Wydanie II

Czysty kod w Pythonie

Twórz wydajny i łatwy w utrzymaniu kod

Mariano Anaya

Helion

Packt

Tytuł oryginału: Clean Code in Python: Develop maintainable and efficient code, 2nd Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-8611-2

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/czyko2.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/czyko2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 2

Kod pythoniczny

W tym rozdziale zbadamy sposób wyrażania pomysłów w Pythonie, zwracając uwagę na pewne osobliwości. Jeśli znasz standardowe sposoby wykonywania niektórych zadań w programowaniu (na przykład uzyskanie ostatniego elementu listy, iterowanie i wyszukiwanie) lub jeśli wcześniej programowałeś w innych językach (takich jak C, C++ i Java), to przekonasz się, że — ogólnie rzecz biorąc — Python zapewnia własne sposoby wykonywania większości typowych zadań.

W programowaniu idiom to konkretny sposób pisania kodu w celu wykonania określonego zadania. Jest to coś powszechnego, co powtarza się i za każdym razem ma taką samą strukturę. Niektórzy mogą nawet nazywać go wzorcem, ale uważaj, ponieważ nie są to wzorce projektowe (te zbadamy później). Główną różnicą pomiędzy idiomami a wzorcami jest to, że wzorce projektowe są pomysłami wysokiego poziomu, niezależnymi od języka (w pewnym sensie), ale nie przekładają się natychmiast na kod. Z drugiej strony, idiomy są faktycznie kodowane. Idiomy to sposoby zapisu konstrukcji programowych w celu wykonania określonego zadania.

Ponieważ idiomy są kodem, są zależne od języka. Każdy język ma swoje idiomy, czyli sposoby, w jakie są wykonywane działania w tym konkretnym języku (na przykład otwieranie i zapisywanie pliku w językach C lub C++). Gdy kod korzysta z tych idiomów, określa się go jako idiomatyczny. W przypadku Pythona często mówi się, że kod jest pythoniczny.

Istnieje wiele powodów, dla których warto stosować zalecenia i pisać pythoniczny kod (co zobaczymy i co będziemy analizować). To dlatego, że pisanie idiomatycznego kodu zwykle lepiej się sprawdza. Taki kod jest również bardziej kompaktowy i łatwiejszy do zrozumienia. Są to cechy, które zawsze są w kodzie pożądane, ponieważ dzięki nim staje się wydajniejszy.

Po drugie, tak, jak napisałem w poprzednim rozdziale, jest ważne, aby cały zespół programistów przyzwyczaił się do tych samych wzorców i struktury kodu, ponieważ to pomoże im skupić się na prawdziwej istocie problemu i pozwoli uniknąć popełnienia błędów.

Oto cele niniejszego rozdziału.

- Zapoznanie się z indeksami i wycinkami oraz poprawnym sposobem implementacji obiektów, które mogą być indeksowane.

- Nauczenie się sposobu implementacji sekwencji i innych obiektów iterowalnych.
- Zapoznanie się z dobrymi przypadkami użycia menedżerów kontekstu i nauczenie się pisania skutecznych konstrukcji tego typu.
- Nauczenie się implementowania bardziej idiomatycznego kodu dzięki zastosowaniu metod magicznych.
- Unikanie popularnych w Pythonie pomyłek prowadzących do niepożądanych efektów ubocznych.

Zacznę od omówienia pierwszego elementu na liście (czyli indeksów i wycinków).

Indeksy i wycinki

W Pythonie, podobnie jak w innych językach, niektóre struktury danych lub typy obsługują dostęp do ich elementów za pośrednictwem indeksów. Inną cechą wspólną Pythona z większością języków programowania jest to, że pierwszy element sekwencji znajduje się pod indeksem 0. Jednak w przeciwieństwie do innych języków, gdy chcesz uzyskać dostęp do elementów w innej kolejności niż zwykle, Python oferuje dodatkowe własności.

Oto przykład: w jaki sposób można uzyskać dostęp do ostatniego elementu tablicy w języku C? Właśnie to działanie postanowiłem wypróbować, gdy zacząłem korzystać z Pythona po raz pierwszy. Myśląc w ten sam sposób, jak w C, chciałbym uzyskać element z pozycji odpowiadającej długości tablicy minus jeden. W Pythonie ten sposób również działa, ale można również użyć ujemnej wartości indeksu, co powoduje liczenie od ostatniego elementu. Pokazałem to w poniższym przykładzie:

```
>>> my_numbers = (4, 5, 3, 9)
>>> my_numbers[-1]
9
>>> my_numbers[-3]
5
```

Jest to przykład preferowanego (pythonicznego) sposobu wykonywania działań.

Oprócz możliwości uzyskania tylko jednego elementu, możemy uzyskać wiele za pomocą wycinków. Oto przykład:

```
>>> my_numbers = (1, 1, 2, 3, 5, 8, 13, 21)
>>> my_numbers[2:5]
(2, 3, 5)
```

W tym przypadku zastosowanie składni korzystającej z nawiasów kwadratowych pozwala na uzyskanie wszystkich elementów w postaci krotki: począwszy od indeksu wskazanego jako pierwszy (włącznie z nim), aż do indeksu wymienionego jako drugi (bez niego). Wycinki w Pythonie działają w ten sposób za sprawą wykluczenia ostatniego elementu wskazanego przedziału.

