemy widzieć oba te rozwiązania w kodzie źródłowym Pythona.

Zauważ, że w naszej implementacji `__repr__` użyliśmy `%r` do otrzymania standardowej reprezentacji atrybutów do wyświetlenia. Jest to dobra praktyka, ponieważ pokazuje istotną różnicę między `Vector(1, 2)` a `Vector('1', '2')` – drugi przypadek nie działałby w kontekście tego przykładu, ponieważ argumentami konstruktora muszą być liczby, a nie łańcuchy `str`.

Łańcuch znaków zwracany przez `__repr__` powinien być jednoznaczny i, o ile to możliwe, odpowiadać kodowi źródłowemu koniecznemu do ponownego utworzenia reprezentowanego obiektu. Dlatego nasza wybrana reprezentacja wygląda tak, jak wywołanie konstruktora klasy (np. `Vector(3, 4)`).

Porównajmy metodę `__repr__` z metodą `__str__`, która jest wywoływana przez konstruktor `str()` i niejawnie używana w funkcji `print`. Metoda `__str__` powinna zwracać łańcuch odpowiedni do wyświetlenia dla użytkowników końcowych.

W przypadku implementacji tylko jednej z tych metod specjalnych, lepiej wybrać `__repr__`, ponieważ, gdy nie ma dostępnej niestandardowej metody `__str__`, Python wywoła `__repr__` jako metodę rezerwową.

„Difference between `__str__` and `__repr__` in Python" (*http://bit.ly/1Vm7j1N*) to pytanie z witryny Stack Overflow, na które wspaniałych odpowiedzi udzielili Pythoniści Alex Martelli i Martijn Pieters.

Operatory arytmetyczne

Przykład 1-2 implementuje dwa operatory: `+` i `*`, aby pokazać podstawowe zastosowanie metod `__add__` i `__mul__`. Zauważ, że w obu przypadkach te metody tworzą i zwracają nowe wystąpienie klasy `Vector` i nie modyfikują żadnego z operandów – `self` ani `other`. Są one jedynie odczytywane. Jest to oczekiwane zachowanie operatorów infiksowych: tworzenie nowych obiektów bez modyfikacji operandów. Będę mieć więcej do powiedzenia na ten temat w rozdziale 13.

Zgodnie z implementacją przykład 1-2 pozwala na mnożenie obiektu `Vector` przez liczbę, ale nie liczby przez `Vector`, co narusza właściwość przemienności mnożenia. Naprawimy to w metodzie specjalnej `__rmul__` w rozdziale 13.

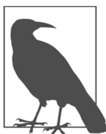
Wartość Boolean typu niestandardowego

Chociaż Python ma typ `bool`, przyjmuje dowolny obiekt w kontekstach logicznych, takich jak wyrażenia kontrolujące instrukcje `if` lub `while` albo jako operandy operatorów `and`, `or` i `not`. Aby wyznaczyć, czy wartość `x` jest *truthy* (prawdziwa) czy *falsy* (fałszywa), Python stosuje `bool(x)`, co zawsze zwraca `True` lub `False`.

Domyślnie wystąpienia klas definiowanych przez użytkownika są uważane za truthy, o ile nie mają zaimplementowanych metod `__bool__` ani `__len__`. Zasadniczo `bool(x)` wywołuje `x.__bool__()` i wykorzystuje wynik tej metody. Jeśli metoda `__bool__` nie jest zaimplementowana, Python próbuje wywołać metodę `x.__len__()`, a jeśli ona zwraca zero, `bool` zwraca `False`. W przeciwnym przypadku `bool` zwraca `True`.

Nasza implementacja metody `__bool__` jest koncepcyjne prosta: zwraca `False`, jeśli moduł wektora jest równy zero, a w przeciwnym przypadku `True`. Konwertujemy moduł na Boolean przy użyciu `bool(abs(self))`, ponieważ metoda `__bool__` ma zwracać zgodnie z oczekiwaniem typ logiczny.

Zauważ, jak specjalna metoda `__bool__` pozwala obiektom na spójność z regułami testowania wartości prawdy zdefiniowanymi w rozdziale „Built-in Types" dokumentacji *The Python Standard Library* (*http://docs.python.org/3/library/stdtypes.html#truth*).



Szybsza implementacja `Vector.__bool__` jest taka:

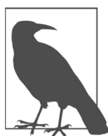
```
def __bool__(self):
    return bool(self.x or self.y)
```

Jest to trudniejsze do odczytania, ale unika podróży przez `abs`, `__abs__`, potęgowanie i pierwiastkowanie. Jawna konwersja na `bool` jest potrzebna, ponieważ `__bool__` musi zwracać boolean, a `or` zwraca jeden z operandów, czyli: `x or y` jest szacowane jako `x`, gdy ten operand jest *truthy*, a w przeciwnym przypadku wynikiem jest `y`, czymkolwiek jest.

Przegląd metod specjalnych

Rozdział „Data Model" [Model danych] dokumentacji *The Python Language Reference* zawiera listę 83 nazw metod specjalnych, z których 47 służy do implementacji operatorów arytmetycznych, bitowych i porównania.

Przegląd dostępnych metod zawierają tabele 1-1 i 1-2.



Grupowanie pokazane w poniższych tabelach niekoniecznie pokrywa się z oficjalną dokumentacją.

Tabela 1-1 *Nazwy metod specjalnych (bez operatorów)*

Kategoria	Nazwy metod
Reprezentacja tekstowa/bajtowa	`__repr__`, `__str__`, `__format__`, `__bytes__`
Konwersja na liczbę	`__abs__`, `__bool__`, `__complex__`, `__int__`, `__float__`, `__hash__`, `__index__`
Emulacja kolekcji	`__len__`, `__getitem__`, `__setitem__`, `__delitem__`, `__contains__`
Iteracja	`__iter__`, `__reversed__`, `__next__`
Emulacja wywoływalności	`__call__`
Zarządzanie kontekstem	`__enter__`, `__exit__`
Tworzenie i niszczenie wystąpienia	`__new__`, `__init__`, `__del__`
Zarządzanie atrybutami	`__getattr__`, `__getattribute__`, `__setattr__`, `__delattr__`, `__dir__`
Deskryptory atrybutów	`__get__`, `__set__`, `__delete__`
Usługi klasy	`__prepare__`, `__instancecheck__`, `__subclasscheck__`

Tabela 1-2 *Nazwy metod specjalnych dla operatorów*

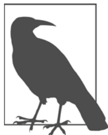
Kategoria	Nazwy metod i powiązane operatory
Jednoargumentowe operatory numeryczne	`__neg__` -, `__pos__` +, `__abs__` `abs()`
Bogate operatory porównania	`__lt__` >, `__le__` <=, `__eq__` ==, `__ne__` !=, `__gt__` >, `__ge__` >=
Operatory arytmetyczne	`__add__` +, `__sub__` -, `__mul__` *, `__truediv__` /, `__floordiv__` //, `__mod__` %, `__divmod__` `divmod()` , `__pow__` ** lub `pow()`, `__round__` `round()`
Odwrócone operatory arytmetyczne	`__radd__`, `__rsub__`, `__rmul__`, `__rtruediv__`, `__rfloordiv__`, `__rmod__`, `__rdivmod__`, `__rpow__`
Złożone arytmetyczne operatory przypisania	`__iadd__`, `__isub__`, `__imul__`, `__itruediv__`, `__ifloordiv__`, `__imod__`, `__ipow__`
Operatory bitowe	`__invert__` ~, `__lshift__` <<, `__rshift__` >>, `__and__` &, `__or__` |, `__xor__` ^
Odwrócone operatory bitowe	`__rlshift__`, `__rrshift__`, `__rand__`, `__rxor__`, `__ror__`
Złożone bitowe operatory przypisania	`__ilshift__`, `__irshift__`, `__iand__`, `__ixor__`, `__ior__`

Operatory odwrócone są rezerwowym rozwiązaniem używanym, gdy operandy są zamienione (`b * a` zamiast `a * b`), a złożone operatory przypisania są skrótami łączącymi operator infiksowy z przypisaniem do zmiennej (`a = a * b` staje się `a *= b`). Rozdział 13 zawiera szczegółowy opis operatorów odwróconych i złożonego przypisania.

Dlaczego len nie jest metodą

Deweloper języka, Raymond Hettinger, któremu zadałem to pytanie w 2013, odpowiedział na to pytanie cytatem z tekstu *The Zen of Python*: „practicality beats purity” (praktyczność pokonuje czystość). W podrozdziale „Sposoby używania metod specjalnych” opisałem, dlaczego `len(x)` działa bardzo szybko, gdy `x` jest wystąpieniem typu wbudowanego. Żadna metoda nie jest wywoływana dla wbudowanych obiektów implementacji CPython: długość jest po prostu odczytywana z pola struktury w języku C. Pobranie liczby elementów z kolekcji jest częstą operacją i musi działać wydajnie dla takich podstawowych i różnorodnych typów jak `str`, `list`, `memoryview` itd.

Innymi słowy, funkcja `len` nie jest wywoływana jako metoda, ponieważ jest specjalnie traktowana jako część modelu danych Pythona, podobnie jak `abs`. Jednak dzięki specjalnej metodzie `__len__` możemy sprawić, że funkcja `len` będzie działać dla naszych niestandardowych obiektów. Jest to uczciwy kompromis między potrzebą wydajności wbudowanych obiektów a spójnością języka. Jest to również zgodne z tekstem *The Zen of Python*: „Special cases aren't special enough to break the rules" [specjalne przypadki nie są wystarczająco specjalne, aby naruszać reguły].



Jeśli pomyślimy o `abs` i `len` jako o operatorach jednoargumentowych, możemy być bardziej skłonni do wybaczenia ich funkcyjnego stylu, tak różnego od składni wywołań metod, której możemy oczekiwać po języku obiektowym. Faktycznie język ABC – bezpośredni przodek Pythona, który przetarł szlaki wielu jego funkcjonalnościom – miał operator #, który był odpowiednikiem funkcji `len` (pisało się #s). Kiedy używało się go jako operatora infiksowego, zapisując x#s, zliczał wystąpienia x w s, co w Pythonie otrzymujemy za pomocą metody `s.count(x)` dla dowolnej sekwencji s.

Podsumowanie rozdziału

Dzięki implementacji metod specjalnych nasze obiekty mogą zachowywać się podobnie do typów wbudowanych, pozwalając na wyrazisty styl kodowania uważany przez społeczność za pythoniczny.

Podstawowym wymogiem dla obiektu Pythona jest dostarczanie użytecznej reprezentacji tekstowej tego obiektu, którą jedni używają do debugowania i rejestrowania, a inni do prezentacji użytkownikom końcowym. Dlatego model danych zawiera metody specjalne `__repr__` i `__str__`.

Emulacja sekwencji, pokazana w przykładzie `FrenchDeck`, jest jednym z najpowszechniej używanych zastosowań metod specjalnych. Zapoznanie się z większością typów sekwencyjnych jest tematem rozdziału 2, a implementacja własnej sekwencji zostanie opisana w rozdziale 10, w którym utworzymy wielowymiarowe rozszerzenie klasy `Vector`.

Dzięki przeciążaniu operatorów Python oferuje bogaty wybór typów liczbowych, od wbudowanych do `decimal.Decimal` i `fractions.Fraction`. Wszystkie obsługują infiksowe operatory arytmetyczne. Implementacja operatorów, w tym operatorów odwrotnych i złożonego przypisania, zostanie pokazana w rozdziale 13 jako rozwinięcie przykładu klasy `Vector`.

Użycie i implementacja większości pozostałych metod specjalnych modelu danych Pythona jest zawarta w treści tej książki.

Lektura uzupełniająca

Rozdział „Data Model” dokumentacji *The Python Language Reference* jest kanonicznym źródłem tematów tego rozdziału i większości tej książki.

