Spis treści

Część II. Funkcje jako obiekty

Część III. Klasy i protokoły

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 ;-)[1]
>
> *– 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://fpy.li/p-2*). 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

1 W wiadomości wysłanej na Usenetową grupę comp.lang.python 23 grudnia 2002: „Acrimony in c.l.p" (*https://fpy.li/p-1*).

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, którzy chcą osiągnąć biegłą znajomość Pythona 3. Przykłady testowałem w Pythonie 3.10 – większość z nich również w Pythonie 3.9 i 3.8. Gdy przykład wymaga Pythona 3.10, powinno być to jasno oznaczone

Jeśli nie wiesz, czy znasz Pythona wystarczająco, aby skorzystać z tej książki, przejrzyj tematy oficjalnego samouczka (*https://fpy.li/p-3*). Tematy opisane w samouczku nie zostaną tu wyjaśnione, poza pewnymi funkcjonalnościami, które są nowościami.

Dla kogo nie jest ta książka

Jeśli po prostu uczysz się Pythona, ta książka może się okazać nazbyt 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.

Pięć książek w jednej

Zalecam, aby każdy przeczytał rozdział 1, „Model danych Pythona". Docelowi odbiorcy tej książki nie powinni mieć problemu z przeskoczeniem bezpośrednio do dowolnego rozdziału w tej książce po przeczytaniu rozdziału 1. Często jednak zakładałem, że Czytelnik przeczytał poprzedzające rozdziały danej części. O każdej z pięciu części możemy myśleć, jak o samodzielnej książce w ramach tej książki.

Próbowałem podkreślić używanie dostępnych rozwiązań przed omawianiem, jak zbudować własne. Na przykład rozdział 2 w części I 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 III, gdzie zobaczymy także, jak wykorzystać abstrakcyjne klasy bazowe (ABC) z modułu `collections.abc`. Tworzenie własnych klas ABC jest omówione jeszcze dalej w części III, 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.

Organizacja książki

Oto parę głównych tematów w każdej części tej książki:

Część I, „Struktury danych"
: Rozdział 1 wprowadza model danych Pythona i wyjaśnia, dlaczego metody specjalne (np. `__repr__`) są kluczowe dla spójnego działania obiektów wszystkich typów – w języku, który jest ceniony za swoją spójność. Metody specjalne są omawiane szczegółowo w różnych miejscach ksiażki. Pozostałe rozdziały 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. Ponadto omówione są klasy budowniczych wysokiego poziomu dostępne w bibliotece standardowej: fabryki nazwanych krotek oraz dekorator `@dataclass`. Dopasowywanie wzorców – nowość w Pythonie 3.10 – omawiane jest w rozdziałach 2, 3 i 5, w których przedyskutujemy wzorce sekwencji, wzorce odwzorowań i wzorce klas. Ostatni rozdział części I dotyczy cyklu życia obiektów: referencji, zmienności i sprzątania pamięci.

Część II, „Funkcje jako obiekty"
: 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. Rozdział 8 wprowadza obszerny nowy temat wskazówek typów w sygnaturach funkcji.

Część III, „Klasy i protokoły"
: Teraz skupimy się na „odręcznym" budowaniu klas, jako przeciwieństwie używania budowniczych klas omówionych w rozdziale 5. Podobnie jak każdy język obiektowy (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 budować własne kolekcje, abstrakcyjne klasy bazowe (ABC) i protokoły, a także jak radzić sobie z wielokrotnym dziedziczeniem i jak implementować przeciążanie operatorów – kiedy to ma sens. Rozdział 15 kontynuuje omówienie wskazówek typów.

Część IV, „Przepływ sterowania”

W tej części opisane są konstrukcje językowe i biblioteki, które wykraczają poza tradycyjny przepływ sterowania za pomocą instrukcji warunkowych, pętli i podprogramów. Zaczynamy od generatorów, następnie zajmujemy się menedżerami kontekstu i współprogramami, w tym wymagającą, ale potężną nową składnię `yield from`. Rozdział 19, „Modele współbieżności w Pythonie” to nowy rozdział, przedstawiający przegląd alternatyw dla współbieżności i przetwarzania równoległego, ich ograniczenia, a także to, jak architektura oprogramowania pozwala Pythonowi działać w skali sieci Web. Przepisałem też na nowo rozdział dotyczący *programowania asynchronicznego*, aby podkreślić podstawowe funkcjonalności języka – czyli `await`, `async dev`, `async for` oraz `async with`, a na koniec pokazać, jak można z nich korzystać w połączeniu z *asyncio* i innymi frameworkami.

Część V, „Metaprogramowanie”

Ta część zaczyna się od przeglądu technik do budowania klas z atrybutami tworzonymi dynamicznie do obsługi danych semistrukturalnych, takich jak zbiory 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 V 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* – pętla odczyt-przetwarzanie-wydruk).

Jeden ze standardowych pakietów testowych Pythona, `doctest` (*https://fpy.li/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 samodzielnie wypróbować demonstrację.

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`, zajrzyj do jego dokumentacji (*https://fpy.li/doctest*) oraz repozytorium kodu źródłowego tej książki (*https://fpy.li/code*).

Napisałem również testy jednostkowe dla części co większych przykładów, wykorzystując *pytest* – który jest w mojej opini łatwiejszy w użyciu i bardziej wydajny, niż moduł *unittest* z biblioteki standardowej. Zobaczysz, że możesz zweryfikować poprawność większości kodu w tej książce, wpisując `python3 -m doctest example_script.py` w powłoce poleceń swojego systemu operacyjnego. Plik konfiguracyjny *pytest.ini* w katalogu głównym repozytorium przykładów kodu gwarantuje, że testy zostaną zebrane i wykonane przez polecenie `pytest`.

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 każdego rozdziału znajdziesz ramki „Pogadanka" z moimi własnymi spostrzeżeniami dotyczącymi Pythona i innych języków. Możesz swobodnie pominąć te uwagi, jeśli Cię nie interesują. Ich zawartość jest całkowicie opcjonalna.