Możesz wykluczyć jeden z przedziałów, początek lub koniec. W takim przypadku wycinek będzie tworzony odpowiednio od początku lub końca sekwencji, tak jak pokazałem poniżej:

```
>>> my_numbers[:3]
(1, 1, 2)
>>> my_numbers[3:]
(3, 5, 8, 13, 21)
>>> my_numbers[::] # także my_numbers[:], zwraca kopię sekwencji
(1, 1, 2, 3, 5, 8, 13, 21)
>>> my_numbers[1:7:2]
(1, 3, 8)
```

W pierwszym przykładzie wycinek będzie zawierał wszystko aż do indeksu numer 3. W drugim przykładzie w wycinku znajdą się wszystkie liczby, począwszy od pozycji 3. (włącznie) aż do końca. W przykładach od drugiego do ostatniego, gdzie oba końce są wykluczone, wycinek zawiera kopię wejściowej krotki.

W ostatnim przykładzie został uwzględniony trzeci parametr, który oznacza krok. Wskazuje liczbę elementów, o które należy przeskoczyć podczas iterowania wycinka. W tym przypadku oznaczałoby to uzyskanie elementów pomiędzy pozycjami od 1. do 7. z przeskokiem co dwa.

We wszystkich tych przypadkach, kiedy przekazujemy do sekwencji przedziały, w gruncie rzeczy przekazujemy wycinek (obiekt `slice`). Warto zauważyć, że `slice` w Pythonie to wbudowany obiekt, który można samodzielnie zbudować i bezpośrednio przekazać:

```
>>> interval = slice(1, 7, 2)
>>> my_numbers[interval]
(1, 3, 8)

>>> interval = slice(None, 3)
>>> my_numbers[interval] == my_numbers[:3]
True
```

Zwróć uwagę, że gdy brakuje jednego z elementów (początek, koniec lub krok), ten element jest interpretowany jako `None`.

> Zawsze należy dążyć do korzystania z wbudowanej składni wycinków, w przeciwieństwie do iterowania wewnątrz pętli `for` po krotce, ciągu znaków lub liście i wyłączania elementów ręcznie.

Tworzenie własnych sekwencji

Funkcjonalność, którą właśnie omówiłem, działa dzięki magicznej metodzie (magiczne metody to te, których nazwy są otoczone podwójnymi znakami podkreślenia; Python używa ich do wykonywania specjalnych działań) o nazwie `__getitem__`. Jest to metoda wywoływana w przypadku odwołania się do takiej konstrukcji jak `myobject[klucz]`, gdzie przekazujemy `klucz` (wartość wewnątrz nawiasów kwadratowych) jako parametr. Dokładniej mówiąc, sekwencja jest obiektem, który implementuje zarówno metody `__getitem__`, jak i `__len__` i z tego powodu

może być iterowany. Przykładami obiektów sekwencji w bibliotece standardowej są listy, krotki i ciągi znaków.

W tym punkcie bardziej będzie nas interesowało uzyskiwanie z obiektu pojedynczych elementów za pośrednictwem klucza niż budowanie sekwencji lub obiektów iterowalnych. Ten temat zostanie omówiony w rozdziale 7., „Generatory, iteratory i programowanie asynchroniczne".

Jeśli zamierzasz zaimplementować metodę `__getitem__` w niestandardowej klasie w swojej domenie, musisz wziąć pod uwagę niektóre kwestie po to, by postępować zgodnie z pythonicznym podejściem.

W przypadku, gdy klasa jest opakowaniem obiektu biblioteki standardowej, trzeba w miarę możliwości delegować zachowanie obiektu źródłowego. Oznacza to, że jeśli klasa jest rzeczywiście opakowaniem listy, należy wywoływać na tej liście wszystkie te same metody, aby upewnić się, że obiekt jest nadal zgodny z listą. Na poniższym listingu możemy zaobserwować przykład opakowania listy przez obiekt, a w przypadku metod, które nas interesują, po prostu delegujemy wykonanie do odpowiedniej wersji obiektu `list`:

```
from collections.abc import Sequence

class Items(Sequence):
    def __init__(self, *values):
        self._values = list(values)

    def __len__(self):
        return len(self._values)
    def __getitem__(self, item):
        return self._values.__getitem__(item)
```

Aby zadeklarować, że nasza klasa jest sekwencją, powinna ona implementować interfejs `Sequence` z modułu `collections.abc` (*https://docs.python.org/3/library/collections.abc.html*). W przypadku klas, które samodzielnie piszesz, a które mają zachowywać się jak standardowe typy obiektów (kontenery, mapowania i tak dalej), warto zaimplementować interfejsy z tego modułu. W ten sposób ujawniamy nasze intencje dotyczące tego, czym mają być obiekty tej klasy. Jest to ważne także dlatego, że użycie interfejsów wymusza zaimplementowanie wymaganych metod.

W tym przykładzie wykorzystałem kompozycję (ponieważ użyłem wewnątrz pomocniczego obiektu, który jest listą, zamiast zastosowania dziedziczenia po klasie `list`). Innym sposobem osiągnięcia podobnego celu jest wykorzystanie dziedziczenia klas. W tym przypadku należałoby rozszerzyć klasę bazową `collections.UserList`, biorąc pod uwagę względy i zastrzeżenia wymienione pod koniec tego rozdziału.

Jeśli jednak implementujesz własną sekwencję, która nie jest opakowaniem lub nie bazuje na żadnym wbudowanym obiekcie, powinieneś pamiętać o następujących sprawach.