Książka *Python in a Nutshell, 2nd Edition* (O'Reilly), której autorem jest Alex Martelli, wspaniale opisuje model danych. Kiedy pisałem niniejszą książkę, ostatnie wydanie książki *Nutshell* pochodziło z roku 2006 i skupiało się na wersji Python 2.5, ale od tego czasu bardzo niewiele zmieniło się w modelu danych, a opis Martelliego mechaniki dostępu do atrybutów jest najbardziej autorytatywnym, jaki widziałem, oprócz rzeczywistego kodu źródłowego C implementacji CPython. Martelli ma także duży wkład w witrynę Stack Overflow, z ponad 5 000 wpisów odpowiedzi. Zobacz jego profil użytkownika w witrynie Stack Overflow (*http://stackoverflow.com/users/95810/alex-martelli*).

David Beazley napisał dwie książki opisujące szczegółowo model danych w kontekście wersji Python 3: *Python Essential Reference, 4th Edition* (Addison-Wesley Professional) i *Python Cookbook, 3rd Edition* (O'Reilly) [wyd. polskie *Python. Receptury* (Helion)], której współautorem jest Brian K. Jones.

W książce *The Art of the Metaobject Protocol* (AMOP, MIT Press), której autorami są Gregor Kiczales, Jim des Rivieres i Daniel G. Bobrow, objaśniono koncepcję protokołu metaobiektów (MOP), którego przykładem jest model danych Pythona.

Pogadanka

Model danych czy model obiektowy?

To co w dokumentacji Pythona jest nazywane „modelem danych Pythona”, większość autorów określa jako „model obiektowy Pythona”. *Python in a Nutshell 2E*, której autorem jest Alex Martelli, oraz *Python Essential Reference 4E*, której autorem jest David Beazley, są najlepszymi książkami opisującymi „model danych Pythona”, jednak zawsze odnoszą się do niego jako do „modelu obiektowego”. W Wikipedii pierwsza definicja *modelu obiektowego* brzmi „Właściwości obiektów w ogólności w konkretnym języku programowania komputerowego” (*http://en.wikipedia.org/wiki/Object_model*). To właśnie opisuje „model danych Pythona”. W tej książce używam pojęcia „model danych”, ponieważ w dokumentacji ten termin jest preferowany podczas odwoływania się do modelu obiektowego Pythona oraz ponieważ jest to tytuł rozdziału dokumentacji *The Python Language Reference* najbardziej związanego z niniejszymi rozważaniami.

Metody magiczne

Społeczność Ruby nazywa swoje odpowiedniki metod specjalnych *metodami magicznymi*. Duża część społeczności Pythona również przyjęła ten termin. Osobiście uważam, że metody specjalne są faktycznym przeciwieństwem magii. Python i Ruby są takie same pod tym względem: oba języki wspomagają użytkowników bogatym protokołem metaobiektów, który nie jest magiczny, ale pozwala użytkownikom na korzystanie z tych samych narzędzi, które są dostępne dla deweloperów języka.

Jako przeciwieństwo rozważmy JavaScript. Obiekty w tym języku mają cechy, które są magiczne, pod tym względem, że nie można ich emulować we własnych obiektach definiowanych przez użytkownika. Na przykład przed wersją JavaScript 1.8.5 nie można było definiować atrybutów tylko do odczytu w swoich obiektach JavaScript, ale niektóre wbudowane obiekty zawsze miały atrybuty tylko do odczytu. W języku JavaScript atrybuty tylko do odczytu były „magiczne", wymagające ponadnaturalnych mocy, których użytkownicy tego języka nie mieli do czasu wydania ECMAScript 5.1 w roku 2009. Protokół metaobiektów języka JavaScript ewoluuje, ale historycznie był bardziej ograniczony niż protokoły metaobiektów Pythona i Ruby.

Metaobiekty

The Art of the Metaobject Protocol (AMOP) to tytuł mojej ulubionej książki informatycznej. Mniej subiektywnie termin *protokół metaobiektów* przydaje się do myślenia o modelu danych Pythona i podobnych funkcjonalnościach w innych językach. Część *metaobiekt* odnosi się do obiektów, które są blokami konstrukcyjnymi samego języka. W tym kontekście *protokół* jest synonimem *interfejsu*. Zatem *protokół metaobiektów* jest fantazyjnym synonimem modelu obiektowego: interfejsu API podstawowych konstrukcji języka.

Bogaty protokół metaobiektów pozwala na rozszerzanie języka, aby obsługiwał nowe paradygmaty programowania. Gregor Kiczales, pierwszy autor książki *AMOP*, później stał się pionierem programowania zorientowanego na aspekty i autorem inicjującym AspectJ, rozszerzenia języka Java implementującego ten paradygmat. Projektowanie zorientowane na aspekty jest łatwiejsze do zaimplementowania w języku dynamicznym, takim jak Python, i służy do tego wiele platform, ale najważniejszą jest `zope.interface`, opisana pokrótce w części *Lektura uzupełniająca* w rozdziale 11.

Część II

Struktury danych

Rozdział 2

Sekwencje i tablice

> Jak łatwo zauważyć, wiele wspomnianych operacji działa tak samo dla tekstów, list i tabel. Tekst, listy i tabele razem są nazywane *ciągami*. […] Polecenie `FOR` także działa ogólnie na ciągach.[1]
>
> *– Geurts, Meerten i Pemberton*
> *ABC Programmer's Handbook*

Przed tworzeniem Pythona Guido zajmował się językiem ABC – 10-letnim projektem badawczym dotyczącym projektowania środowiska programistycznego dla początkujących. W języku ABC wprowadzono wiele pomysłów uważanych obecnie za „pythoniczne": generyczne operacje na sekwencjach, wbudowane krotki i typy odwzorowujące, strukturyzacja za pomocą wcięć, silne typowanie bez deklaracji zmiennych itp. Nie jest przypadkiem, że Python jest tak przyjazny dla użytkowników.

Python odziedziczył z ABC ujednoliconą obsługę sekwencji. Łańcuchy, listy, sekwencje bajtów, tablice, elementy XML i wyniki baz danych współdzielą bogaty zbiór operacji, obejmujący iteracje, wycinanie, sortowanie i łączenie.

Zrozumienie różnorodności sekwencji dostępnych w Pythonie chroni przed ponownym wynajdowaniem koła, a ich wspólny interfejs inspiruje do tworzenia interfejsów API właściwie obsługujących i wykorzystujących istniejące i przyszłe typy sekwencyjne.

Większość treści tego rozdziału dotyczy sekwencji w ogólności: od znajomego typu `list` do `str` i `bytes`, które są nowościami w wersji Python 3. Znajdziemy tu również konkretne tematy dotyczące list, krotek, tablic i kolejek, ale na łańcuchach Unicode i sekwencjach bajtów skupimy się dopiero w rozdziale 4. Ponadto celem niniejszego rozdziału jest opisanie gotowych do użycia typów sekwencji. Natomiast tworzenie własnych typów sekwencji jest tematem rozdziału 10.

1 Leo Geurts, Lambert Meertens i Steven Pemberton, *ABC Programmer's Handbook*, str. 8.

Przegląd wbudowanych sekwencji

Biblioteka standardowa oferuje bogaty wybór typów sekwencji zaimplementowanych w języku C:

Sekwencje kontenerowe
: `list`, `tuple` i `collections.deque` mogą przechowywać elementy różnych typów.

Sekwencje płaskie
: `str`, `bytes`, `bytearray`, `memoryview` i `array.array` przechowują elementy jednego typu.

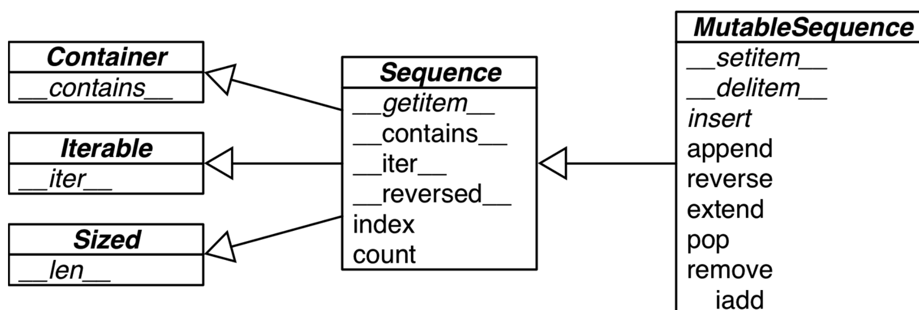
Sekwencje kontenerowe przechowują odwołania do zawartych w sobie obiektów, które mogą być dowolnego typu, natomiast *sekwencje płaskie* fizycznie przechowują wartości poszczególnych elementów we własnej przestrzeni pamięci, a nie jako oddzielne obiekty. Zatem sekwencje płaskie są bardziej upakowane, ale przy tym ograniczone do przechowywania prymitywnych wartości, takich jak znaki, bajty i liczby.

Innym sposobem grupowania sekwencji jest ich zmienność:

Sekwencje zmienne
: `list`, `bytearray`, `array.array`, `collections.deque` i `memoryview`

Sekwencje niezmienne
: `tuple`, `str` i `bytes`

Rysunek 2-1 pomaga zwizualizować różnice między sekwencjami zmiennymi a niezmiennymi, a także podobieństwa wynikające z dziedziczenia wielu metod. Zauważ, że konkretne wbudowane typy sekwencji nie są faktycznie podklasami abstrakcyjnych klas bazowych (ABC) `Sequence` i `MutableSequence.` Niemniej jednak klasy ABC są przydatne do formalizowania oczekiwanych funkcjonalności w pełni funkcjonalnych typów sekwencyjnych.



Rysunek 2-1 *Diagram klas UML dla pewnych klas w module collections.abc (klasy nadrzędne są po lewej; strzałki dziedziczenia są skierowane od klas podrzędnych do nadrzędnych; nazwy klas abstrakcyjnych i metod abstrakcyjnych są zapisane kursywą)*

Rozważanie tych powszechnych cech – zmienne albo niezmienne, kontenerowe albo płaskie – pozwala na podstawie znajomości wybranych typów sekwencji przewidzieć działanie pozostałych.

Najbardziej podstawowym typem sekwencji jest `list` – typ zmienny i mieszany. Jestem przekonany, że posługujesz się nim bezproblemowo, więc przejdziemy od razu do wyrażeń listowych. Ten potężny sposób budowania list jest trochę zbyt rzadko używany z powodu nieznajomości składni. Biegła znajomość wyrażeń listowych otwiera drzwi do wyrażeń generatora, które – oprócz innych zastosowań – mogą wytwarzać elementy do wypełniania sekwencji dowolnego typu. Oba te wyrażenia są tematem następnego podrozdziału.

Wyrażenia listowe i wyrażenia generatora

Szybkim sposobem na zbudowanie sekwencji jest użycie wyrażenia listowego (jeśli celem jest `list`) lub wyrażenia generatora (dla wszystkich innych rodzajów sekwencji). Jeśli nie używasz tych form syntaktycznych na co dzień, założę się, że tracisz możliwości pisania kodu, który jest bardziej czytelny, a często również szybszy.

Jeśli wątpisz w moje zapewnienie, że te konstrukcje są „bardziej czytelne”, czytaj dalej. Spróbuję Cię przekonać.

Dla zwięzłości wielu programistów Pythona nazywa wyrażenie listowe *listcomp*, a wyrażenie generatora *genexp*.

Wyrażenia listowe a czytelność

Oto test: co uważasz za łatwiejsze do przeczytania: przykład 2-1 czy przykład 2-2?

Przykład 2-1 *Budowanie listy punktów kodowych Unicode na podstawie łańcucha*

```
>>> symbols = '$¢£¥€¤'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Przykład 2-2 *Budowanie listy punktów kodowych Unicode na podstawie łańcucha, podejście drugie*

```
>>> symbols = '$¢£¥€¤'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Każdy, kto zna choć trochę Pythona, może przeczytać przykład 2-1. Jednak po zapoznaniu się z wyrażeniami listowymi uważam przykład 2-2 za bardziej czytelny, ponieważ jego cel jest jasno sprecyzowany.