Witryna książki: fluentpython.com

Omówienie nowych funkcjonalności – takich jak wskazówki dla typów, klasy danych i dopasowywanie wzorców – sprawiło, że drugie wydanie jest niemal o 30% większe od pierwszego. Aby zapewnić poręczność tej książki, przeniosłem część treści do witryny *fluentpython.com*. Linki do publikowanych tam artykułów można znaleźć w wielu rozdziałach. W tej witrynie są też dostępne przykładowe rozdziały. Pełny tekst jest dostępny online w ramach subskrypcji O'Reilly Learning (*https://fpy.li/p-4*)[2]. Wszystkie przykłady kodu znajdują się w repozytorium tej książki na GitHubie (*https://fpy.li/code*).

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.

2 Witryna ta udostępnia naturalnie oryginalną (angielską) treść książki (przyp. tłum.).

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, a także treść komentarzy zawartych w przykładach kodu.

Ten element oznacza wskazówkę lub sugestię.

Ten element oznacza uwagę ogólną.

Ten element wskazuje ostrzeżenie lub przestrogę.

Korzystanie z przykładów kodu

Każdy skrypt i większość wycinków kodu występujących w książce są dostępne w repozytorium *Fluent Python* na GitHubie pod adresem *https://fpy.li/code*.

W przypadku pytań technicznych lub problemów z wykorzystaniem przykładów kodów należy przesłać maila na adres *bookquestions@oreilly.com*.

Celem tej książki jest wsparcie wykonywanej pracy. Ogólnie rzecz ujmując, przykładowe kody dołączone do książki można używać w programach i dokumentacji. Nie trzeba się kontaktować z wydawnictwem w celu uzyskania pozwolenia, chyba że jest wykorzystywana znaczna część kodu. Na przykład napisanie programu, który używa kilka fragmentów kodu z tej książki, nie wymaga pozwolenia. Sprzedaż lub dystrybucja przykładów z książek wydawnictwa O'Reilly wymaga już pozwolenia. Odpowiedź na pytanie poprzez zacytowanie tej książki i wykorzystanie przykładowego kodu nie wymaga pozwolenia. Włączenie znacznej ilości przykładowego kodu z tej książki do dokumentacji produktu wymaga pozwolenia.

Doceniamy uznanie autorstwa, ale nie wymagamy go. Zazwyczaj obejmuje ono tytuł, autora, wydawcę i numer ISBN. Na przykład: „*Fluent Python*, 2nd ed., by Luciano Ramalho (O'Reilly). Copyright 2022 Luciano Ramalho, 978-1-492-05635-5".

W przypadku, gdy wykorzystanie przykładów kodu może wykraczać poza dozwolony użytek lub powyżej podane zezwolenie, należy się skontaktować z wydawnictwem poprzez wysłanie maila na adres *permissions@oreilly.com*.

Jak się z nami kontaktować

Istnieje strona internetowa dotycząca tej książki, gdzie znajduje się errata, przykłady i inne dodatkowe informacje. Jej adres to *https://fpy.li/p-4*.

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

Nie spodziewałem się, że aktualizacja książki o Pythonie będzie jakimś większym wyzwaniem, ale właśnie tak było. Marta Mello, moja ukochana żona, zawsze była przy mnie, gdy jej potrebowałem. Mój przyjaciel Leonardo Rochael pomagał mi od samego początku, aż po końcowe przeglądy techniczne, w tym zbierał i podwójnie sprawdzał informacje od innych recenzentów, wczesnych czytelników i redaktorów. Mówiąc szczerze, nie umiem sobie wyobrazić, abym zdołał wykonać tę pracę bez pomocy Marty i Leo. Dziękuję wam bardzo, bardzo!

Jurgen Gmach, Caleb Hattingh, Jess Males, Leonardo Rochael i Miroslav Šedivy stworzyli wyjątkowy zespół recenzentów technicznych dla drugiego wydania. Przeczytali dokładnie całą książkę. Bill Behrman, Bruce Eckel, Renato Oliveira i Rodrigo Bernardo Pimentel dodatkowo recenzowali konkretne rozdziały. Ich sugestie z wielu różnych punktów widzenia sprawiły, że ta książka jest znacznie lepsza.

Wielu czytelników przesyłało poprawki albo wnosiło inny wkład w fazie wczesnego wydania. Są to (w kolejności alfabetycznej): Guilherme Alves, Christiano Anderson, Konstantin Baikov, K. Alex Birch, Michael Boesl, Lucas Brunialti, Sergio Cortez, Gino Crecco, Chukwuerika Dike, Juan Esteras, Federico Fissore, Will Frey, Tim Gates, Alexander Hagerman, Chen Hanxiao, Sam Hyeong, Simon Ilincev, Parag Kalra, Tim

King, David Kwast, Tina Lapine, Wanpeng Li, Guto Maia, Scott Martindale, Mark Meyer, Andy McFarland, Chad McIntire, Diego Rabatone Oliveira, Francesco Piccoli, Meredith Rawls, Michael Robinson, Federico Tula Rovaletti, Tushar Sadhwani, Arthur Constantino Scardua, Randal L. Schwartz, Avichai Sefati, Guannan Shen, William Simpson, Vivek Vashist, Jerry Zhang, Paul Zuradzki – a ponadto wielu innych, którzy nie podali swoich nazwisk, przesłali poprawki, gdy już przekazałem maszynopis do redakcji albo zostali pominięci, bo ja zapomniałem zapisać ich nazwisk – wybaczcie.

W trakcie moich badań nauczyłem się wiele na temat typów, współbieżności, dopasowywania wzorców i metaprogramowania dzięki rozmowom z takimi osobami, jak Michael Albert, Pablo Aguilar, Kaleb Barrett, David Beazley, J. S. O. Bueno, Bruce Eckel, Martin Fowler, Ivan Levkivskyi, Alex Martelli, Peter Norvig, Sebastian Rittau, Guido van Rossum, Carol Willing i Jelle Zijlstra.

Redaktorzy z wydawnictwa O'Reilly, Jeff Bleiel, Jill Leonard i Amelia Blevins zgłaszali sugestie i propozycje, które w wielu miejscach usprawniły przepływ treści książki. Jeff Bleiel i redaktor techniczny Danny Elfanbaum wspierali mnie w całym tym maratonie.