- Podczas indeksowania według zakresu wynik powinien być egzemplarzem tego samego typu klasy.
- W zakresie dostarczonym przez wycinek należy przestrzegać semantyki stosowanej w Pythonie, polegającej na wyłączeniu elementu z końca wycinka.

Nieprzestrzeganie pierwszego punktu to subtelny błąd. Pomyśl o tym — gdy otrzymasz wycinek listy, wynik ma być listą; gdy zażądasz zakresu krotki, wynikiem powinna być krotka; a gdy poprosisz o podciąg, wynikiem powinien być ciąg znaków. W każdym przypadku jest sensowne, aby wynik był tego samego typu co obiekt wejściowy. Jeśli na przykład tworzysz obiekt, który reprezentuje przedział dat, i poprosisz o zakres tego przedziału, błędem byłoby zwrócenie listy, krotki lub czegoś innego. Zamiast tego powinieneś zwrócić nowy egzemplarz tej samej klasy z nowym zestawem przedziałów. Najlepszym tego przykładem jest biblioteka standardowa z funkcją `range`. Jeśli wywołasz `range` z przedziałem, utworzy się obiekt iterowalny, który „wie", jak tworzyć wartości w wybranym zakresie. Po określeniu przedziału dla zakresu otrzymasz nowy zakres (co ma sens), a nie listę:

```
>>> range(1, 100)[25:50]
range(26, 51)
```

Druga reguła dotyczy również spójności — dla użytkowników kod będzie bardziej znajomy i łatwiejszy w użyciu, jeśli będzie zgodny z samym Pythonem. Programiści Pythona są przyzwyczajeni do sposobu, w jaki działają wycinki, jak działa funkcja `range` i tak dalej. Wprowadzenie wyjątku w niestandardowej klasie spowoduje zamieszanie, co oznacza, że posługiwanie się nią będzie trudniejsze do zapamiętania i może prowadzić do błędów.

Teraz, gdy zapoznałeś się z indeksami i wycinkami oraz dowiedziałeś się, jak tworzyć własne, w następnym podrozdziale przyjmiemy takie samo podejście dla menedżerów kontekstu. Najpierw przyjrzymy się, jak działają menedżery kontekstu z biblioteki standardowej, a następnie przejdziemy do następnego poziomu i utworzymy własne.

Menedżery kontekstu

Menedżery kontekstu to bardzo przydatna własność Pythona. Są one tak przydatne dlatego, że poprawnie reagują na wzorzec. Istnieją powtarzające się sytuacje, w których chcemy uruchomić jakiś kod, gdzie występują określone warunki wstępne i końcowe. To oznacza, że chcemy uruchomić pewne działania odpowiednio przed i po pewnym działaniu głównym. Menedżery kontekstu są doskonałymi narzędziami do zastosowania w takich sytuacjach.

W większości przypadków menedżery kontekstu są wykorzystywane do zarządzania zasobami. Gdy na przykład otwieramy pliki, chcemy zadbać o to, aby po zakończeniu wykonywania działań były one zamknięte (aby nie doszło do wycieku deskryptorów plików). Kiedy z kolei otworzymy połączenie z usługą (lub nawet gniazdem), chcemy również mieć pewność, że zostanie ono odpowiednio zamknięta. Podobnie jest w przypadku posługiwania się plikami tymczasowymi i tak dalej.

We wszystkich tych przypadkach zwykle trzeba pamiętać, aby zwolnić wszystkie zasoby, które zostały przydzielone i to jest tylko myślenie o najlepszym przypadku. A co z wyjątkami i obsługą błędów? Biorąc pod uwagę fakt, że obsługa wszystkich możliwych kombinacji i ścieżek wykonywania programu utrudnia jego debugowanie, najczęstszym sposobem rozwiązania tego problemu jest umieszczenie kodu porządkującego w bloku `finally`. W ten sposób mamy pewność, że ten kod nie zostanie pominięty. Oto bardzo prosty przykład:

```
fd = open(filename)
try:
    process_file(fd)
finally:
    fd.close()
```

Niemniej jednak istnieje o wiele bardziej elegancki i pythoniczny sposób osiągnięcia tego samego celu:

```
with open(filename) as fd:
    process_file(fd)
```

Instrukcja `with` (PEP-343) wprowadza menedżera kontekstu. W takim przypadku funkcja `open` implementuje protokół menedżera kontekstu, co oznacza, że plik zostanie automatycznie zamknięty po zakończeniu bloku, nawet jeśli wystąpi wyjątek.

Menedżery kontekstu składają się z dwóch magicznych metod, czyli `__enter__` i `__exit__`. W pierwszym wierszu menedżera kontekstu instrukcja `with` wywoła metodę `__enter__`, a to, co ta metoda zwróci, będzie przypisane do zmiennej wymienionej po `as`. Jest to opcjonalne — w rzeczywistości metoda `__enter__` nie musi niczego konkretnego zwracać, a jeśli nawet coś zwróci, to nadal nie ma szczególnego powodu, aby tę zwracaną wartość przypisywać do zmiennej.

Po wykonaniu tego wiersza sterowanie wchodzi w nowy kontekst, w którym można uruchomić dowolny inny kod Pythona. Po zakończeniu działania ostatniej instrukcji w tym bloku następuje wyjście z kontekstu, co oznacza, że Python wywoła metodę `__exit__` obiektu menedżera kontekstu, który wcześniej wywołaliśmy.