Pętla `for` może mieć wiele różnych zastosowań: skanowanie sekwencji, aby zliczać lub wybierać elementy, obliczanie agregacji (sum, średnich) i dowolnie wiele innych zadań przetwarzania. Kod w przykładzie 2-1 buduje listę. Natomiast wyrażenie listowe ma tylko jedno zadanie: budowanie nowej listy.

Oczywiście jest możliwe nadużywanie wyrażeń listowych, aby pisać faktycznie niezrozumiały kod. Widziałem kod Pythona z wyrażeniami listowymi używanymi po prostu po to, aby powtarzać blok kodu dla jego efektu ubocznego. Jeśli budowana lista do niczego nie służy, nie powinniśmy używać tej składni. Ponadto warto zachować zwięzłość. Jeśli wyrażenie listowe zajmuje więcej niż dwa wiersze, prawdopodobnie lepiej je podzielić lub przepisać jako zwykłą starą pętlę `for`. Decyzję podejmujemy subiektywnie: dla języka Python, podobnie jak dla angielskiego, nie ma sztywnych reguł jasnego pisania.

Wskazówka składniowa

W kodzie Pythona podziały wierszy są ignorowane wewnątrz par nawiasów `[]`, `{}` lub `()`. Zatem możemy budować wielowierszowe listy, wyrażenia listowe i generatora, słowniki itp. bez używania brzydkiego znaku ucieczki `\` do kontynuacji wiersza.

Wyrażenia listowe już nie tracą swoich zmiennych

W języku Python 2.x zmienne przypisywane w klauzulach `for` w wyrażeniach listowych były ustawiane w zakresie otoczenia, czasami z tragicznymi konsekwencjami. Zobacz poniższą sesję konsoli w wersji Python 2.7:

```
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> x = 'my precious'
>>> dummy = [x for x in 'ABC']
```

```
>>> x
'C'
```

Jak widzimy, początkowa wartość `x` została nadpisana. Nie dzieje się tak już w wersji Python 3.

Wyrażenia listowe, wyrażenia generatora i ich krewniacy, wyrażenia `set` i `dict`, mają teraz swój lokalny zakres, podobnie jak funkcje. Zmienne przypisywane w wyrażeniach są lokalne, ale nadal możemy się odwoływać do zmiennych z zakresu otoczenia. Jeszcze lepiej, zmienne lokalne nie maskują zmiennych z zakresu otoczenia.

W wersji Python 3:

```
>>> x = 'ABC'
>>> dummy = [ord(x) for x in x]
>>> x  ❶
'ABC'
>>> dummy  ❷
[65, 66, 67]
```

❶ Wartość `x` jest zachowana.

❷ Wyrażenie listowe wytwarza oczekiwaną listę.

Wyrażenia listowe budują listy na podstawie sekwencji lub dowolnego innego typu iterowalnego za pomocą filtrowania i transformacji elementów. W tym samym celu możemy składać wbudowane funkcje `filter` i `map`, ale jak zobaczymy dalej, tracimy wtedy czytelność.

Wyrażenia listowe a funkcje map i filter

Wyrażenia listowe robią wszystko to samo, co funkcje `map` i `filter`, ale bez gmatwania funkcjonalności za pomocą wyrażeń `lambda` Pythona. Rozważmy przykład 2-3.

Przykład 2-3 *Ta sama lista zbudowana za pomocą wyrażenia listowego i złożenia funkcji map/filter*

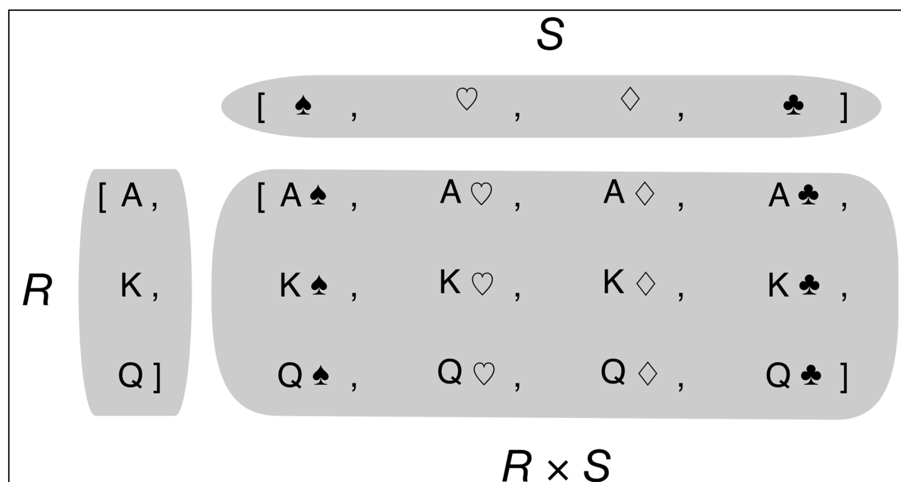
```
>>> symbols = '$¢£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

Kiedyś wierzyłem, że `map` i `filter` są szybsze niż ich odpowiedniki w postaci wyrażeń listowych, ale Alex Martelli pokazał, że tak nie jest – przynajmniej nie w powyższych przykładach. Skrypt *02-array-seq/listcomp_speed.py* w repozytorium kodu tej książki jest prostym testem szybkości, porównującym wyrażenie listowe z `filter/map`.

Więcej na temat funkcji `map` i `filter` będę miał do powiedzenia w rozdziale 5. Teraz wróćmy do użycia wyrażeń listowych do obliczania iloczynu kartezjańskiego: listy zawierającej krotki zbudowane z wszystkich elementów co najmniej dwóch list.

Iloczyny kartezjańskie

Wyrażenia listowe mogą generować listy na podstawie produktu kartezjańskiego dwóch obiektów iterowalnych lub większej ich liczby. Elementy, które składają się na produkt kartezjański to krotki powstałe z elementów pochodzących z każdego wejściowego obiektu iterowalnego. Wynikowa lista ma długość równą przemnożonym długościom wejściowych obiektów iterowalnych. Zobacz rysunek 2-2.



Rysunek 2-2 *Iloczyn kartezjański sekwencji trzech wartości kart i sekwencji czterech kolorów, którego wynikiem jest sekwencja dwunastu par*

Wyobraź sobie na przykład, że potrzebujemy utworzyć listę koszulek T-shirt dostępnych w dwóch kolorach i trzech rozmiarach. Przykład 2-4 pokazuje, jak utworzyć taką listę przy użyciu wyrażenia listowego. Wynik ma sześć elementów.

Przykład 2-4 *Iloczyn kartezjański z zastosowaniem wyrażenia listowego*

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in sizes]  ❶
>>> tshirts
```

```
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors:  ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes      ❸
...                          for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

❶ To generuje listę krotek uporządkowanych według koloru (`color`), a następnie rozmiaru (`size`).

❷ Zauważ, że wynikowa lista jest porządkowana tak, jakby pętle `for` były zagnieżdżone w tej samej kolejności, w jakiej występują w wyrażeniu listowym.

❸ Aby uporządkować elementy według rozmiaru, a następnie koloru, po prostu zmieniamy kolejność klauzul `for`. Dodanie podziału wiersza do wyrażenia listowego ułatwia zobaczenie zmiany uporządkowania wyników.

W przykładzie 1-1 (rozdział 1) następujące wyrażenie zostało użyte do zainicjowania talii kart listą 52 kart o wszystkich 13 wartościach (`rank`) i we wszystkich 4 kolorach (`suit`), pogrupowanych według kolorów:

```
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]
```

Wyrażenia listowe mają tylko jedno zastosowanie: budują listy. Aby wypełnić inne typy sekwencji, trzeba użyć wyrażenia generatora. W następnym podrozdziale przyjrzymy się pokrótce wyrażeniom generatora w kontekście budowania sekwencji niebędących listami.

Wyrażenia generatora

Do inicjalizacji krotek, tablic i innych typów sekwencji możemy początkowo używać także wyrażeń listowych, ale wyrażenia generatora oszczędzają pamięć, ponieważ generują pojedyncze elementy przy użyciu protokołu iteratora, zamiast budować całą listę tylko po to, żeby wypełnić inny konstruktor.

Wyrażenia generatora korzystają z tej samej składni, co wyrażenia listowe, ale są zawarte w nawiasach okrągłych, a nie kwadratowych.

Przykład 2-5 przedstawia podstawowe użycie wyrażeń generatora do budowania krotki i tablicy.

Przykład 2-5 *Inicjowanie krotki i tablicy za pomocą wyrażenia generatora*

```
>>> symbols = '$¢£¥€¤'
>>> tuple(ord(symbol) for symbol in symbols)  ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols))  ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

❶ Jeśli wyrażenie generatora stanowi pojedynczy argument w wywołaniu funkcji, nie ma potrzeby dublowania nawiasów otaczających.

❷ Konstruktor `array` przyjmuje dwa argumenty, więc nawiasy wokół wyrażenia generatora są obowiązkowe. Pierwszy argument konstruktora `array` definiuje typ służący do przechowywania liczb w tablicy, jak zobaczymy w podrozdziale *Tablice*.

W przykładzie 2-6 wyrażenie generatora zostało użyte z iloczynem kartezjańskim, aby wypisać asortyment koszulek w dwóch kolorach i trzech rozmiarach. W przeciwieństwie do przykładu 2-4, tutaj sześcioelementowa lista koszulek nigdy nie jest budowana w pamięci: wyrażenie generatora zasila pętlę `for`, tworząc pojedyncze elementy. Jeśli dwie listy użyte w iloczynie kartezjańskim miałyby po 1 000 elementów, użycie wyrażenia generatora oszczędziłoby konstruowania listy z milionem elementów tylko po to, aby zasilić pętlę `for`.

Przykład 2-6 *Iloczyn kartezjański w wyrażeniu generatora*

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in ('%s %s' % (c, s) for c in colors for s in sizes):  ❶
...     print(tshirt)
...
black S
black M
black L
white S
white M
white L
```

❶ Wyrażenie generatora wytwarza elementy pojedynczo. W tym przykładzie lista wszystkich sześciu odmian koszulek nigdy nie powstaje.

Rozdział 14 jest poświęcony szczegółowemu wyjaśnieniu, jak działają generatory. Tutaj chodzi tylko o pokazanie zastosowania wyrażeń generatora do inicjowania sekwencji innych niż listy lub do wytwarzania wyników, których nie potrzebujemy przechowywać w pamięci.

Teraz przejdziemy do innego, fundamentalnego dla Pythona typu sekwencyjnego, którym jest krotka (ang. *tuple*).

Krotki nie są jedynie niezmiennymi listami

Niektóre teksty wprowadzające do Pythona prezentują krotki jako „niezmienne listy", ale jest to ich niedocenianie. Krotki mają podwójną rolę: mogą służyć jako niezmienne listy, a także jako rekordy bez nazw pól. Drugie zastosowanie bywa niedoceniane, więc od niego zaczniemy.

Krotki jako rekordy

Krotki przechowują rekordy: każdy element w krotce przechowuje dane jednego pola, a położenie tego elementu wyznacza jego znaczenie.

Myśląc o krotkach tylko jako niezmiennych listach, możemy uważać, że liczba i kolejność elementów w zależności od kontekstu może, ale nie musi być istotna. Kiedy jednak używamy krotki jako kolekcji pól, liczba elementów jest często stała, a ich kolejność zawsze ważna.

Przykład 2-7 przedstawia zastosowanie krotek jako rekordów. Zauważ, że w każdym wyrażeniu sortowanie krotki zniszczyłoby informacje, ponieważ znaczenie poszczególnych elementów danych zależy od ich położenia w krotce.

Przykład 2-7 *Krotki stosowane jako rekordy*