Wskazówki i sugestie każdego z nich sprawiły, że książka jest lepsza i dokładniejsza. Nieuniknione jest jednak, że nadal można znaleźc w niej błędy mojego własnego autorstwa. Z góry przepraszam za to Czytelników.

Na koniec chcę wyrazić moje gorące podziękowania kolegom z Thoughtworks Brazil – zaś szczególnie mojemu sponsorowi, Alexey Boas – którzy wspierali ten projekt na wiele sposobów, przez cały czas.

No i oczywiście każdy, kto pomógł mi zrozumieć Pythona i napisać pierwsze wydanie, zasługuje dziś na podwójne podziękowania. Nie byłoby drugiego wydania, gdyby nie pierwsze.

Podziękowania do 1 wydania

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 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

Struktury danych

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.
>
> *– Jim Hugunin, twórca Jython, współtwórca AspectJ, architekt .Net DLR*[1]

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 uczyły się innego języka obiektowego przed Pythonem, 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 wykonywać podstawowe operacje na obiektach, często wyzwalane przez specjalną składnię. Nazwy metod specjalnych są zawsze zapisywane z dwoma podkreśleniami z przodu

1 *Story of Jython* [Historia Jythona] (*https://fpy.li/1-1*), napisana jako przedmowa do książki *Jython Essentials* (O'Reilly, 2002), której autorami są Samuele Pedroni i Noel Rappin.

i z tyłu (tj. `__getitem__`). W celu przetworzenia kodu `my_collection[key]` interpreter wywołuje metodę `my_collection.__getitem__(key)`.

Implementujemy metody specjalne, gdy chcemy, aby nasze obiekty obsługiwały podstawowe konstrukcje języka oraz interakcję z nimi. Przykładami podstawowych konstrukcji języka są:

- Kolekcje
- Dostęp do atrybutów
- Iteracje (w tym asynchroniczne, używające `async for`)
- Przeciążanie operatorów
- Wywoływanie funkcji i metod
- Reprezentacja i formatowanie łańcuchów
- Asynchroniczne programowanie przy użyciu `await`
- Tworzenie i niszczenie obiektów
- Konteksty zarządzane używające instrukcji `with`) lub `async`

Magic i dunder

Termin *metoda magiczna* to slangowe określenie metody specjalnej, ale jak możemy mówić o konkretnej metodzie, takiej jak `__getitem__`? Od Steve'a Holdena nauczyłem się mówić „dunder-getitem". „Dunder" jest skrótem od „dwa podkreślenia (*underscore*) przed i po". To dlatego metody specjalne są znane jako *metody dunder*. Rozdział „Lexical Analysis" (*https://fpy.li/1-3*) podręcznika *The Python Language Reference* ostrzega, że „dowolne użycie nazw `__*__`, w dowolnym kontekście, jeśli nie jest jawnie zgodne z udokumentowanym użyciem, może ulec awarii bez ostrzeżenia".

Co nowego w tym rozdziale

Rozdział ten uległ nieznacznym zmianom od pierwszego wydania, gdyż jest wprowadzeniem do modelu danych Pythona, który jest dość stabilny. Najbardziej znaczące zmiany to:

- Metody specjalne obsługujące programowanie asynchroniczne i inne nowe funkcjonalności, które zostały dodane do tabel w podrozdziale „Przegląd metod specjalnych" na stronie 16.
- Nowy rysunek 1-2, pokazujący użycie metod specjalnych w nowym podrozdziale „API kolekcji" na stronie 14, w tym abstrakcyjną klasę bazową `collections.abc.Collection` wprowadzoną w Pythonie 3.6.

Dodatkowo zarówno tu, jak i w całym wydaniu drugim przyjąłem składnię *f-string* wprowadzoną w Pythonie 3.6, jako że jest bardziej czytelna i często wygodniejsza, niż starsze notacje formatowania łańcuchów: metoda `str.format()` oraz operator `%`.

Jednym z powodów, aby nadal używać składni `my_fmt.format()`, jest sytuacja, gdy definicja `my_fmt` musi znajdować się w innym miejscu kodu, niż miejsce wykonywania operacji formatującej. Na przykład wtedy, gdy `my_fmt` zawiera wiele wierszy kodu i lepiej byłoby ją zdefiniować w stałej albo gdy musi pochodzić z pliku konfiguracyjnego albo z bazy danych. Są to rzeczywiste potrzeby, ale nie zdarzają się zbyt często.

Pythoniczna talia kart

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

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. Używamy `namedtuple` 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 odpowiada 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 będzie 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`, 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 13 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 wywoła zaimplementowaną metodę `__len__`.

Jednak interpreter używa skrótu w przypadku typów wbudowanych, takich jak `list`, `str`, `bytearray` lub rozszerzeń, takich jak tablice NumPy. Kolekcje Pythona o zmiennym rozmiarze napisane w C zawierają strukturę[2] o nazwie `PyVarObject`, która zawiera pole `ob_size` przechowujące liczbę elementów kolekcji. Tak więc, jeśli `my_object` jest wystąpieniem jednego z tych wbudowanych typów, wywołanie `len(my_object)` odczytuje wartość pola `ob_size`, co jest znacznie szybsze od wywołania metody.

Najczęściej wywoływanie metod specjalnych odbywa się niejawnie. Na przykład instrukcja `for i in x:` w rzeczywistości powoduje wywołanie funkcji `iter(x)`, która z kolei może wywołać metodę `x.__iter__()`, jeśli jest ona dostępna, albo użyć `x.__getitem__()`, jak w przypadku naszego przykładu `FrenchDeck`.

Zwykle kod nie powinien zawierać zbyt wielu bezpośrednich wywołań metod specjalnych. O ile nie zajmujemy się metaprogramowaniem, powinniśmy znacznie 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 – w przypadku typów wbudowanych – są szybsze od wywołań metod. Zobacz na przykład „Używanie iter wobec obiektów wywoływalnych” na stronie 620.