Jeśli w bloku menedżera kontekstu wystąpi wyjątek lub błąd, metoda `__exit__` i tak zostanie wywołana, co ułatwia bezpieczne zarządzanie czyszczeniem warunków. W rzeczywistości ta metoda otrzymuje wyjątek, który został zainicjowany na bloku na wypadek, gdybyśmy chcieli obsłużyć go w niestandardowy sposób.

Mimo że menedżery kontekstu są bardzo często wykorzystywane do obsługi zasobów (jak we wspomnianym przykładzie dotyczącym plików, połączeń i tak dalej), nie jest to jedyne ich zastosowanie. Można zaimplementować własne menedżery kontekstu w celu obsłużenia konkretnej logiki, której potrzebujemy.

Menedżery kontekstu są dobrym sposobem na rozdzielenie odpowiedzialności i odizolowanie fragmentów kodu, które powinny być niezależne, ponieważ jeśli zostaną zmieszane, logika stanie się trudniejsza do utrzymania.

Dla przykładu rozważmy sytuację, w której chcemy uruchomić kopię zapasową bazy danych za pomocą skryptu. Haczyk polega na tym, że kopia zapasowa jest offline, co oznacza, że możemy ją wykonać tylko wtedy, gdy baza danych nie jest uruchomiona, a do tego trzeba ją zatrzymać. Po uruchomieniu kopii zapasowej chcemy mieć pewność, że rozpoczniemy proces od nowa, niezależnie od tego, jak przebiegał sam proces wykonywania kopii zapasowej.

Pierwsze podejście mogłoby polegać na utworzeniu ogromnej monolitycznej funkcji, która próbuje zrobić wszystko w tym samym miejscu: zatrzymuje usługę, wykonuje zadanie tworzenia kopii zapasowej, obsługuje wyjątki i wszystkie możliwe przypadki brzegowe, a następnie ponownie próbuje uruchomić usługę. Łatwo sobie wyobrazić taką funkcję, dlatego oszczędzę Ci szczegółów. Zamiast tego bezpośrednio przejdę do omówienia możliwego sposobu rozwiązania tego problemu z użyciem menedżerów kontekstu:

```
def stop_database():
    run("systemctl stop postgresql.service")

def start_database():
    run("systemctl start postgresql.service")

    class DBHandler:
        def __enter__(self):
            stop_database()
            return self

    def __exit__(self, exc_type, ex_value, ex_traceback):
        start_database()

    def db_backup():
        run("pg_dump database")

def main():
    with DBHandler():
        db_backup()
```

W tym przykładzie nie potrzebujemy wyniku menedżera kontekstu wewnątrz bloku i dlatego możemy uznać, że wartość zwracana przez metodę `__enter__` jest nieistotna. Trzeba to wziąć pod uwagę przy projektowaniu menedżerów kontekstu — warto zadać pytanie, czego potrzebujemy, gdy blok zostanie uruchomiony? Dobrą standardową praktyką (choć nieobowiązkową) powinno być zwracanie jakiejś wartości przez metodę `__enter__`.

W tym bloku, jak widać, uruchamiamy wyłącznie zadanie utworzenia kopii zapasowej, niezależnie od zadań utrzymania. Wspominałem również, że nawet jeśli zadanie tworzenia kopii zapasowej napotka na błąd, metoda `__exit__` nadal zostanie wywołana.

Zwróć uwagę na sygnaturę metody `__exit__`. Metoda pobiera wartości dla wyjątku, który został zgłoszony w bloku. Jeśli nie było wyjątku w bloku, wszystkie te argumenty mają wartość `None`.

Należy uważnie się zastanowić nad wartością zwracaną przez metodę `__exit__`. Zwykle należy pozostawić tę metodę bez zwracania niczego konkretnego. Jeśli metoda zwraca wartość `True`, oznacza to, że potencjalnie zgłoszony wyjątek nie będzie propagowany do kodu wywołującego. Czasami jest to pożądany efekt, zwłaszcza w przypadku niektórych typów zgłaszanych wyjątków, ale ogólnie rzecz biorąc, „połykanie wyjątków" nie jest dobrym pomysłem. Zapamiętaj, że błędy nigdy nie powinny przechodzić bezgłośnie.

> Pamiętaj, aby przypadkowo nie zwrócić `True` z funkcji `_exit__`. Jeśli tak zrobisz, upewnij się, że właśnie tego chcesz i że jest ku temu dobry powód.

Implementacja menedżerów kontekstu

Ogólnie rzecz biorąc, możemy zaimplementować własne menedżery kontekstu, podobne do tego, który pokazałem w poprzednim przykładzie. Jedyne, co trzeba zrobić, to utworzyć klasę, która implementuje magiczne metody `__enter__` i `__exit__`. Obiekt tej klasy może obsługiwać protokół menedżera kontekstu. Chociaż jest to najczęstszy sposób implementacji menedżerów kontekstu, nie jest jedyny.

W tym punkcie nie tylko pokażę różne (czasami bardziej kompaktowe) sposoby implementacji menedżerów kontekstu, ale także opowiem, jak je w pełni wykorzystać za pośrednictwem biblioteki standardowej, a szczególnie z użyciem modułu `contextlib`.

Moduł `contextlib` zawiera wiele funkcji pomocniczych i obiektów do implementacji menedżerów kontekstu lub korzystania z menedżerów gotowych. Dzięki temu możemy pisać bardziej kompaktowy kod.