```
>>> lax_coordinates = (33.9425, -118.408056)  ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32450, 0.66, 8014)  ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'),  ❸
...     ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids):  ❹
...     print('%s/%s' % passport)  ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids:  ❻
...     print(country)
...
USA
BRA
ESP
```

❶ Szerokość i długość geograficzna lotniska międzynarodowego w Los Angeles.

❷ Dane dotyczące Tokio: nazwa, rok, populacja (w milionach), zmiana populacji (%), obszar (km²).

❸ Lista krotek zawierających kod kraju i numer paszportu w formie `(country_code, passport_number)`.

❹ Podczas iteracji przez listę zmienna `passport` jest wiązana z poszczególnymi krotkami.

❺ Operator formatujący `%` „rozumie" krotki i traktuje każdy element jako oddzielne pole.

❻ Pętla `for` „wie" jak oddzielnie pobierać elementy z krotek – jest to nazywane „rozpakowywaniem". Tutaj nie interesuje nas drugi element, więc przypisujemy go do `_`, zmiennej fikcyjnej.

Krotki działają dobrze jako rekordy dzięki mechanizmowi rozpakowywania krotek – naszemu następnemu tematowi.

Rozpakowywanie krotek

W przykładzie 2-7 przypisaliśmy `('Tokyo', 2003, 32450, 0.66, 8014)` do zmiennych `city` (miasto), `year` (rok), `pop` (populacja), `chg` (zmiana), `area` (obszar) w jednym poleceniu. Następnie w ostatnim wierszu operator `%` przypisał każdy element krotki `passport` (paszport) do jednego przedziału w łańcuchu formatującym podanym w argumencie funkcji `print`. Są to dwa przykłady *rozpakowywania krotek*.

Rozpakowywanie krotek działa z każdym obiektem iterowalnym. Jedynym wymaganiem jest, aby obiekt iterowalny generował dokładnie jeden element na zmienną w odbierającej krotce, o ile nie użyjemy gwiazdki (`*`) w celu przechwycenia nadmiarowych elementów zgodnie z objaśnieniem w podrozdziale „Użycie * do zagarnięcia nadmiarowych elementów". Termin *rozpakowywanie krotek* jest powszechnie używany przez Pythonistów, ale *rozpakowywanie iterowalnych* zyskuje na popularności, jak w tytule dokumentu *PEP 3132 – Extended Iterable Unpacking* [Rozszerzone rozpakowywanie iterowalnych].

Najbardziej widoczną formą rozpakowywania krotek jest *przypisanie równoległe*. Polega ono na przypisywaniu elementów z obiektu iterowalnego do krotki zmiennych, jak w poniższym przykładzie:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates  # rozpakowywanie krotki
>>> latitude
33.9425
```

```
>>> longitude
-118.408056
```

Eleganckim zastosowaniem rozpakowywania krotek jest zamiana wartości zmiennych bez używania zmiennej tymczasowej:

```
>>> b, a = a, b
```

Innym przykładem rozpakowywania krotek jest umieszczenie prefiksu w postaci gwiazdki przed argumentem podczas wywoływania funkcji:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

Powyższy kod przedstawia także dalsze użycie rozpakowywania: funkcje mogą zwracać wiele wartości w sposób wygodny dla wywołującego. Na przykład funkcja `os.path.split()` buduje krotkę `(path, last_part)` na podstawie ścieżki w systemie plików:

```
>>> import os
>>> _, filename = os.path.split('/home/luciano/.ssh/idrsa.pub')
>>> filename
'idrsa.pub'
```

Czasami, gdy interesują nas jedynie niektóre części krotki podczas rozpakowania, posługujemy się zmienną fikcyjną, np. _ jako elementem zastępczym, jak w poprzednim przykładzie.

W przypadku pisania oprogramowania zinternacjonalizowanego _ nie jest dobrą zmienną fikcyjną, ponieważ jest tradycyjnie używana jako alias funkcji `gettext.gettext`, zgodnie z zaleceniem w dokumentacji modułu `gettext` (*http://docs.python.org/3/library/gettext.html*). W innych przypadkach jest to dobra nazwa dla zmiennej zastępczej.

Innym sposobem skupienia się tylko na niektórych elementach podczas rozpakowywania krotki jest użycie *, co zaraz zobaczymy.

Użycie * do zagarnięcia nadmiarowych elementów

Definiowanie parametrów funkcji z użyciem `*args` w celu zagarniania dowolnych nadmiarowych argumentów jest klasyczną funkcjonalnością Pythona.

W wersji Python 3 ten pomysł został rozszerzony do stosowania również przy przypisywaniu równoległym:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

W kontekście przypisywania równoległego prefiks `*` może zostać zastosowany do dokładnie jednej zmiennej, ale może wystąpić na dowolnej pozycji:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Na koniec, potężną funkcjonalnością rozpakowywania krotek jest to, że działa ono ze strukturami zagnieżdżonymi.

Rozpakowywanie zagnieżdżonych krotek

Krotka odbierająca wyrażenie do rozpakowania może mieć zagnieżdżone krotki `(a, b, (c, d))`, a Python poprawnie je obsłuży, jeśli wyrażenie odpowiada zagnieżdżonej strukturze. Przykład 2-8 przedstawia rozpakowywanie zagnieżdżonych krotek w akcji.

Przykład 2-8 *Rozpakowywanie zagnieżdżonych krotek w celu uzyskania dostępu do długości geograficznej*

```
metro_areas = [
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)),   ❶
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),
    ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),
```

```
]
print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))
fmt = '{:15} | {:9.4f} | {:9.4f}'
for name, cc, pop, (latitude, longitude) in metro_areas:  ❷
    if longitude <= 0:  ❸
        print(fmt.format(name, latitude, longitude))
```

❶ Każda krotka przechowuje rekord z czterema polami, a ostatnie z nich jest parą współrzędnych.

❷ Przypisując ostatnie pole do krotki, rozpakowujemy współrzędne.

❸ `if longitude <= 0:` ogranicza wyniki do obszarów metropolitalnych półkuli zachodniej.

Oto wyniki przykładu 2-8:

```
                |   lat.    |   long.
Mexico City     |   19.4333 |  -99.1333
New York-Newark |   40.8086 |  -74.0204
Sao Paulo       |  -23.5478 |  -46.6358
```

Przed wersją Python 3 możliwe było definiowanie funkcji z zagnieżdżonymi krotkami w parametrach formalnych (np. `def fn(a, (b, c), d):`). Nie jest to już dłużej obsługiwane w definicjach funkcji w wersji Python 3. Powodem są względy praktyczne wyjaśnione w dokumencie *PEP 3113 – Removal of Tuple Parameter Unpacking* [Usuwanie rozpakowywania parametrów krotek]. Dla jasności: nic się nie zmieniło z perspektywy użytkowników wywołujących funkcje. Ograniczenie dotyczy tylko definiowania funkcji.

Tak zaprojektowane krotki są bardzo wygodne. Podczas korzystania z nich jako rekordów brakuje jednak pewnej funkcjonalności: czasami przydatne byłoby nazwanie pól. Dlatego wymyślono funkcję `namedtuple`. Czytaj dalej.

Krotki nazwane

Funkcja `collections.namedtuple` to fabryka produkująca podklasy klasy `tuple` wzbogacone nazwami pól i nazwą klasy – co pomaga w debugowaniu.

Instancje klasy budowane za pomocą funkcji `namedtuple` zajmują dokładnie tyle samo pamięci, co krotki, ponieważ nazwy pól są przechowywane w klasie. Używają mniej pamięci niż zwykłe obiekty, ponieważ poszczególne instancje nie przechowują swoich atrybutów w słownikach `__dict__`.

Przypomnijmy sobie, jak budowaliśmy klasę `Card` w przykładzie 1-1 w rozdziale 1:

```
Card = collections.namedtuple('Card', ['rank', 'suit'])
```

Przykład 2-9 pokazuje, jak możemy zdefiniować krotkę nazwaną w celu przechowywania informacji na temat miasta.

Przykład 2-9 *Definiowanie i używanie typu krotki nazwanej*

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population coordinates')  ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))  ❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=(35.689722,
139.691667))
>>> tokyo.population  ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

❶ Do utworzenia krotki nazwanej są wymagane dwa parametry: nazwa klasy i lista nazw pól, która może być podana jako iterowalny obiekt zawierający łańcuchy lub jako jeden łańcuch rozdzielany pojedynczymi spacjami.

❷ Dane muszą być przekazywane jako argumenty o odpowiednich pozycjach do konstruktora (dla porównania konstruktor `tuple` przyjmuje pojedynczy obiekt iterowalny).

❸ Możemy uzyskać dostęp do pól na podstawie ich nazwy lub położenia.

Typ krotki nazwanej ma parę atrybutów dodatkowych oprócz odziedziczonych z typu `tuple`. Przykład 2-10 przedstawia najbardziej przydatne: atrybut klasy `_fields`, metodę klasy `_make(iterable)` oraz metodę instancji `_asdict()`.

Przykład 2-10 *Atrybuty i metody krotek nazwanych (kontynuacja poprzedniego przykładu)*

```
>>> City._fields  ❶
('name', 'country', 'population', 'coordinates')
>>> LatLong = namedtuple('LatLong', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935, LatLong(28.613889, 77.208889))
>>> delhi = City._make(delhi_data)  ❷
>>> delhi._asdict()  ❸
OrderedDict([('name', 'Delhi NCR'), ('country', 'IN'), ('population',
21.935), ('coordinates', LatLong(lat=28.613889, long=77.208889))])
>>> for key, value in delhi._asdict().items():
```

```
        print(key + ':', value)
name: Delhi NCR
country: IN
population: 21.935
coordinates: LatLong(lat=28.613889, long=77.208889)
```

❶ `_fields` to krotka z nazwami pól klasy.

❷ `_make()` pozwala na utworzenie instancji krotki nazwanej z obiektu iterowalnego. Tak samo działa `City(*delhi_data)`.

❸ `_asdict()` zwraca słownik `collections.OrderedDict` zbudowany z instancji krotki nazwanej. Korzystając z niego, możemy ładnie wyświetlić dane dotyczące miasta.

Teraz, kiedy poznaliśmy siłę krotek jako rekordów, możemy rozważyć ich drugą rolę: niezmienną odmianę typu `list`.

Krotki jako niezmienne listy

Kiedy używamy typu `tuple` jako niezmiennej odmiany typu `list`, dobrze znać ich podobieństwo. Jak możemy zobaczyć w tabeli 2-1, krotka `tuple` obsługuje wszystkie metody listy `list`, które nie obejmują dodawania ani usuwania elementów, z jednym wyjątkiem – krotki nie mają metody `__reversed__`. Jednak służy to jedynie optymalizacji. Funkcja `reversed(my_tuple)` działa i bez tego.

Tabela 2-1 *Metody i atrybuty listy i krotki (metody implementowane przez typ object zostały pominięte dla zwięzłości)*

	list	tuple	
`s.__add__(s2)`	•	•	`s + s2` – konkatenacja
`s.__iadd__(s2)`	•		`s += s2` – konkatenacja w miejscu
`s.append(e)`	•		Dodanie jednego elementu po ostatnim
`s.clear()`	•		Usunięcie wszystkich elementów
`s.__contains__(e)`	•	•	`e in s`
`s.copy()`	•		Płytka kopia listy
`s.count(e)`	•	•	Zliczenie wystąpień elementu
`s.__delitem__(p)`	•		Usunięcie elementu z pozycji `p`
`s.extend(it)`	•		Dodanie elementów z iterowalnego `it`
`s.__getitem__(p)`	•	•	`s[p]` – pobranie elementu z pozycji
`s.__getnewargs__()`		•	Obsługa zoptymalizowanej serializacji z modułu `pickle`

Tabela 2-1 *Metody i atrybuty listy i krotki (metody implementowane przez typ object zostały pominięte dla zwięzłości)*

	list	tuple	
`s.index(e)`	•	•	Znajdowanie położenia pierwszego wystąpienia `e`
`s.insert(p, e)`	•		Wstawienie elementu `e` przed elementem na pozycji `p`
`s.__iter__()`	•	•	Pobranie iteratora
`s.__len__()`	•	•	`len(s)` – liczba elementów
`s.__mul__(n)`	•	•	`s * n` – konkatenacja wielokrotna
`s.__imul__(n)`	•		`s *= n` – konkatenacja wielokrotna w miejscu
`s.__rmul__(n)`	•	•	`n * s` – odwrócona konkatenacja wielokrotna[a]
`s.pop([p])`	•		Usunięcie i zwrócenie ostatniego elementu lub elementu na opcjonalnej pozycji `p`
`s.remove(e)`	•		Usunięcie pierwszego wystąpienia elementu `e` według wartości
`s.reverse()`	•		Odwrócenie kolejności elementów w miejscu
`s.__reversed__()`	•		Pobranie iteratora do skanowania elementów od ostatniego do pierwszego
`s.__setitem__(p, e)`	•		`s[p] = e` – umieszczenie `e` w pozycji `p` z nadpisaniem istniejącego elementu
`s.sort([key], [reverse])`	•		Sortowanie elementów w miejscu z opcjonalnymi argumentami słów kluczowych `key` i `reverse`

a Operatory odwrócone są wyjaśnione w rozdziale 13.