W kolejnych podrozdziałach zobaczymy niektóre spośród najważniejszych zastosowań metod specjalnych:

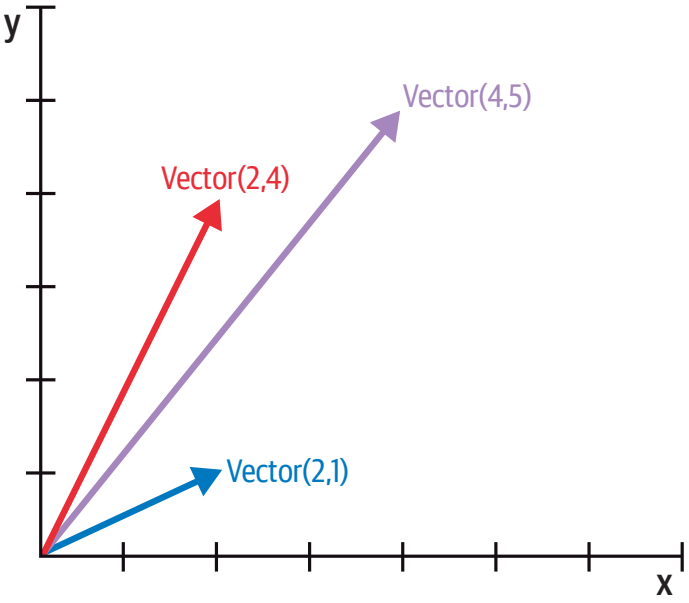
- Emulowanie typów numerycznych
- Reprezentacja tekstowa obiektów
- Wartość logiczna obiektu
- Implementowanie kolekcji

Emulacja typów liczbowych

Wiele metod specjalnych pozwala obiektom użytkownika reagować na operatory, takie jak `+`. Zajmiemy się tym bardziej szczegółowo w rozdziale 16. 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).

2 W języku C *struktura* to obiekt typu rekordu z nazwanymi polami.



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 17.

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 + v2
Vector(4, 5)
```

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

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

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

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)
15.0
```

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*

```
"""
vector2d.py: uproszczona klasa demonstrująca niektóre metody specjalne.
Została uproszona dla celów dydaktycznych. Brakuje w niej właściwej obsługi
błędów, szczególnie w metodach ``__add__`` i ``__mul__``.
Przykład ten zostanie znacznie rozbudowany w dalszej części książki.
Dodawanie::
    >>> v1 = Vector(2, 4)
    >>> v2 = Vector(2, 1)
    >>> v1 + v2
    Vector(4, 5)
Wartość bezwzględna::
    >>> v = Vector(3, 4)
    >>> abs(v)
    5.0
Mnożenie skalarne::
    >>> v * 3
    Vector(9, 12)
    >>> abs(v * 3)
    15.0
"""

import math

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
```

```
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.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)
```

Zaimplementowaliśmy cztery metody specjalne jako uzupełnienie `__init__`. Można zauważyć, że 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.

Przykład 1-2 implementuje dwa operatory: `+` oraz `*`, aby pokazać podstawowe użycie metod `__add__` i `__mul__`. W obu przypadkach te metody zwracają nową instancję klasy `Vector` i nie modyfikują żadnego operandu – `self` lub `other` są jedynie odczytywane. Jest to oczekiwane zachowanie operatorów infiksowych: mają tworzyć nowe obiekty i nie tykać swoich operandów. Znacznie więcej powiemy na ten temat w rozdziale 16.

Tak, jak go zaimplementowaliśmy, przykład 1-2 pozwala na mnożenie wektora przez liczbę, ale nie liczby przez wektor, co narusza przemienność iloczynu skalarnego. Poprawimy to za pomocą metody specjalnej `__rmul__` w rozdziale 16.

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. Gdybyśmy nie zaimplementowali 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 formatowania (*https://fpy.li/1-4*) używanej przez *f-strings* w metodzie `str.format`.

Zauważ, że w *f-string* w naszej implementacji `__repr__` używa `!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.

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)`).

Dla kontrastu metoda `__str__` jest wywoływana przez wbudowaną funkcję `str()` i niejawnie używana w funkcji `print`. Metoda `__str__` powinna zwracać łańcuch odpowiedni do wyświetlenia dla użytkowników końcowych.

Czasami sam łańcuch zwracany przez `__repr__` jest dostatecznie przyjazny dla użytkownika i nie ma potrzeby kodowania `__str__`, gdyż implementacja odziedziczona z klasy `object` wywołuje `__repr__` jako działanie zapasowe. Przykład 5-2 jest jednym z kilku przykładów w tej książce, które zawierają niestandardową metodę `__str__`.

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.

„What is the difference between __str__ and __repr__ in Python" (*https://fpy.li/1-5*) to pytanie z witryny Stack Overflow, na które wspaniałych odpowiedzi udzielili Pythoniści Alex Martelli i Martijn Pieters.

Wartość logiczna typu niestandardowego

Chociaż Python ma typ `bool`, akceptuje 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 koncepcyjnie 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. Poza metodą `__bool__` rzadko zdarza się konieczność jawnego wywołania `bool()`, gdyż każdego obiektu można użyć w kontekście logicznym.

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* (*https://fpy.li/1-6*).

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.

API kolekcji

Rysunek 1-2 dokumentuje interfejs podstawowych typów kolekcji występujących w języku. Wszystkie klasy w tym diagramie to ABC – abstrakcyjne klasy bazowe. ABC i moduł `collections.abc` omówimy bliżej w rozdziale 13. Celem tego krótkiego podrozdziału jest przedstawienie ogólnego obrazu najważniejszych interfejsów kolekcji w Pythonie i pokazanie, jak są budowane z metod specjalnych.