Zacznijmy od przyjrzenia się dekoratorowi `contextmanager`.

Po zastosowaniu do funkcji dekoratora `contextlib.contextmanager` kod tej funkcji jest konwertowany na menedżera kontekstu. Funkcja, do której zastosowano dekorator, musi być szczególnym rodzajem funkcji zwanej funkcją generatorową, która podzieli instrukcje na to, co stanie się metodami magicznymi odpowiednio `__enter__` i `__exit__`.

Jeśli w tym momencie nie wiesz, czym są dekoratory i generatory, nie przejmuj się. Przykłady, którym się przyjrzymy, będą samodzielne, a recepturę bądź idiom będzie można zastosować i zrozumieć niezależnie od rozumienia wspomnianych pojęć. Tematy dekoratorów i generatorów zostały szczegółowo omówione w rozdziale 7., „Generatory, iteratory i programowanie asynchroniczne”.

Kod równoważny poprzedniemu przykładowi można przepisać z wykorzystaniem dekoratora `contextmanager` w następujący sposób:

```
import contextlib

@contextlib.contextmanager
def db_handler():
    try:
        stop_database()
        yield
    finally:
        start_database()

with db_handler():
    db_backup()
```

W tym przykładzie zdefiniowaliśmy funkcję generatorową i zastosowaliśmy do niej dekorator `@contextlib.contextmanager`. Funkcja zawiera instrukcję `yield`, co czyni ją funkcją generatorową. Jak wspominałem wcześniej, szczegóły dotyczące generatorów nie są w tym przypadku istotne. Wszystko, co musisz wiedzieć, to to, że po zastosowaniu tego dekoratora wszystko

przed instrukcją `yield` zostanie uruchomione tak, jakby było częścią metody `__enter__`. Następnie uzyskana wartość będzie wynikiem oceny menedżera kontekstu (tym, co zwróciłaby metoda `__enter__`) i co zostałoby przypisane do zmiennej, gdybyśmy zdecydowali się przypisać ją za pomocą frazy `as x:` — w tym przypadku nic nie jest zwracane (co oznacza, że uzyskana wartość będzie niejawnie obiektem `None`), ale gdybyśmy chcieli, moglibyśmy zastosować instrukcję `yield`, zwracającą obiekt, którego moglibyśmy użyć w bloku menedżera kontekstu.

W tym momencie funkcja generatorowa zostaje zawieszona i wprowadzany jest menedżer kontekstu, w którym uruchamiamy kod wykonujący kopię zapasową naszej bazy danych. Po zakończeniu wykonywania kopii zapasowej sterowanie zostanie wznowione, więc możemy uznać, że każdy wiersz występujący za instrukcją `yield` będzie częścią logiki metody `__exit__`.

Pisanie takich menedżerów kontekstu ma tę zaletę, że łatwiej zrefaktoryzować istniejące funkcje, wielokrotnie korzystać z kodu i ogólnie jest dobrym pomysłem, gdy potrzebujemy menedżera kontekstu, który nie należy do żadnego konkretnego obiektu (w przeciwnym razie musiałbyś utworzyć „fałszywą" klasę bez prawdziwego celu w sensie obiektowym).

Wprowadzenie dodatkowych magicznych metod sprawiłoby, że kolejny obiekt naszej domeny byłby bardziej sprzężony i miałby szerszą odpowiedzialność oraz obsługiwałby działania, których prawdopodobnie nie powinien obsługiwać. Kiedy potrzebujesz jedynie funkcji menedżera kontekstu, bez zachowywania wielu stanów, całkowicie odizolowanej i niezależnej od reszty klas, pokazana droga jest prawdopodobnie dobra.

Istnieje jednak więcej sposobów, na jakie możemy implementować menedżery kontekstu. Jak już wspominałem, kluczem do ich tworzenia jest pakiet `contextlib` z biblioteki standardowej.

Innym narzędziem pomocniczym, którego moglibyśmy użyć, jest klasa `contextlib.ContextDecorator`. Jest to klasa bazowa, która dostarcza logiki zastosowania dekoratora do funkcji. Dzięki niej funkcja będzie działać wewnątrz menedżera kontekstu. Logikę samego menedżera kontekstu zapewnia implementacja wymienionych wyżej magicznych metod. W rezultacie powstaje klasa, która dla funkcji działa jako dekorator lub która może być umieszczona w hierarchii innych klas tak, aby zachowywały się jak menedżery kontekstu.

Aby skorzystać z tego mechanizmu, należy rozszerzyć klasę bazową i zaimplementować logikę dla wymagany metod:

```
class dbhandler_decorator(contextlib.ContextDecorator):
    def __enter__(self):
        stop_database()
        return self

    def __exit__(self, ext_type, ex_value, ex_traceback):
        start_database()

@dbhandler_decorator()
def offline_backup():
    run("pg_dump database")
```

Czy zauważyłeś coś, co wyróżnia ten przykład w porównaniu do pokazanych wcześniej? Nie ma instrukcji `with`. Wystarczy wywołać funkcję, a funkcja `offline_backup()` automatycznie uruchomi się w menedżerze kontekstu. Jest to logika, którą zapewnia klasa bazowa, pozwalająca używać jej jako dekoratora opakowującego oryginalną funkcję tak, aby działała wewnątrz menedżera kontekstu.

