Spis treści

Część V: **Przepływ sterowania**

Część VI: **Metaprogramowanie**

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.

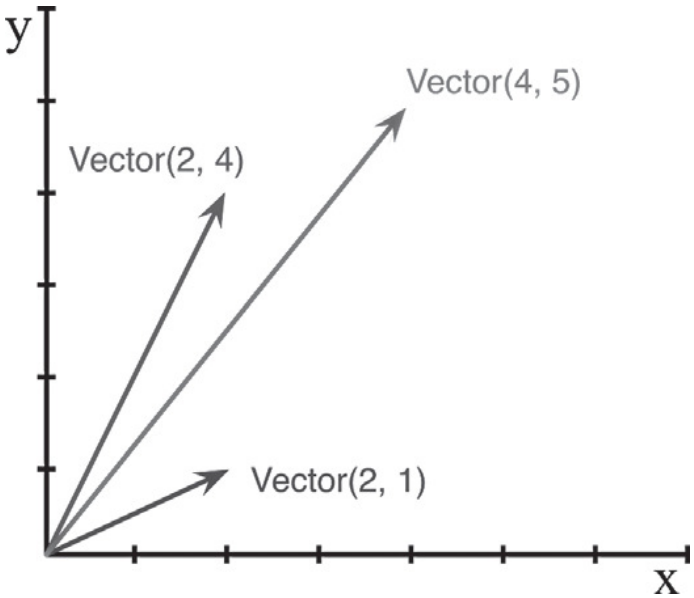
– w przypadku typów wbudowanych – są szybsze od wywołań metod. Zobacz na przykład „Bliższe przyjrzenie się funkcji iter” w rozdziale 14.

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.