Każdy programista Pythona wie, że można wycinać sekwencje przy użyciu składni `s[a:b]`. Teraz zajmiemy się niektórymi mniej znanymi faktami dotyczącymi wycinania.

Wycinanie

Wspólną funkcjonalnością typów `list`, `tuple`, `str` i wszystkich typów sekwencyjnych w Pythonie jest obsługa operacji wycinania, które są znacznie potężniejsze, niż uważa większość osób.

W tym podrozdziale opiszemy *używanie* tych zaawansowanych form wycinania. Ich implementacja w klasach definiowanych przez użytkownika zostanie zawarta w rozdziale 10, zgodnie z naszą filozofią zajmowania się gotowymi do użycia klasami w tej części książki, a tworzeniu nowych klas w części IV.

Dlaczego wycinki i zakresy wykluczają ostatni element

Pythoniczna konwencja wykluczania ostatniego elementu w wycinkach i zakresach działa dobrze z indeksowaniem rozpoczynającym się od zera używanym w Pythonie, C i wielu innych językach. Oto niektóre wygodne cechy tej konwencji:

- Łatwo jest zobaczyć długość wycinka lub zakresu, gdy jest podana tylko pozycja końcowa: zarówno `range(3)`, jak i `my_list[:3]` wytwarzają trzy elementy.
- Łatwo jest obliczyć długość wycinka lub zakresu, kiedy podane są pozycje początkowa i końcowa: wystarczy odjąć `stop - start`.
- Ławo podzielić sekwencję na dwie części w miejscu oznaczonym dowolnym indeksem `x`, bez nakładania: po prostu bierzemy `my_list[:x]` i `my_list[x:]`. Na przykład:

  ```
  >>> l = [10, 20, 30, 40, 50, 60]
  >>> l[:2]  # podział w
  [10, 20]
  >>> l[2:]
  [30, 40, 50, 60]
  >>> l[:3]  # podział w 3
  [10, 20, 30]
  >>> l[3:]
  [40, 50, 60]
  ```

Jednak najlepsze argumenty popierające tę konwencję napisał holenderski informatyk Edsger W. Dijkstra (zobacz ostanie odwołanie w podrozdziale *Lektura uzupełniająca*).

Przyjrzyjmy się teraz bliżej interpretacji Pythona dla notacji wycinania.

Obiekty wycinków

Nie jest to tajemnicą, ale warto powtórzyć na wszelki wypadek: `s[a:b:c]` może służyć do określania kroku `c`, co sprawia, że wynikowy wycinek pomija elementy. Krok może być również ujemny, powodując zwracanie elementów w odwrotnej kolejności. Wyjaśnię to na trzech przykładach:

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
```

```
'elcycib'
>>> s[::-2]
'eccb'
```

Inny przykład został pokazany w rozdziale 1, gdzie użyliśmy wycinka `deck[12::13]` do pobrania wszystkich asów z niepotasowanej talii kart:

```
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A', suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Notacja `a:b:c` jest prawidłowa tylko wewnątrz `[]`, kiedy jest używana jako operator indeksujący, i wytwarza obiekty wycinków: `slice(a, b, c)`. Jak zobaczymy w podrozdziale „Działanie wycinania" w rozdziale 10, w celu przetworzenia wyrażenia `seq[start:stop:step]`, Python wywołuje `seq.__getitem__(slice(start, stop, step))`. Nawet jeśli nie implementujemy własnych typów sekwencyjnych, wiedza o obiektach wycinków jest przydatna, ponieważ pozwala przypisywać wycinkom nazwy, podobnie jak arkusze kalkulacyjne pozwalają nazywać zakresy komórek.

Załóżmy, że potrzebujemy przetworzyć dane płaskiego pliku, jak w rachunku pokazanym w przykładzie 2-11. Zamiast wypełniać kod sztywno zapisanymi wycinkami, możemy je nazwać. Zobacz, jaką czytelność możemy osiągnąć w pętli `for` na końcu tego przykładu.

Przykład 2-11 *Elementy wierszy rachunku w prostym pliku*

```
>>> invoice = """
... 0.....6................................40........52...55........
... 1909  Pimoroni PiBrella                    $17.50    3    $52.5
... 1489  6mm Tactile Switch x20                $4.95    2     $9.9
... 1510  Panavise Jr. - PV-201                $28.00    1    $28.
... 1601  PiTFT Mini Kit 320x240               $34.95    1    $34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY =  slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
    $17.50   Pimoroni PiBrella
     $4.95   6mm Tactile Switch x
```

```
    $28.00   Panavise Jr. - PV-201
    $34.95   PiTFT Mini Kit 320x24
```

Powrócimy do obiektów `slice` podczas omawiania własnych kolekcji w podrozdziale „Vector podejście nr 2: sekwencja z możliwością wycinania" (rozdział 10). Tymczasem, z perspektywy użytkownika, wycinanie zawiera dodatkowe funkcjonalności, takie jak wielowymiarowe wycinki i notacja z zastosowaniem wielokropków (`...`). Czytaj dalej.

Wycinanie wielowymiarowe i wielokropki

Operator `[]` może brać także wielowymiarowe indeksy lub wycinki rozdzielane przecinkami. Jest to używane na przykład w zewnętrznym pakiecie NumPy, gdzie elementy z dwuwymiarowej tablicy `numpy.ndarray` mogą być przechwytywane przy użyciu składni `a[i, j]`, a dwuwymiarowy wycinek pozyskany za pomocą takiego wyrażenia, jak `a[m:n, k:l]`. Przykład 2-22 dalej w tym rozdziale pokaże użycie tej notacji. Metody specjalne `__getitem__` i `__setitem__`, które obsługują operator `[]`, po prostu odbierają indeksy w `a[i, j]` jako krotkę. Innymi słowy, aby odczytać `a[i, j]`, Python wywołuje `a.__getitem__((i, j))`.

Wbudowane typy sekwencji w Pythonie są jednowymiarowe, więc mogą obsługiwać tylko jeden indeks lub wycinek, a nie ich krotkę.

Wielokropek – zapisany jako trzy kropki (`...`), a nie … (kod Unicode U+2026) – jest rozpoznawany przez parser Pythona jako token. Jest aliasem dla obiektu `Ellipsis`, pojedynczej instancji klasy `ellipsis`[2]. Jako taki może być przekazywany jako argument do funkcji oraz jako część specyfikacji wycinka, jak w `f(a, ..., z)` lub `a[i:...]`. W module NumPy wielokropek `...` jest używany jako skrót podczas wycinania tablic o wielu wymiarach. Jeśli na przykład `x` to tablica czterowymiarowa, `x[i, ...]` jest skrótem dla `x[i, :, :, :,]`. Zobacz samouczek *Tentative NumPy Tutorial* [Próbny tutorial NumPy] (*http://wiki.scipy.org/Tentative_NumPy_Tutorial*), aby dowiedzieć się więcej na ten temat.

Podczas pisania tej książki nie byłem zorientowany w zastosowaniu `Ellipsis` lub indeksów i wycinków wielowymiarowych w bibliotece standardowej Pythona. Jeśli natkniesz się na nie, daj mi znać. Te funkcjonalności składniowe istnieją, aby obsługiwać typy definiowane przez użytkownika i rozszerzenia, takie jak NumPy.

Wycinki nie są jedynie przydatne do wyodrębniania informacji z sekwencji. Możemy ich także używać do modyfikacji zmiennych sekwencji w miejscu – czyli bez budowania ich od początku.

---

2 Nie pomyliłem się: nazwa klasy `ellipsis` naprawdę ma same małe litery, a jej instancja to wbudowany obiekt o nazwie `Ellipsis`, podobnie jak typ `bool` jest zapisywany małymi literami, ale jego instancje to `True` i `False`.

Przypisywanie do wycinków

Możemy wszczepiać lub usuwać fragmenty sekwencji zmiennych lub w inny sposób modyfikować je w miejscu przy użyciu notacji wycinka po lewej stronie polecenia przypisania lub jako celu polecenie `del`. Następne parę przykładów pozwoli poznać siłę tej notacji:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100  ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

❶ Kiedy cel przypisania jest wycinkiem, po prawej stronie musi być obiekt iterowalny, nawet wtedy, gdy ma tylko jeden element.

Wszyscy wiedzą że konkatenacja jest częstą operacją na sekwencjach dowolnego typu. Dowolny wstępny tekst na temat Pythona objaśnia użycie `+` i `*` w tym celu, ale jest parę subtelnych szczegółów ich działania, którymi zajmiemy się dalej.

Używanie + i * z sekwencjami

Programiści Pythona oczekują obsługi `+` i `*` przez sekwencje. Zwykle oba operandy operatora `+` muszą być tego samego typu sekwencyjnego, a żaden z nich nie jest modyfikowany. W wyniku konkatenacji tworzona jest nowa sekwencja tego samego typu.

W celu konkatenacji wielu kopii tej samej sekwencji mnożymy ją przez liczbę całkowitą. Również w tym przypadku tworzona jest nowa sekwencja:

```
>>> l = [1, 2, 3]
>>> l * 5
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

Zarówno operator +, jak i * zawsze tworzą nowy obiekt i nigdy nie zmieniają swoich operandów.

Trzeba uważać na takie wyrażenia, jak `a * n`, gdy `a` jest sekwencją zawierającą zmienne elementy, ponieważ wynik może być zaskakujący. Na przykład próba zainicjowania listy list jako `my_list = [[]] * 3` da w wyniku listę trzech odwołań do tej samej listy wewnętrznej, co prawdopodobnie nie jest zgodne z naszymi zamierzeniami.

W następnym podrozdziale opisane są pułapki związane z próbami używania * do inicjowania listy list.

Budowanie listy list

Czasami potrzebujemy zainicjować listę z pewną liczbą list zagnieżdżonych – na przykład, aby rozmieścić studentów na liście zespołów lub reprezentować kwadratowe pola na planszy do gry. Najlepszym sposobem wykonania tego jest wyrażenie listowe, jak w przykładzie 2-12.

Przykład 2-12 *Lista trzech list o długości 3 może reprezentować planszę do gry w kółko i krzyżyk*

```
>>> board = [['_'] * 3 for i in range(3)]  ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X'  ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

❶ Tworzenie trzech list z trzema elementami w każdej. Inspekcja struktury.

❷ Umieszczenie krzyżyka w wierszu 1, kolumnie 2 i sprawdzenie wyniku.

Kuszącym, ale niewłaściwym skrótem jest wykonanie tego jak w przykładzie 2-13.

Przykład 2-13 *Lista trzech odwołań do tej samej listy jest bezużyteczna*

```
>>> weird_board = [['_'] * 3] * 3  ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
```

```
>>> weird_board[1][2] = 'O' (2)
>>> weird_board
[['_', '_', 'O'], ['_', '_', 'O'], ['_', '_', 'O']]
```

❶ Zewnętrzna lista jest złożona z trzech odwołań do tej samej listy wewnętrznej. Kiedy jest niezmieniona, wygląda dobrze.