Jedynym minusem tego podejścia jest to, że w związku ze sposobem, w jaki działają obiekty, są one całkowicie niezależne (co jest dobrą cechą) — dekorator nie wie nic o funkcji, którą dekoruje i odwrotnie. Choć jest to dobre, jednak oznacza, że funkcja `offline_backup` nie może uzyskać dostępu do obiektu dekoratora, gdyby tego potrzebowała. Nic jednak nie stoi na przeszkodzie, abyśmy mogli wywołać ten dekorator wewnątrz funkcji, aby uzyskać do niego dostęp.

Można to zrobić w następujący sposób:

```
def offline_backup():
    with dbhandler_decorator() as handler: ...
```

Dzięki temu, że klasa jest dekoratorem, zyskujemy również to, że logika jest definiowana tylko raz i możemy ją wielokrotnie wykorzystać tyle razy, ile chcemy. Wystarczy, że zastosujemy dekoratory do innych funkcji wymagających tej samej, niezmiennej logiki.

Przyjrzyjmy się ostatniej własności pakietu `contextlib`, aby zobaczyć, czego możemy oczekiwać od menedżerów kontekstu i zorientować się, do czego można je zastosować.

W pakiecie `contextlib` dostępne jest narzędzie `contextlib.suppress`, pozwalające uniknąć pewnych wyjątków w sytuacjach, w których wiemy, że można je bezpiecznie zignorować. Jest to podobne do uruchamiania tego samego kodu w bloku `try/except` i przekazywania wyjątku lub po prostu rejestrowania go, ale różnica polega na tym, że wywołanie metody `suppress` sprawia, że fakt, iż wyjątki są kontrolowane w ramach naszej logiki, jest bardziej jawny.

Rozważmy następujący kod:

```
import contextlib

with contextlib.suppress(DataConversionException):
    parse_data(input_json_or_dict)
```

W tym przypadku obecność wyjątku oznacza, że dane wejściowe są już w oczekiwanym formacie, więc nie ma potrzeby konwersji. W związku z tym ten wyjątek można bezpiecznie zignorować.

Menedżery kontekstu są dość osobliwą cechą, która wyróżnia Pythona na tle innych języków. Dlatego korzystanie z menedżerów kontekstu można uznać za idiomatyczne. W następnym podrozdziale przyjrzymy się innej interesującej cesze Pythona, która pomoże napisać bardziej zwięzły kod; są to wyrażenia składane i wyrażenia przypisania.

Wyrażenia składane i wyrażenia przypisania

Z wyrażeniami składanymi zetkniemy się w tej książce wiele razy. To dlatego, że zwykle zapewniają one bardziej zwięzły sposób pisania kodu, a poza tym kod napisany z użyciem wyrażeń składanych jest zazwyczaj łatwiejszy do czytania. Powiedziałem zazwyczaj, ponieważ czasami, jeśli trzeba wykonać pewne transformacje zbieranych danych, użycie wyrażeń składanych może prowadzić do bardziej skomplikowanego kodu. W takich przypadkach należy raczej stosować proste pętle `for`.

Istnieje jednak ostatnia deska ratunku, którą można zastosować w takiej sytuacji: mianowicie wyrażenia przypisania. W tym podrozdziale omówię obie te konstrukcje.

Wyrażenia składane zaleca się stosować w celu tworzenia struktur danych za pomocą pojedynczej instrukcji, zamiast z wykorzystaniem wielu instrukcji. Aby na przykład utworzyć listę zawierającą wyniki obliczeń na liczbach, zamiast zapisywania kodu:

```
numbers = []
for i in range(10):
    numbers.append(run_calculation(i))
```

możemy utworzyć listę wyników bezpośrednio:

```
numbers = [run_calculation(i) for i in range(10)]
```

Kod napisany w tej formie zwykle działa lepiej, ponieważ używa pojedynczej instrukcji Pythona zamiast wielokrotnego wywoływania instrukcji `list.append`. Jeśli interesują Cię wewnętrzne mechanizmy działania Pythona albo różnice pomiędzy wersjami kodu, możesz skorzystać z modułu `dis` i wywołać go dla przedstawionych przykładów.

Zobaczmy przykład funkcji, która pobiera kilka ciągów reprezentujących zasoby w środowisku chmury obliczeniowej (na przykład ARN) i zwróci zbiór z identyfikatorami kont znalezionymi w tych ciągach. Oto najbardziej naiwny sposób napisania takiej funkcji:

```
from typing import Iterable, Set

def collect_account_ids_from_arns(arns: Iterable[str]) -> Set[str]:
    """Na podstawie kilku identyfikatorów ARN w postaci

      arn:partition:service:region:account-id:resource-id

    Pobierz unikatowe identyfikatory kont znalezione wewnątrz tych ciągów i je zwróć.
    """
    collected_account_ids = set()
    for arn in arns:
        matched = re.match(ARN_REGEX, arn)
        if matched is not None:
            account_id = matched.groupdict()["account_id"]
            collected_account_ids.add(account_id)
    return collected_account_ids
```

Wyraźnie widać, że ten kod składa się z wielu wierszy, pomimo że robi coś stosunkowo prostego. Czytelnik tego kodu może być zdezorientowany przez te liczne instrukcje, a podczas pracy z tym kodem może nieumyślnie popełnić błąd. Byłoby lepiej, gdyby można było go uprościć. Tę samą funkcjonalność możemy zaimplementować za pomocą mniejszej liczby wierszy. Wystarczy użyć kilku wyrażeń składanych w konstrukcji przypominającej programowanie funkcyjne:

```
def collect_account_ids_from_arns(arns):
    matched_arns = filter(None, (re.match(ARN_REGEX, arn) for arn in arns))
    return {m.groupdict()["account_id"] for m in matched_arns}
```

Pierwszy wiersz funkcji wydaje się podobny do wykorzystania funkcji `map` i `filter`: najpierw stosujemy wynik próby dopasowania wyrażenia regularnego do wszystkich dostarczonych ciągów, a następnie filtrujemy te, które są różne od `None`. W rezultacie otrzymujemy iterator, którego później użyjemy do wyodrębnienia identyfikatora konta za pomocą wyrażenia zbioru składanego.