Każda z najwyższych klas ABC zawiera pojedynczą metodę specjalną. Klasa ABC `Collection` (nowa w wersji Python 3.6) unifikuje trzy kluczowe interfejsy, które powinna implementować każda kolekcja:

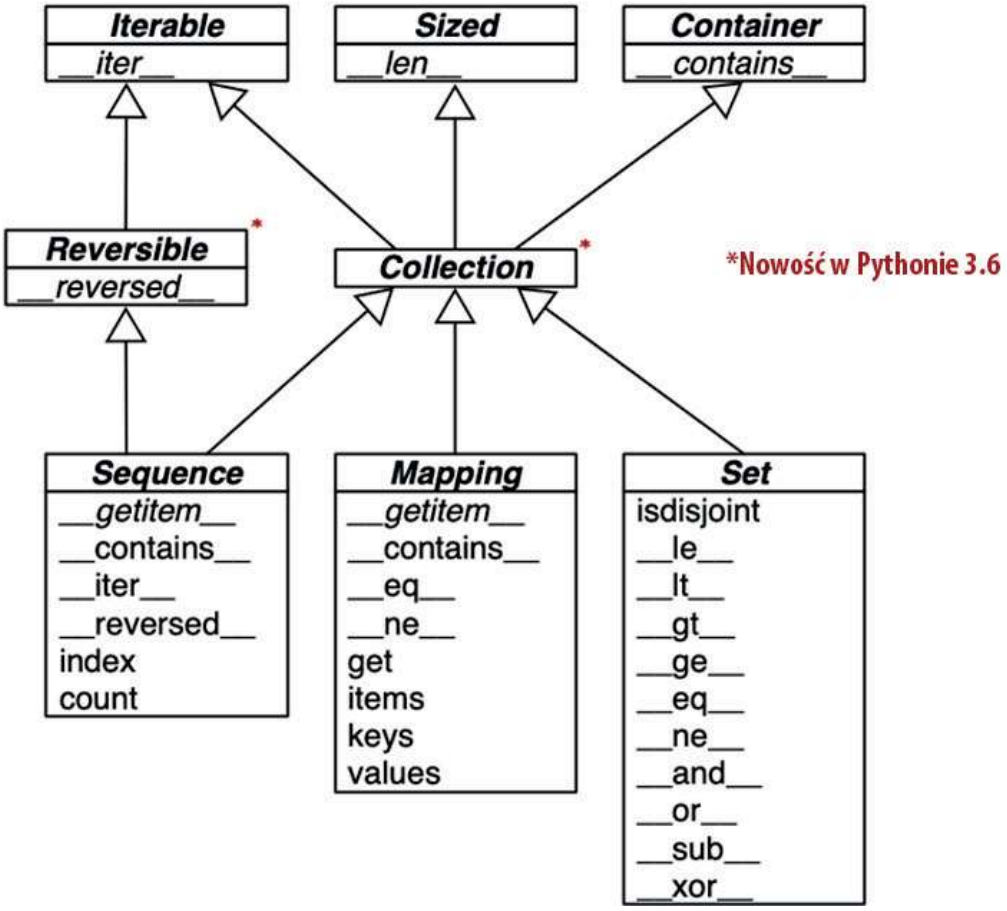
- `Iterable` w celu obsługi `for`, rozpakowywania (*https://fpy.li/1-7*) i innych form iteracji.
- `Sized` w celu wsparcia wbudowanej funkcji `len`.
- `Container` dla obsługi operatora `in`.

Python nie wymaga od konkretnych klas, aby rzeczywiście wywodziły się z którejkolwiek z tych klas ABC. Dowolna klasa implementująca `__len__` spełnia wymagania interfejsu `Sized`.

Trzy bardzo ważne specjalizacje `Collection` to:

- `Sequence`, które formalizuje interfejs wbudowanych funkcji, takich jak `list` i `str`.
- `Mapping`, implementowane przez `dict`, `collections.defaultdict` itp.
- `Set`, czyli interfejs typów wbudowanych `set` i `frozenset`.

Tylko klasa `Sequence` jest `Reversible`, gdyż sekwencje wspierają dowolne porządkowanie swojej zawartości, czego nie obsługują odwzorowania ani zbiory.



Rysunek 1-2 *Diagram UML klas z podstawowymi typami kolekcji. Metody o nazwach napisanych kursywą są abstrakcyjne, zatem muszą być implementowane przez konkretne podklasy, takie jak* `list` *i* `dict`*. Pozostałe metody mają konkretne implementacje, zatem podklasy mogą je dziedziczyć.*

Począwszy od Pythona 3.7 typ `dict` jest oficjalnie „uporządkowany", ale oznacza to tylko tyle, że zachowywana jest kolejność wstawiania kluczy. Nie można zmienić kolejności kluczy w `dict` wedle naszego życzenia.

Wszystkie metody specjalne w klasie ABC `Set` implementują operatory infiksowe. Na przykład `a & b` oblicza przecięcie (iloczyn) zbiorów `a` i `b` i jest implementowane w metodzie specjalnej `__and__`.

W kolejnych dwóch rozdziałach szczegółowo omówimy sekwencje, odwzorowania i zbiory z biblioteki standardowej.

Zajmijmy się teraz głównymi kategoriami metod specjalnych zdefiniowanymi w modelu danych Pythona.

Przegląd metod specjalnych

Rozdział „Data Model" [Model danych] dokumentacji *The Python Language Reference* (*https://fpy.li/dtmodel*) zawiera listę ponad 80 nazw metod specjalnych, z których ponad połowa służy do implementacji operatorów arytmetycznych, bitowych i porównania. Przegląd dostępnych metod zawierają poniższe tabele.

Tabela 1-1 pokazuje nazwy metod specjalnych, z wyjątkiem tych, które są używane do implementacji operatorów infiksowych albo podstawowych funkcji matematycznych, takich jak `abs`. Większość tych metod zostanie omówiona w dalszej części książki, włącznie z najnowszymi uzupełnieniami: asynchronicznymi metodami specjalnymi, takimi jak `__anext__` (dodana w Pythonie 3.5) oraz hook dostosowywania klas `__init_subclass__` (z Pythona 3.6).

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

Kategoria	Nazwy metod
Reprezentacja tekstowa/bajtowa	`__repr__ __str__ __format__ __bytes__ __fspath__`
Konwersja na liczbę	`__bool__ __complex__ __int__ __float__ __hash__ __index__`
Emulacja kolekcji	`__len__ __getitem__ __setitem__ __delitem__ __contains__`
Iteracja	`__iter__ __aiter__ __next__ __anext__ __reversed__`
Emulacja wywoływalności	`__call__ __await__`
Zarządzanie kontekstem	`__enter__ __exit__ __aexit__ __aenter__`
Tworzenie i niszczenie wystąpienia	`__new__ __init__ __del__`
Zarządzanie atrybutami	`__getattr__ __getattribute__ __setattr__ __delattr__ __dir__`
Deskryptory atrybutów	`__get__ __set__ __delete__ __set_name__`
Abstrakcyjne klasy bazowe	`__instancecheck__ __subclasscheck__`
Metaprogramowanie klas	`__prepare__ __init_subclass__ __class_getitem__ __mro_entries__`

Infiksowe i numeryczne operatory są obsługiwane przez metody specjalne wymienione w tabeli 1-2. W tym miejscu najnowsze uzupełnienia to `__matmul__`, `__rmatmul__` oraz `__imatmul__`, dodane w Pythonie 3.5 w celu obsługii użycia `@` jako operatora infiksowego mnożenia macierzy, co zobaczymy w rozdziale 16.