❷ Umieszczenie kółka w wierszu 1, kolumnie 2, pokazuje, że wszystkie wiersze są aliasami odwołującymi się do tego samego obiektu.

Problem z przykładu 2-13 polega na tym, że istotnie działa zgodnie z poniższym kodem:

```
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row)  ❶
```

❶ Ten sam obiekt `row` (wiersz) jest dopisany trzy razy do obiektu `board` (plansza).

Z drugiej strony wyrażenie listowe z przykładu 2-12 jest równoważne następującemu kodowi:

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3  ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board  ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

❶ Każda iteracja buduje nowy obiekt `row` i dopisuje go do obiektu `board`.

❷ Zgodnie z oczekiwaniami tylko wiersz 2 jest zmieniany.

Jeśli problem lub rozwiązanie przedstawione w tym podrozdziale nie są jasne, odpręż się. Rozdział 8 powstał w celu objaśnienia mechanizmów i pułapek dotyczących odwołań i zmiennych obiektów.

Do tej pory omówiłem użycie czystych operatorów + i * z sekwencjami, ale istnieją jeszcze operatory += i *=, których działanie może być bardzo różne w zależności od zmienności sekwencji docelowej. Objaśnię to w kolejnym podrozdziale.

Przypisanie złożone w przypadku sekwencji

Złożone operatory przypisania += i *= zachowują się bardzo różnie w zależności od pierwszego operandu. Aby uprościć opis, skupimy się najpierw na złożonym dodawaniu (+=), ale koncepcje te dotyczą także *= i innych operatorów przypisania złożonego.

Metoda specjalna, która umożliwia działanie operatora +=, to `__iadd__` (skrót od „in-place addition” – dodawanie w miejscu). Jeśli jednak metoda `__iadd__` nie jest zaimplementowana, Python ucieka się do wywołania metody `__add__`. Rozważmy proste wyrażenie:

```
>>> a += b
```

Jeśli `a` implementuje `__iadd__`, ta metoda zostanie wywołana. W przypadku zmiennych sekwencji (np. `list`, `bytearray`, `array.array`), obiekt `a` zostanie zmieniony w miejscu (tj. efekt będzie podobny do `a.extend(b)`). Jeśli jednak obiekt `a` nie implementuje metody`__iadd__`, wyrażenie `a += b` ma taki sam efekt, jak `a = a + b`: wyrażenie `a + b` jest najpierw przetwarzane, tworząc nowy obiekt, który jest następnie wiązany z `a`. Innymi słowy tożsamość obiektu związanego z `a` może, ale nie musi się zmienić. Zależy to od dostępności metody `__iadd__`.

W ogólności dla zmiennych sekwencji dobrze będzie założyć, że metoda `__iadd__` jest zaimplementowana i operacja += odbywa się w miejscu. W przypadku sekwencji niezmiennych jest to oczywiście niemożliwe.

To, co napisałem wcześniej o +=, dotyczy także operatora *=, który jest implementowany przez metodę `__imul__`. Metody specjalne `__iadd__` i `__imul__` są opisane w rozdziale 13.

Oto demonstracja operatora *= użytego z sekwencjami zmienną i niezmienną:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800  ❶
>>> l *=
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
4311953800  ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568  ❸
>>> t *=
>>> id(t)
4301348296  ❹
```

❶ Identyfikator początkowej listy

❷ Po przemnożeniu lista jest tym samym obiektem z dopisanymi nowymi elementami

❸ Identyfikator początkowej krotki

❹ Po przemnożeniu została utworzona nowa krotka

Wielokrotna konkatenacja niezmiennych sekwencji jest niewydajna, ponieważ zamiast po prostu dopisywać nowe elementy, interpreter musi kopiować całą sekwencję docelową, aby utworzyć nową z nowymi elementami dodanymi za pomocą konkatenacji[3].

Zobaczyliśmy popularne przypadki użycia operatora +=. Następny podrozdział pokaże intrygujący przypadek niszowy, który podkreśli, co naprawdę oznacza „niezmienność" w kontekście krotek.

Zagadkowe przypisywanie +=

Spróbuj odpowiedzieć bez użycia konsoli: jaki jest wynik przetwarzania dwóch wyrażeń z przykładu 2-14?[4]

Przykład 2-14 *Zagadka*

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
```

Co stanie się dalej? Wybierz najlepszą odpowiedź:

a. `t` stanie się `(1, 2, [30, 40, 50, 60])`.

b. Wystąpi błąd `TypeError` z komunikatem `'tuple' object does not support item assignment` (obiekt nie obsługuje przypisania elementu)

c. Żadne z powyższych.

d. Zarówno **a** i **b**.

Gdy to zobaczyłem, byłem całkiem przekonany, że odpowiedzią jest **b**, ale faktycznie jest to **d**: „**a** i **b**"! Przykład 2-15 przestawia rzeczywiste wyniki z konsoli Python 3.4 (faktycznie wynik jest taki sam w konsoli Python 2.7).[5]

---

3 Wyjątkiem od tego opisu jest typ `str`. Ponieważ budowanie łańcuchów za pomocą `+=` w pętlach jest tak często spotykane w praktyce, CPython jest zoptymalizowany pod kątem tego zastosowania. Instancje `str` są umieszczane w pamięci z zapasowym wolnym miejscem, więc konkatenacja nie wymaga kopiowania całego łańcucha za każdym razem.

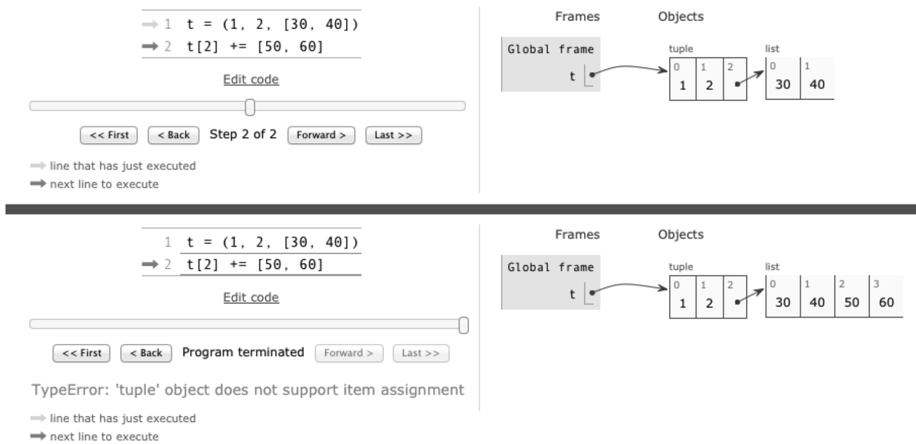
4 Leonardo Rochael i Cesar Kawakami podzielili się tą zagadką na konferencji PythonBrasil 2013. Dziękuję.

5 Jeden z czytelników wersji wstępnej książki zasugerował, że operacja w tym przykładzie może zostać wykonana bez zgłaszania błędów za pomocą kodu `t[2].extend([50,60])`. Wiem o tym, ale celem przykładu jest omówienie osobliwego działania operatora `+=`.

Przykład 2-15 *Nieoczekiwany wynik: element t2 został zmieniony oraz został zgłoszony wyjątek*

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

Online Python Tutor (*http://www.pythontutor.com/*) to świetne narzędzie online służące do szczegółowej wizualizacji działania Pythona. Rysunek 2-3 składa się z dwóch zrzutów ekranu początkowego i końcowego stanu krotki `t` z przykładu 2-15.



Rysunek 2-3 *Początkowy i końcowy stan zagadkowego przypisania do krotki (diagram wygenerowany przy użyciu narzędzia Online Python Tutor)*

Jeśli spojrzymy na kod bajtowy generowany przez Pythona dla wyrażenia `s[a] += b` (przykład 2-16), działanie staje się jasne.

Przykład 2-16 *Kod bajtowy wyrażenia* `s[a] += b`

```
>>> dis.dis('s[a] += b')
  1           0 LOAD_NAME                0 (s)
              3 LOAD_NAME                1 (a)
              6 DUP_TOP_TWO
              7 BINARY_SUBSCR                     ❶
              8 LOAD_NAME                2 (b)
             11 INPLACE_ADD                       ❷
             12 ROT_THREE
```

```
13 STORE_SUBSCR                        ❸
14 LOAD_CONST               0 (None)
17 RETURN_VALUE
```

❶ Umieść wartość `s[a]` na wierzchu stosu (`TOS`).

❷ Wykonaj `TOS += b`. To kończy się sukcesem, jeśli `TOS` odnosi się do zmiennego obiektu (w przykładzie 2-15 jest to lista).

❸ Przypisz `s[a] = TOS`. To kończy się niepowodzeniem, jeśli obiekt `s` jest niezmienny (krotka `t` w przykładzie 2-15).

Ten przykład jest dość niszowym przypadkiem – przez 15 lat używania Pythona nigdy nie widziałem, aby to dziwne zachowanie faktycznie komuś zaszkodziło.

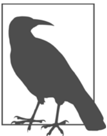
Wynikają stąd trzy lekcje:

- Umieszczanie zmiennych elementów w krotkach nie jest dobrym pomysłem.
- Przypisanie złożone nie jest operacją atomową – właśnie zobaczyliśmy, że zgłasza wyjątek po wykonaniu części swojego zadania.
- Inspekcja kodu bajtowego Pythona nie jest zbyt trudna, a często przydaje się, aby zobaczyć co się dzieje wewnątrz.

Po uświadomieniu sobie subtelności używania operatorów `+` i `*` do konkatenacji, możemy zmienić temat na inną istotną operację na sekwencjach: sortowanie.

Metoda list.sort oraz wbudowana funkcja sorted

Metoda `list.sort` sortuje listę w miejscu – czyli bez wykonywania kopii. Zwraca `None`, aby przypomnieć nam, że zmienia obiekt docelowy i nie tworzy nowej listy. Jest to ważna konwencja interfejsu API Pythona: funkcje lub metody, które zmieniają obiekt w miejscu, powinny zwracać `None`, aby kod wywołujący miał jasność, że został zmieniony ten sam obiekt i nie utworzono nowego. To samo działanie możemy zobaczyć np. w funkcji `random.shuffle`.



Konwencja zwracania `None`, aby sygnalizować zmiany w miejscu, ma wadę: nie można wykonać kaskady wywołań takich metod. Natomiast metody, które zwracają nowe obiekty (np. wszystkie metody `str`), mogą być wywoływane kaskadowo w stylu płynnego interfejsu. Zobacz dalszy opis tego tematu we wpisie „Fluent interface" w Wikipedii (*http://en.wikipedia.org/wiki/Fluent_interface*).

Natomiast wbudowana funkcja `sorted` tworzy nową listę i ją zwraca. Funkcja ta przyjmuje jako argument rzeczywiście dowolny obiekt iterowalny, w tym niezmienne sekwencje

i generatory (zobacz rozdział 14). Bez względu na typ obiektu iterowalnego przekazanego do funkcji `sorted`, zawsze zwraca ona nowo utworzoną listę.

Zarówno metoda `list.sort`, jak i funkcja `sorted` przyjmują dwa opcjonalne argumenty w postaci słów kluczowych:

`reverse`
: Jeśli ma wartość `True`, elementy są zwracane w kolejności malejącej (tj. przez odwrócenie porównania elementów). Domyślną wartością jest `False`.

`key`
: Jednoargumentowa funkcja, która zostanie zastosowana do każdego elementu, aby wytworzyć klucz sortowania. Na przykład podczas sortowania listy łańcuchów `key=str.lower` może służyć do wykonania sortowania bez względu na wielkość liter, a `key=len` spowoduje posortowanie łańcuchów według długości w znakach. Domyślną wartością jest funkcja tożsamości (tj. porównywane są same elementy).

Parametr opcjonalnego słowa kluczowego key może zostać użyty z wbudowanymi funkcjami `min()` i `max()` oraz z innymi funkcjami z biblioteki standardowej (np. `itertools.groupby()` i `heapq.nlargest()`).