Poprzednia funkcja jest łatwiejsza w utrzymaniu od pierwszego przykładu, ale nadal wymaga dwóch instrukcji. Przed wydaniem Pythona 3.8 osiągnięcie bardziej kompaktowej wersji było niemożliwe. Jednak wraz z wprowadzeniem wyrażeń przypisania w dokumencie PEP-572 (*https://www.python.org/dev/peps/pep-0572/*) możemy uzyskać ten sam wynik za pomocą pojedynczej instrukcji:

```
def collect_account_ids_from_arns(arns: Iterable[str]) -> Set[str]:
    return {
        matched.groupdict()["account_id"]
        for arn in arns
        if (matched := re.match(ARN_REGEX, arn)) is not None
    }
```

Zwróć uwagę na składnię w trzecim wierszu wewnątrz wyrażenia składanego. Instrukcja ustawia tymczasowy identyfikator wewnątrz zakresu, co jest wynikiem zastosowania wyrażenia regularnego do ciągu. Instrukcja może być ponownie użyta w innych fragmentach tego samego zakresu.

W tym konkretnym przykładzie można by się spierać, czy trzeci przykład jest lepszy od drugiego (ale nie powinno być wątpliwości, że oba są lepsze od pierwszego!). Uważam, że ostatni przykład jest bardziej ekspresyjny, ponieważ jest mniej wnioskowania w kodzie, a wszystko, co czytelnik musi wiedzieć o tym, jak są pobierane wartości, należy do tego samego zakresu.

Należy pamiętać, że bardziej kompaktowy kod nie zawsze oznacza lepszy kod. Jeśli w celu napisania jednowierszowca musimy utworzyć zawiłe wyrażenie, nie warto tego robić i lepiej zastosować naiwne podejście. Jest to związane z zasadą *zachowaj prostotę*, którą omówię w następnym rozdziale.

> Weź pod uwagę czytelność wyrażeń składanych i nie staraj się za wszelką cenę, aby Twój kod był „jednowierszowcem", jeśli dążenie do tego utrudni zrozumienie kodu.

Innym dobrym powodem używania wyrażeń przypisania w ogóle (nie tylko w wyrażeniach składanych) są względy wydajności. Jeśli musimy użyć funkcji jako części naszej logiki transformacji, nie chcemy wywoływać jej częściej, niż to konieczne. Przypisanie wyniku funkcji do tymczasowego identyfikatora (tak jak w przypadku przypisywania wyrażeń w nowych zakresach) to dobra technika optymalizacji, która jednocześnie sprawia, że kod staje się bardziej czytelny.

> Oceń ulepszenia wydajności, które można uzyskać za pomocą wyrażeń przypisania.

W następnym podrozdziale omówię inną idiomatyczną cechę Pythona, czyli *właściwości*. Ponadto opiszę różne sposoby ujawniania lub ukrywania danych w obiektach Pythona.

Właściwości, atrybuty i różne typy metod obiektów

Wszystkie właściwości i funkcje obiektu w języku Python są publiczne. Pod tym względem Python różni się od innych języków, w których właściwości mogą być publiczne, prywatne lub chronione. Oznacza to, że uniemożliwianie obiektom wywołującym odwoływanie się do jakichkolwiek atrybutów obiektu nie ma sensu. Jest to kolejna różnica w porównaniu z innymi językami programowania, w których niektóre atrybuty można oznaczyć jako prywatne lub chronione.

Nie istnieje mechanizm ścisłego egzekwowania, ale istnieją pewne konwencje. Atrybut, którego nazwa zaczyna się od znaku podkreślenia, ma być prywatny dla tego obiektu i oczekujemy, że żaden zewnętrzny agent go nie wywoła (ale, jak wspominałem, nic tego nie uniemożliwia).

Zanim przejdziemy do szczegółów właściwości, warto wspomnieć o kilku cechach znaków podkreślenia w Pythonie, a także o rozumieniu konwencji i zakresie atrybutów.

Znaki podkreślenia w Pythonie

Istnieją pewne konwencje i szczegóły implementacji, które wykorzystują podkreślenia w Pythonie. Jest to interesujący temat, który warto poddać analizie.

Jak wspomniałem wcześniej, domyślnie wszystkie atrybuty obiektu są publiczne. Aby to zilustrować, rozważmy następujący przykład:

```
>>> class Connector:
...     def __init__(self, source):
...         self.source = source
...         self._timeout = 60
...
>>> conn = Connector("postgresql://localhost")
```

```
>>> conn.source
'postgresql://localhost'
>>> conn._timeout
60
>>> conn.__dict__
{'source': 'postgresql://localhost', '_timeout': 60}
```

W tym przypadku obiekt `Connector` jest tworzony z parametrem `source`, a jego metoda `__init__` zaczyna się od przypisania dwóch atrybutów — wspomnianego `source` oznaczającego źródło i `_timeout` określającego limit czasu. Ten pierwszy jest publiczny, a drugi prywatny. Jednak, jak widać na podstawie poniższego kodu, podczas tworzenia obiektu `Connector` bez trudu możemy uzyskać dostęp do obu tych atrybutów.