Tabela 1-2 *Nazwy metod specjalnych dla operatorów*

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

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 16 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" na stronie 8 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 traktowana specjalnie 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. Dla przykładu, biblioteki bazodanowe często zwracają wyniki zapytań opakowane w kolekcje podobne do sekwencji. Zapoznanie się z większością typów sekwencyjnych jest tematem rozdziału 2, a implementacja własnej sekwencji zostanie opisana w rozdziale 12, 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 16 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, 3rd Edition* (O'Reilly), której autorem jest Alex Martelli, wspaniale opisuje model danych. 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 6 000 wpisów odpowiedzi. Zobacz jego profil użytkownika w witrynie Stack Overflow (*https://fpy.li/1-9*).

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 3E*, 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" (*https://fpy.li/1-10*). 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.

Mugolske metody

The Original Hacker's Dictionary (*https://fpy.li/1-11*) definiuje *magic* (magiczne) jako „jeszcze nie wyjaśnione albo zbyt skomplikowane, aby wyjaśnić" albo „funkcjonalność, która nie jest w ogólności znana publicznie, która pozwala na coś niemożliwego w innym przypadku".

Społeczność Ruby nazywa swoje odpowiedniki metod specjalnych *metodami magicznymi*. Duża część społeczności Pythona również przyjęła ten termin. Osobiście uważam, że metody specjalne są w istocie *przeciwieństwem* magii. Python i Ruby uzbrajają swoich użytkowników w bogaty protokół metaobiektów, który jest w pełni udokumentowany, co pozwala takim mugolom, jak ja i ty, emulować wiele funkcjonalności dostępnych dla deweloperów jądra, piszących interpretery tych języków.

Dla kontrastu rozważmy Go. Niektóre obiekty w tym języku mają cechy, które są magiczne w tym sensie, że nie możemy ich emulować w naszych własnych typach definiowanych przez użytkownika. Dla przykładu tablice, łańcuchy i odwzorowania w Go wspierają użycie nawiasów kwadratowych dla dostępu do elementów, na przykład `a[i]`. Nie ma jednak sposobu, aby notacja `[]` działała w nowym, zdefiniowanym przez nas typie kolekcji. Co jeszcze gorsze, Go na poziomie użytkownika nie udostępnia koncepcji iterowalnego interfejsu ani obiektu iteratora, zatem jego składnia `for/range` jest ograniczona tylko do obsługi pięciu „magicznych" typów wbudowanych, obejmujących tablice, łańcuchy i odwzorowania.

Być może kiedyś projektanci Go ulepszą protokół metaobiektowy. Jednak obecnie jest on znacznie bardziej ograniczony od tego, co mamy w Pythonie lub Ruby.

Metaobiekty

The Art of the Metaobject Protocol (AMOP) to moja ulubiona książka informatyczna. Wspominam o niej, gdyż termin *protokół metaobiektów* przydaje się przy rozważaniu modelu danych Pythona i podobnych funkcjonalności 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* (*https://fpy.li/1-12*), część frameworku, na którym zbudowano system zarządzania treściami Plone (*https://fpy.li/1-13*).

Rozdział 2

Tablica sekwencji

> 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.
>
> *– Geurts, Meerten i Pemberton*[1]

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 zostały dodane w Pythonie 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 12.

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

Oto główne zagadnienia, które zostaną przedstawione w tym rozdziale:

- Zrozumienie list oraz podstawy wyrażeń generatorów.
- Używanie krotek jako rekordów kontra ich wykorzystywanie jako niezmiennych list.
- Rozpakowywanie sekwencji i wzorce sekwencji.
- Czytanie i zapisywanie wycinków.
- Specjalizowane typy sekwencji, jak tablice i kolejki.

Co nowego w tym rozdziale

Najbardziej znaczącym uzupełnieniem tego rozdziału jest podrozdział „Dopasowywanie wzorców w sekwencjach" na stronie 40. To tu w drugim wydaniu po raz pierwszy pojawia się nowa funkcjonalność: dopasowywanie wzorców wprowadzone w Pythonie 3.10.

Pozostałe zmiany nie są uzupełnieniami, ale ulepszeniami pierwszego wydania:

- Nowy diagram i opis wewnętrznych mechanizmów sekwencji, przeciwstawiający sobie kontenery i płaskie sekwencje.
- Krótkie porównanie cech wydajnościowych i pamięciowych list oraz krotek.
- Pułapki występujące przy krotkach o zmiennych elementach i jak je wykrywać, jeśli zajdzie potrzeba.

Omówienie nazwanych krotek przeniosłem do podrozdzialu „Klasyczne krotki nazwane" na stronie 176 w rozdziale 5, gdzie zostaną porównane z `typing.NamedTuple` i `@dataclass`.

Aby zrobić miejsce na nową treść i utrzymać liczbę stron książki w rozsądnych granicach, podrozdział „Managing Ordered Sequences with Bisect" (Zarządzanie uporządkowanymi sekwencjami za pomocą bisect) z pierwszego wydania jest teraz postem w witrynie towarzyszącej *fluentpython.com* (*https://fpy.li/bisect*).