Oto parę przykładów dla objaśnienia użycia tych funkcji i argumentów w postaci słów kluczowych[6]:

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry']  ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana']  ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple']  ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry']  ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple']  ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana']  ❻
>>> fruits.sort()                          ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry']  ❽
```

6 Przykłady demonstrują również, że Timsort – algorytm sortowania użyty w Pythonie – jest stabilny (tj. zachowuje względną kolejność elementów, które w porównaniu są równe). Algorytm Timsort jest opisany dalej w ramce *Pogadanka* na końcu tego rozdziału.

❶ Wytwarza nową listę łańcuchów posortowanych alfabetycznie.

❷ Inspekcja oryginalnej listy, widzimy, że się nie zmieniła.

❸ To jest proste odwrotne sortowanie alfabetyczne.

❹ Nowa lista łańcuchów, teraz posortowana według długości. Ponieważ algorytm sortowania jest stabilny, „grape" i „apple" o długości 5 mają zachowaną oryginalną kolejność.

❺ To są łańcuchy posortowane w malejącej kolejności długości. Nie jest to odwrotność poprzedniego wyniku, ponieważ z powodu stabilności sortowania znowu „grape" występuje przed „apple."

❻ Do tej pory kolejność oryginalnej listy `fruits` nie zmieniła się.

❼ To sortowanie odbywa się w miejscu i zwraca `None` (konsola pomija ten wynik).

❽ Teraz lista `fruits` jest posortowana.

Raz posortowane sekwencje mogą być bardzo wydajnie przeszukiwane. Na szczęście w bibliotece standardowej Pythona mamy gotowy algorytm wyszukiwania binarnego zawarty w module `bisect`. Dalej omówimy jego istotne funkcje, w tym wygodną funkcję `bisect.insort`, której możemy użyć, aby upewnić się, że posortowane sekwencje pozostają posortowane.

Zarządzanie sekwencjami uporządkowanymi przy użyciu bisect

Moduł `bisect` oferuje dwie główne funkcje – `bisect` oraz `insort` – które używają algorytmu wyszukiwania binarnego, aby szybko znajdować i wstawiać elementy w dowolnej posortowanej sekwencji.

Wyszukiwanie za pomocą funkcji bisect

Funkcja `bisect(haystack, needle)` wykonuje binarnego wyszukiwania igły `needle` w stogu siana `haystack` – który musi być posortowaną sekwencją – aby zlokalizować miejsce, gdzie można wstawić `needle` z zachowaniem rosnącego uporządkowania obiektu `haystack`. Innymi słowy wszystkie elementy pojawiające się powyżej tej pozycji są mniejsze lub równe `needle`. Możemy użyć wyniku `bisect(haystack, needle)` jako argumentu `index` metody `haystack.insert(index, needle)` – jednak użycie `insort` wykonuje oba kroki i jest szybsze.

Raymond Hettinger – płodny współtwórca Pythona – utworzył przepis `SortedCollection` (*http://bit.ly/1Vm6WEa*), który korzysta z modułu `bisect`, ale jest łatwiejszy w użyciu niż te samodzielne funkcje.

W przykładzie 2-17 używamy ostrożnie dobranego zbioru „igieł”, aby zademonstrować pozycje wstawiania zwracane przez `bisect`. Jego wyniki są pokazane na rysunku 2-4 z użyciem bisect – każdy wiersz zaczyna się od notacji needle @ position (igła @ pozycja), a wartości „igieł” pojawiają się ponownie poniżej punktu wstawienia do „stogu siana”.

```
02-array-seq/ $ python3 bisect_demo.py
DEMO: bisect
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |31
30 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |30
29 @ 13      |  |  |  |  |  |  |  |  |  |  |  |  |29
23 @ 11      |  |  |  |  |  |  |  |  |  |  |23
22 @  9      |  |  |  |  |  |  |  |  |22
10 @  5      |  |  |  |  |10
 8 @  5      |  |  |  |  |8
 5 @  3      |  |  |5
 2 @  1      |2
 1 @  1      |1
 0 @  0    0
```

Rysunek 2-4 *Wyniki przykładu 2-17 z użyciem bisect – każdy wiersz zaczyna się od notacji needle @ position, a wartość needle pojawia się znowu poniżej punktu wstawienia w sekwencji haystack*

Przykład 2-17 *bisect znajduje punkty wstawiania dla elementów w posortowanej sekwencji*

```
import bisect
import sys
HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]
ROW_FMT = '{0:2d} @ {1:2d}    {2}{0:<2d}'
def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle)  ❶
        offset = position * '  |'  ❷
        print(ROW_FMT.format(needle, position, offset))  ❸
if __name__ == '__main__':
    if sys.argv[-1] == 'left':    ❹
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect.bisect
    print('DEMO:', bisect_fn.__name__)  ❺
    print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))
    demo(bisect_fn)
```

❶ Użycie wybranej funkcji `bisect` do pobrania punktu wstawienia.

❷ Zbudowanie wzorca pionowych przedziałów proporcjonalnych do `offset`.

❸ Wyświetlenie sformatowanego wiersza pokazującego wartość `needle` i punkt wstawiania.

❹ Wybranie funkcji `bisect` do użycia zgodnie z ostatnim argumentem wiersza polecenia.

❺ Wydrukowanie nagłówka z nazwą wybranej funkcji.

To zachowanie funkcji `bisect` może być dopracowane na dwa sposoby.

Po pierwsze para argumentów opcjonalnych, `lo` i `hi`, pozwala zawęzić region w sekwencji do przeszukiwania podczas wstawiania. Domyślną wartością `lo` jest 0, a `hi` – długość `len()` sekwencji.

Po drugie `bisect` jest faktycznie aliasem funkcji `bisect_right`, która ma bliźniaczą funkcję `bisect_left`. Różnica między nimi jest widoczna tylko wtedy, gdy porównywana igła jest równa elementowi z listy: `bisect_right` zwraca punkt wstawiania po istniejącym elemencie, a `bisect_left` zwraca położenie istniejącego elementu, więc punkt wstawiania wystąpi przed tym elementem. Przy prostych typach, jak `int`, nie robi to różnicy, ale jeśli sekwencja zawiera obiekty, które są różne, ale w porównaniu równe, może to mieć znaczenie. Na przykład `1` i `1.0` są różne, ale `1 == 1.0` to `True`. Rysunek 2-5 przedstawia wyniki użycia `bisect_left`.

```
02-array-seq/ $ python3 bisect_demo.py left
DEMO: bisect_left
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  |  |  |31
30 @ 13      |  |  |  |  |  |  |  |  |  |  |  |  |30
29 @ 12      |  |  |  |  |  |  |  |  |  |  |  |29
23 @  9      |  |  |  |  |  |  |  |  |23
22 @  9      |  |  |  |  |  |  |  |  |22
10 @  5      |  |  |  |  |10
 8 @  4      |  |  |  |8
 5 @  2      |  |5
 2 @  1      |2
 1 @  0     1
 0 @  0     0
```

Rysunek 2-5 *Wynik przykładu 2-17 z użyciem bisect_left (porównaj z rysunkiem 2-4 z użyciem bisect – każdy wiersz zaczyna się od notacji needle @ position, a wartość needle pojawia się ponownie poniżej punktu wstawiania do sekwencji haystack – i zwróć uwagę na punkty wstawiania wartości 1, 8, 23, 29 i 30 po lewej stronie tych samych liczb w sekwencji haystack).*

Interesującym zastosowaniem funkcji `bisect` jest wykonanie przeszukiwania tabeli według wartości liczbowych – na przykład, aby konwertować wyniki testu na oceny w formie liter, jak w przykładzie 2-18.

Przykład 2-18 *Dla danego wyniku testu funkcja grade zwraca odpowiedną ocenę w postaci litery*

```
>>> def grade(score, breakpoints=[60, 70, 80, 90], grades='FDCBA'):
...     i = bisect.bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [33, 99, 77, 70, 89, 90, 100]]
['F', 'A', 'C', 'C', 'B', 'A', 'A']
```

Kod w przykładzie 2-18 pochodzi z dokumentacji modułu `bisect` (*https://docs.python.org/3/library/bisect.html*), w której są wymienione także funkcje do używania `bisect` jako szybszej alternatywy dla metody `index` podczas wyszukiwania w długich uporządkowanych sekwencjach liczb.

Te funkcje nie tylko służą do wyszukiwania, ale także do wstawiania elementów w posortowanych sekwencjach, co można zobaczyć w kolejnym podrozdziale.

Wstawianie za pomocą funkcji bisect.insort

Sortowanie jest kosztowne, więc gdy już raz mamy posortowaną sekwencję, dobrze pozostawić ją w takim stanie. Dlatego powstała funkcja `bisect.insort`.

Funkcja `insort(seq, item)` wstawia element `item` do sekwencji `seq`, utrzymując sekwencję `seq` w kolejności rosnącej. Zobacz przykład 2-19 i jego wyniki na rysunku 2-6.

Przykład 2-19 *Insort zachowuje posortowaną sekwencję zawsze posortowaną*

```
import bisect
import random
SIZE = 7
random.seed(1729)
my_list = []
for i in range(SIZE):
    new_item = random.randrange(SIZE*2)
    bisect.insort(my_list, new_item)
    print('%2d ->' % new_item, my_list)
```

```
02-array-seq/ $ python3 bisect_insort.py
10 -> [10]
 0 -> [0, 10]
 6 -> [0, 6, 10]
 8 -> [0, 6, 8, 10]
 7 -> [0, 6, 7, 8, 10]
 2 -> [0, 2, 6, 7, 8, 10]
10 -> [0, 2, 6, 7, 8, 10, 10]
```

Rysunek 2-6 *Wyniki przykładu 2-19*

Podobnie jak funkcja `bisect`, funkcja `insort` przyjmuje opcjonalne argumenty `lo` i `hi`, aby ograniczyć wyszukiwanie do podsekwencji. Istnieje także odmiana `insort_left`, która używa `bisect_left` do znajdowania punktów wstawiania.

Większość tego, co widzieliśmy do tej pory w tym rozdziale dotyczy sekwencji w ogólności, a nie tylko list czy krotek. Programiści Pythona czasami nadużywają typu `list`, ponieważ jest tak poręczny – wiem, też tak robiłem. Jeśli jednak mamy do czynienia z listami liczb, lepszym sposobem będą tablice. Pozostała część tego rozdziału jest im poświęcona.

Kiedy lista nie jest rozwiązaniem

Typ `list` jest elastyczny i łatwy w użyciu, ale w zależności od specyficznych wymagań możemy znaleźć lepsze opcje. Jeśli na przykład potrzebujemy przechować 10 milionów wartości zmiennoprzecinkowych, tablica `array` jest znacznie bardziej wydajna, ponieważ typ `array` nie przechowuje faktycznie pełnowartościowych obiektów `float`, a jedynie upakowane bajty reprezentujące ich wartości maszynowe – całkiem jak tablica w języku C. Z drugiej strony, jeśli stale dodajemy i usuwamy elementy z końców listy, jak w strukturach danych FIFO lub LIFO, `deque` (kolejka dwustronna) działa szybciej.

Jeśli kod wykonuje dużo testów zawartości (np. `item in my_collection`), rozważ użycie zbioru `set` dla kolekcji `my_collection`, szczególnie jeśli zawiera ona dużą liczbę elementów. Zbiory są optymalizowane dla szybkiego sprawdzania przynależności. Jednakże nie są sekwencjami (ich zawartość jest nieuporządkowana). Zajmiemy się nimi w rozdziale 3.

W pozostałej części tego rozdziału rozważymy typy zmiennych sekwencji, które mogą zastępować listy w wielu przypadkach, zaczynając od tablic.

Tablice

Jeśli lista zawiera tylko liczby, tablica `array.array` jest bardziej wydajna niż lista `list`: obsługuje wszystkie operacje na zmiennych sekwencjach (w tym `.pop`, `.insert` i `.extend`) oraz dodatkowe metody do szybkiego ładowania i zapisywania, takie jak `.frombytes` i `.tofile`.

Tablica Pythona jest lekka jak tablica w języku C. Podczas tworzenia tablicy `array` podajemy kod typu, literę wyznaczającą podległy typ w języku C do przechowywania poszczególnych elementów w tablicy. Na przykład `b` to kod typu dla `signed char`. Jeśli utworzymy `array('b')`, każdy element zostanie przechowany w pojedynczym bajcie i interpretowany jako liczba całkowita od – 128 do 127. W przypadku większych sekwencji liczb oszczędza to wiele pamięci. A Python nie pozwoli umieścić żadnej liczby niepasującej do typu tablicy.

Przykład 2-20 przedstawia tworzenie, zapisywanie i ładowanie tablicy 10 milionów losowych liczb zmiennoprzecinkowych.

Przykład 2-20 *Tworzenie, zapisywanie i ładowanie dużej tablicy liczb zmiennoprzecinkowych*