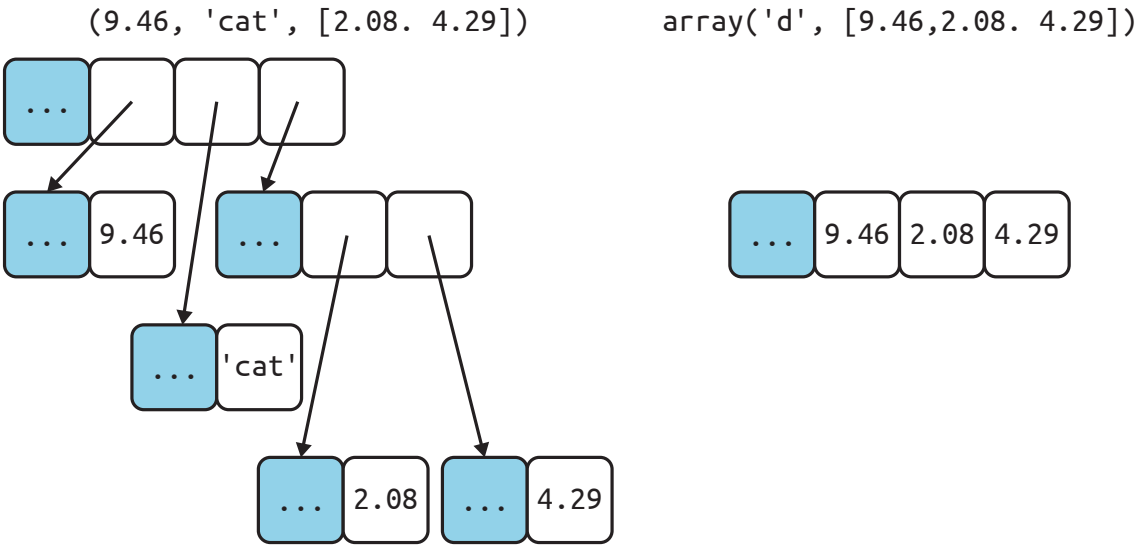
Przegląd wbudowanych sekwencji

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

Sekwencje kontenerowe
: Mogą przechowywać elementy różnych typów, włącznie z innymi kontenerami. Przykłady to `list`, `tuple` i `collections.deque`.

Sekwencje płaskie
: Przechowują elementy jednego typu. Przykłady to `str`, `bytes` i `array.array`.

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.



Rysunek 2-1 *Uproszczone diagramy pamięci dla krotki i tablicy, każdej z trzema elementami. Szare komórki reprezentują nagłówek w pamięci każdego obiektu Pythona – bez zachowania skali. Krotka (`tuple`) zawiera tablicę referencji do swoich elementów. Każdy z nich jest oddzielnym obiektem Pythona, być może zawierającym referencje do innych obiektów, jak dwuelementowa lista w tym przykładzie. Dla kontrastu, tablica (`array`) jest pojedynczym obiektem, zawierającym tablicę języka C złożoną z trzech liczb zmiennoprzecinkowych.*

Zatem sekwencje płaskie są bardziej upakowane, ale przy tym ograniczone do przechowywania prymitywnych wartości, takich jak znaki, bajty i liczby.

Każdy obiekt Pythona w pamięci zawiera nagłówek z metadanymi. Najprostszy obiekt Pythona, czyli `float`, ma pole wartości oraz dwa pola metadanych:

- `ob_refcnt`: licznik referencji do obiektu.
- `ob_type`: wskaźnik do typu obiektu.
- `ob_fval`: liczba `double` języka C, przechowująca wartość.

W 64-bitowej kompilacji Pythona każde z tych pól zajmuje 8 bajtów. To dlatego tablica wartości `float` jest znacznie bardziej zwarta, niż krotka z tymi samymi wartościami: tablica to pojedynczy obiekt przechowujący surowe wartości, podczas gdy krotka składa się z wielu obiektów – siebie samej oraz każdego obiektu `float` w niej zawartego.

Innym sposobem grupowania sekwencji jest ich zmienność:

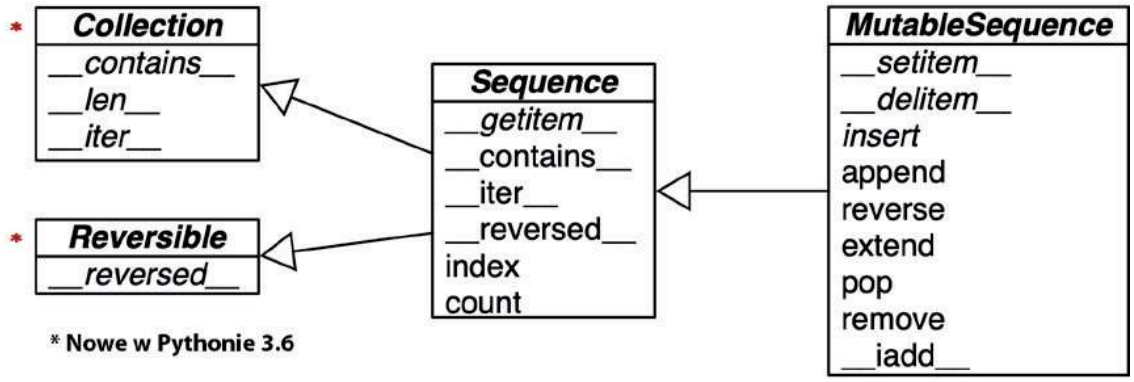
Sekwencje zmienne
: Na przykład `list`, `bytearray`, `array.array` i `collections.deque`.

Sekwencje niezmienne
: Na przykład `tuple`, `str` i `bytes`.

Rysunek 2-2 pomaga zwizualizować, że sekwencje zmienne dziedziczą wszystkie metody z sekwencji niezmiennych i implementują wiele dodatkowcyh metod. Konkretne wbudowane typy sekwencji nie są faktycznie podklasami abstrakcyjnych klas bazowych (ABC) `Sequence` i `MutableSequence`, ale są *podklasami wirtualnymi* zarejestrowanymi względem tych klas ABC, co zobaczymy w rozdziale 13. Dzięki temu, że są podrzędnymi klasami wirtualnymi, `tuple` i `list` przechodzą poniższe testy:

```
>>> from collections import abc
>>> issubclass(tuple, abc.Sequence)
True
>>> issubclass(list, abc.MutableSequence)
True
```

Niemniej jednak klasy ABC są przydatne do formalizowania oczekiwanych funkcjonalności w pełni funkcjonalnych typów sekwencyjnych.



Rysunek 2-2 *Diagram 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)*

Trzeba mieć na uwadze te wspólne cechy: zmienne kontra niezmienne, konterowe kontra płaskie. Są one pomocne przy ekstrapolowaniu na inne typy tego, co wiemy o jednym typie sekwencji.

Najbardziej podstawowym typem sekwencji jest `list` – zmienny kontener. 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. Są one 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 z 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 z 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 oznaczenia kontynuacji wiersza, który w dodatku nie zadziała, jeśli przypadkowo wpiszemy po nim spację. Ponadto gdy te pary delimiterów są używane do definiowania literału z rozdzielanej przecinkami listy elementów, końcowy przecinek jest ignorowany. Tak więc przy kodowaniu wielowierszowego literału listowego rozsądne jest umieszczenie przecinka po ostatniej pozycji, co ułatwi kolejnemu koderowi dodanie następnej pozycji do tej listy, a także zmniejszy zamęt przy wyszukiwaniu różnic.

Lokalny zasięg w wyrażeniach listowych i generatora

W Pythonie 3 wyrażenia listowe, wyrażenia generatora i ich kuzyni, czyli wyrażenia zbiorów (*setcomp*) oraz słownikowe (*dictcomp*), mają zasięg lokalny do przechowywania zmiennych przypisanych w klauzuli `for`.

Jednak zmienne przypisane za pomocą „operatora morsa” `:=` pozostają dostępne po powroce z tych wyrażeń – w przeciwieństwie do lokalnych zmiennych w funkcji. Dokument „PEP 572 – Assignment Expressions” [Wyrażenia przypisania] (*https://fpy.li/pep572*) definiuje zasięg obiektu docelowego `:=` jako ograniczającą funkcję, chyba że istnieje globalna albo nielokalna deklaracja tego obiektu[2].

```
>>> x = 'ABC'
>>> codes = [ord(x) for x in x]
```

2 Dziękuję czytelniczce Tinie Lapine za zwrócenie uwagi na ten fakt.

```
>>> x  ❶
'ABC'
>>> codes  ❷
[65, 66, 67]
>>> codes = [last := ord(c) for c in x]
>>> last
67
>>> c  ❸
Traceback (most recent call last):

File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

❶ `x` nie został oderwany: nadal jest powiązany z `'ABC'`.

❷ `last` pozostało.

❸ `c` jest nieobecne; istniało tylko wewnątrz wyrażenia listowego.

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*

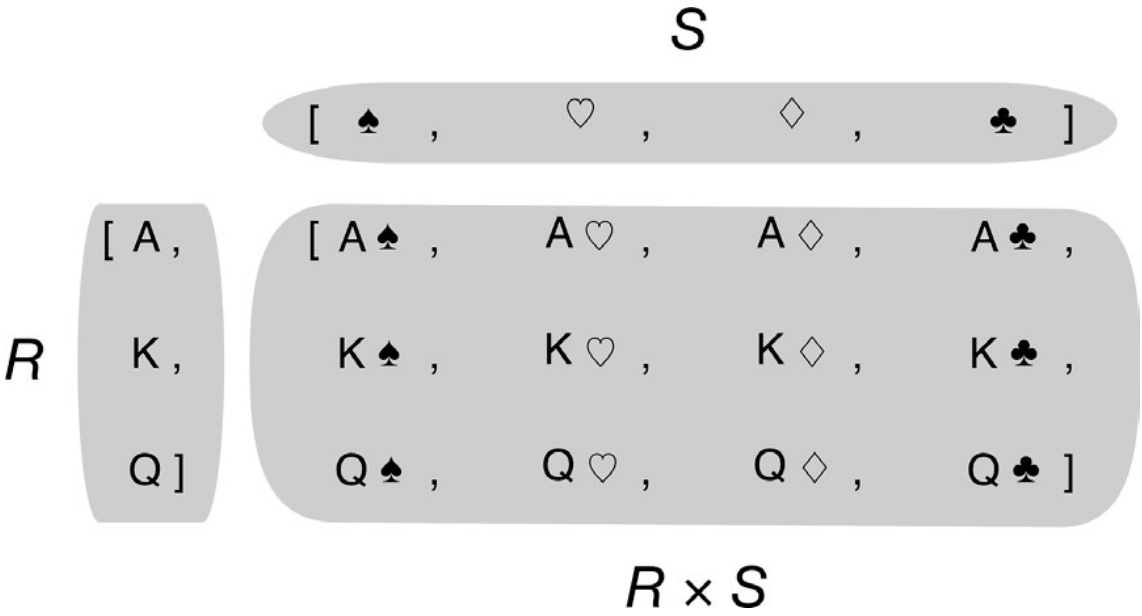
```
>>> symbols = '$¢£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord, symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

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 (*https://fpy.li/code*) jest prostym testem szybkości, porównującym wyrażenie listowe z `filter/map`.

Więcej na temat funkcji `map` i `filter` będę miał do powiedzenia w rozdziale 7. Teraz zajmiemy się użyciem 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 iloczynu kartezjańskiego dwóch obiektów iterowalnych lub większej ich liczby. Elementy składające się na iloczyn kartezjański to krotki powstałe z elementów pochodzących z każdego wejściowego obiektu iterowalnego. Wynikowa lista ma długość równą przemnożonym długościom wejściowych obiektów iterowalnych. Zobacz rysunek 2-3.



Rysunek 2-3 *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 załadować ją do innego konstruktora.

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” na stronie 61.

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 17 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 niepełna prawda. Krotki mają podwójną rolę: mogą służyć jako niezmienne listy, ale 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 (km^2).

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.

Często myślimy o rekordach jako strukturach danych z nazwanymi polami. W rozdziale 5 przedstawię dwa sposoby tworzenia krotek z nazwanymi polami.

Często jednak nie ma potrzeby przechodzenia przez trudy tworzenia klasy jedynie po to, aby nazwać pola, szczególnie gdy wykorzystamy rozpakowywanie i unikniemy używania indeksów w celu dostępu do pól. 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*.

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].

Podrozdział „Rozpakowywanie sekwencji i typów iterowalnych" na stronie 36 zawiera znacznie więcej informacji o rozpakowywaniu nie tylko krotek, ale w ogólności sekwencji i obiektów iterowalnych.

Teraz możemy rozważyć ich klasę `tuple` jako niezmienną odmianę klasy `list`.