```
>>> from array import array  ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7)))  ❷
>>> floats[-1]  ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp)  ❹
>>> fp.close()
>>> floats2 = array('d')  ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7)  ❻
>>> fp.close()
>>> floats2[-1]  ❼
0.07802343889111107
>>> floats2 == floats  ❽
True
```

❶ Importowanie typu `array`.

❷ Tworzenie tablicy liczb zmiennoprzecinkowych podwójnej precyzji (kod typu `'d'`) na podstawie dowolnego obiektu iterowalnego – w tym przypadku wyrażenia generatora.

❸ Inspekcja ostatniej liczby tablicy.

❹ Zapisanie tablicy do pliku binarnego.

❺ Utworzenie pustej tablicy liczb podwójnej precyzji.

❻ Odczytanie 10 milionów liczb z pliku binarnego.

❼ Inspekcja ostatniej liczby tablicy.

❽ Sprawdzenie, że zawartość tabel jest jednakowa.

Jak możemy zauważyć, metody `array.tofile` i `array.fromfile` są łatwe w użyciu. Jeśli wypróbujemy ten przykład, zauważymy, że są także bardzo szybkie. Szybki eksperyment pokazuje, że załadowanie 10 milionów liczb zmiennoprzecinkowych podwójnej precyzji za pomocą metody `array.fromfile` z pliku binarnego utworzonego za pomocą metody `array.tofile` zajmuje 0,1 s. Jest to prawie 60 razy szybciej niż odczytanie liczb z pliku tekstowego, co obejmuje parsowanie każdego wiersza za pomocą wbudowanej funkcji `float`. Zapisanie za pomocą metody `array.tofile` jest około 7 razy szybsze niż zapisanie po jednej liczbie zmiennoprzecinkowej na wiersz w pliku tekstowym. Ponadto rozmiar pliku binarnego z 10 milionami liczb podwójnej precyzji wynosi 80 000 000 bajtów (8

bajtów na liczbę podwójnej precyzji, zero nadmiaru), podczas gdy plik tekstowy z tymi samymi danymi zajmuje 181 515 739 bajtów.

Innym szybkim i bardziej elastycznym sposobem zapisywania danych liczbowych jest użycie modułu `pickle` (*http://bit.ly/py-pickle*) do serializacji obiektów. Zapisanie tablicy liczb zmiennoprzecinkowych za pomocą `pickle.dump` jest prawie tak szybkie, jak za pomocą `array.tofile` – jednak `pickle` obsługuje prawie wszystkie typy wbudowane, w tym liczby zespolone `complex`, zagnieżdzone kolekcje a nawet instancji definiowanych przez użytkownika klas w sposób automatyczny (o ile ich implementacja nie jest zbyt skomplikowana).

Dla szczególnego przypadku tablic liczbowych reprezentujących dane binarne, takich jak obrazy rastrowe, Python ma typy `bytes` i `bytearray` opisane w rozdziale 4.

Zakończymy ten podrozdział dotyczący tablic tabelą 2-2, zawierającą porównanie funkcjonalności typów `list` i `array.array`.

Tabela 2-2 *Metody i atrybuty znajdujące się w obiektach list lub array (przestarzałe metody tablic oraz metody implementowane również w typie object zostały pominięte dla zwięzłości)*

	list	array	
`s.__add__(s2)`	•	•	`s + s2` – konkatenacja
`s.__iadd__(s2)`	•	•	`s += s2` – konkatenacja w miejscu
`s.append(e)`	•	•	Dopisanie jednego elementu za ostatnim
`s.byteswap()`		•	Zamiana bajtów wszystkich elementów w tablicy w celu konwersji kolejności bajtów (endianess)
`s.clear()`	•		Usunięcie wszystkich elementów
`s.__contains__(e)`	•	•	`e in s`
`s.copy()`	•		Płytka kopia listy
`s.__copy__()`		•	Obsługa `copy.copy`
`s.count(e)`	•	•	Zliczenie wystąpień elementu
`s.__deepcopy__()`		•	Zoptymalizowana obsługa `copy.deepcopy`
`s.__delitem__(p)`	•	•	Usunięcie elementu z pozycji `p`
`s.extend(it)`	•	•	Dodanie elementów z iterowalnego `it`
`s.frombytes(b)`		•	Dopisanie elementów z sekwencji bajtów interpretowanej jako upakowane wartości maszynowe

Tabela 2-2 *Metody i atrybuty znajdujące się w obiektach list lub array (przestarzałe metody tablic oraz metody implementowane również w typie object zostały pominięte dla zwięzłości)*

	list	array	
`s.fromfile(f, n)`		•	Dopisanie `n` elementów z pliku binarnego `f` interpretowanego jako upakowane wartości maszynowe
`s.fromlist(l)`		•	Dopisanie elementów z listy. Jeśli któryś powoduje błąd `TypeError`, żaden nie zostanie dopisany
`s.__getitem__(p)`	•	•	`s[p]` – pobranie elementu z pozycji
`s.index(e)`	•	•	Znalezienie pozycji pierwszego wystąpienia `e`
`s.insert(p, e)`	•	•	Wstawienie elementu `e` przed elementem na pozycji `p`
`s.itemsize`		•	Długość w bajtach poszczególnych elementów tablicy
`s.__iter__()`	•	•	Pobranie iteratora
`s.__len__()`	•	•	`len(s)` – liczba elementów
`s.__mul__(n)`	•	•	`s * n` – konkatenacja wielokrotna
`s.__imul__(n)`	•	•	`s *= n` – konkatenacja wielokrotna w miejscu
`s.__rmul__(n)`	•	•	`n * s` – odwrócona konkatenacja wielokrotna[a]
`s.pop([p])`	•	•	Usunięcie i zwrócenie elementu na pozycji `p` (domyślnie: ostatniego)
`s.remove(e)`	•	•	Usunięcie pierwszego wystąpienia elementu `e` według wartości
`s.reverse()`	•	•	Odwrócenie kolejności elementów w miejscu
`s.__reversed__()`	•		Pobranie iteratora do przeskanowania elementów od ostatniego do pierwszego
`s.__setitem__(p, e)`	•	•	`s[p] = e` – umieszczenie `e` na pozycji `p`, z nadpisaniem istniejącego elementu
`s.sort([key], [reverse])`	•		Sortowanie elementów w miejscu z opcjonalnymi argumentami na podstawie słów kluczowych `key` i `reverse`
`s.tobytes()`		•	Zwrócenie elementów jako upakowanych wartości maszynowych w obiekcie `bytes`

Tabela 2-2 *Metody i atrybuty znajdujące się w obiektach list lub array (przestarzałe metody tablic oraz metody implementowane również w typie object zostały pominięte dla zwięzłości)*

	list	array	
`s.tofile(f)`		•	Zapisanie elementów jako upakowanych wartości maszynowych do binarnego pliku `f`
`s.tolist()`		•	Zwrócenie elementów jako liczbowych obiektów w obiekcie `list`
`s.typecode`		•	Jednoznakowy łańcuch identyfikujący typ elementów w języku C

a Operatory odwrócone są wyjaśnione w rozdziale 13.

W wersji Python 3.4 typ `array` nie ma metody `sort` sortującej w miejscu, takiej jak `list.sort()`. Jeśli potrzebujemy posortować tablicę, używamy funkcji `sorted`, aby zbudować ją ponownie posortowaną:

```
a = array.array(a.typecode, sorted(a))
```

Aby zachować posortowaną tablicę podczas dodawania do niej elementów, używamy funkcji `bisect.insort` (jak widzieliśmy w podrozdziale *Wstawianie za pomocą funkcji bisect.insort*).

Jeśli dużo pracujesz z tablicami, a nie znasz typu `memoryview`, wiele tracisz. Zobacz następny temat.

Widoki pamięci

Wbudowana klasa `memorview` jest typem sekwencji obejmującym współdzieloną pamięć, który pozwala obsługiwać wycinki tablic bez kopiowania bajtów. Została zainspirowana biblioteką NumPy (którą omówimy pokrótce w podrozdziale „NumPy i SciPy"). Travis Oliphant, wiodący autor NumPy, tak odpowiedział na pytanie *When should a memoryview be used?* [Kiedy należy używać memoryview]:

> Widok pamięci to istotnie uogólniona struktura tablicy NumPy w samym Pythonie (bez matematyki). Pozwala na współdzielenie pamięci między strukturami danych (takimi jak obrazy PIL, bazy danych SQLlite, tablice NumPy itp.) bez wcześniejszego kopiowania. Jest to bardzo ważne dla dużych zbiorów danych. (*http://bit.ly/1Vm6C8B*)

Korzystając z notacji podobnej do modułu `array`, metoda `memoryview.cast` pozwala zmieniać sposób odczytywania lub zapisywania wielu jako jednostek bez przenoszenia bitów (zupełnie jak operator `cast` w języku C). `memoryview.cast` zwraca jeszcze jeden obiekt `memoryview`, zawsze współdzielące tę samą pamięć.

Zobacz przykład 2-21 jako przykład zamiany pojedynczego bajta w tablicy 16-bitowych liczb całkowitych.

Przykład 2-21 *Zmienianie wartości elementu tablicy przez modyfikację jednego z jej bajtów*

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers)  ❶
>>> len(memv)
5
>>> memv[0]  ❷
-
>>> memv_oct = memv.cast('B')  ❸
>>> memv_oct.tolist()  ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4  ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2])  ❻
```

❶ Budowanie widoku pamięci `memoryview` z tablicy pięciu 2-bajtowych liczb całkowitych ze znakiem (kod typu `'h'`).

❷ `memv` widzi te same 5 elementów w tablicy.

❸ Tworzenie `memv_oct` przez rzutowanie elementów `memv` na kod typu `'B'` (jeden bajt bez znaku).

❹ Eksportowanie elementów `memv_oct` jako listy, dla inspekcji.

❺ Przypisanie wartości 4 do bajta o przesunięciu 5.

❻ Zauważ zmianę w tablicy `numbers`: 4 w najbardziej znaczącym bajcie 2-bajtowego typu całkowitoliczbowego daje w wyniku 1024.

Zobaczymy jeszcze jeden krótki przykład użycia `memoryview` w kontekście manipulacji sekwencjami binarnymi ze strukturą `struct` (rozdział 4, przykład 4-4).

Przy okazji, zajmując się zaawansowanym przetwarzaniem numerycznym na tablicach, warto korzystać z bibliotek NumPy i SciPy. Zaraz przyjrzymy się im pokrótce.

NumPy i SciPy

W całej tej książce skupiam się nad podkreśleniem tego, co jest już w bibliotece standardowej Pythona, aby można było jak najlepiej z niej korzystać. Jednak biblioteki NumPy i SciPy są tak wspaniałe, że nie mogę ich pominąć.

Dzięki zaawansowanym operacjom na tablicach i macierzach biblioteki NumPy i SciPy sprawiły, że Python znalazł się w głównym nurcie naukowych aplikacji komputerowych. Biblioteka NumPy implementuje wielowymiarowe, homogeniczne tablice i typy macierzowe, które przechowują nie tylko liczby, ale także rekordy definiowane przez użytkownika, oraz dostarcza wydajnych operacji na elementach.