Interpretacja tego kodu jest taka, że dostęp do atrybutu `_timeout` powinien być możliwy tylko w samym obiekcie `Connector`, a nigdy z kodu wywołującego. Oznacza to, że powinieneś zorganizować kod w taki sposób, aby można było zawsze bezpiecznie zrefaktoryzować limit czasu, gdy jest to potrzebne, opierając się na fakcie, że atrybut ten nie jest wywoływany spoza obiektu (tylko wewnątrz niego). Taka refaktoryzacja w żaden sposób nie wpływa na interfejs. Przestrzeganie tych reguł sprawia, że kod jest łatwiejszy w utrzymaniu i bardziej niezawodny, ponieważ nie musimy się martwić o skutki uboczne podczas refaktoryzacji kodu, jeśli dbamy o utrzymanie interfejsu obiektu. Identyczna zasada dotyczy również metod.

> Klasy powinny eksponować tylko te atrybuty i metody, które są istotne dla zewnętrznego obiektu wywołującego, to znaczy te, które składają się na jego interfejs. Wszystkie atrybuty, które nie są częścią interfejsu obiektu, powinny być poprzedzone pojedynczym znakiem podkreślenia.

Atrybuty, których nazwy zaczynają się od znaku podkreślenia, powinny być traktowane jako prywatne i nie mogą być wywoływane z zewnątrz. Z drugiej strony, jako wyjątek od tej reguły, możemy powiedzieć, że w testach jednostkowych możemy dopuścić dostęp do atrybutów wewnętrznych, jeśli ułatwia to testowanie (zauważ jednak, że skorzystanie z tego pragmatycznego podejścia nadal wiąże się z pewnymi kosztami utrzymania, gdy zdecydujesz się na refaktoryzację głównej klasy). Warto jednak pamiętać o następujących zaleceniach.

> Używanie zbyt wielu wewnętrznych metod i atrybutów może być oznaką, że klasa wykonuje zbyt wiele zadań i narusza zasadę pojedynczej odpowiedzialności. Może to być znak, że trzeba wyodrębnić niektóre z jej obowiązków do klas współpracujących.

Używanie pojedynczego znaku podkreślenia jako prefiksu jest pythonicznym sposobem wyraźnego rozgraniczenia interfejsu obiektu. Istnieje jednak powszechne błędne przekonanie, że niektóre atrybuty i metody można faktycznie zmienić na prywatne. Wyobraźmy sobie, że teraz atrybut określający limit czasu został zdefiniowany jako `__timeout`, z wiodącym podwójnym podkreśleniem:

```
>>> class Connector:
...     def __init__(self, source):
...         self.source = source
...         self.__timeout = 60
...
...     def connect(self):
...         print("połączenie z limitem czasu {0}s".format(self.__timeout))
...         # ...
...
>>> conn = Connector("postgresql://localhost")
>>> conn.connect()
połączenie z limitem czasu 60s
>>> conn.__timeout
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Connector' object has no attribute '__timeout'
```

Niektórzy programiści stosują ten sposób do ukrywania niektórych atrybutów, myśląc, jak w tym przykładzie, że taki atrybut będzie rzeczywiście prywatny i żaden inny obiekt nie będzie mógł go zmodyfikować. Spójrz teraz na wyjątek zgłaszany podczas próby uzyskania dostępu do atrybutu `__timeout`. Ten wyjątek to `AttributeError`, mówiący, że taki atrybut nie istnieje. Nie mówi czegoś w stylu „jest prywatny" lub „nie można uzyskać do niego dostępu". Mówi, że nie istnieje. To powinno dać nam wskazówkę, że w rzeczywistości dzieje się coś innego i to zachowanie jest tylko efektem ubocznym, ale nie rzeczywistym zaplanowanym zachowaniem.

W rzeczywistości chodzi o to, że po zastosowaniu podwójnego znaku podkreślenia Python tworzy inną nazwę atrybutu (określa się to jako *mangling*). Zamiast atrybutu z podwójnym znakiem podkreślenia Python tworzy atrybut o nazwie: *`_<nazwa-klasy>__<nazwa atrybutu>`*. W tym przypadku zostanie utworzony atrybut o nazwie `_Connector__timeout`, do którego można bez trudu uzyskać dostęp (i zmodyfikować go) w następujący sposób:

```
>>> vars(conn)
{'source': 'postgresql://localhost', '_Connector__timeout': 60}
>>> conn._Connector__timeout
60
>>> conn._Connector__timeout = 30
>>> conn.connect()
Połączenie z limitem czasu 30s
```

Zwróć uwagę na efekt uboczny, o którym wspominałem wcześniej — atrybut nadal istnieje, tylko o innej nazwie, i z tego powodu przy pierwszej próbie uzyskania dostępu do atrybutu został zgłoszony wyjątek `AttributeError`.

Idea stosowania w Pythonie symbolu podwójnego podkreślenia jest zupełnie inna. Powstał jako mechanizm zastępowania różnych metod klasy, które są kilkakrotnie rozszerzane, aby nie stwarzać zagrożenia kolizji z nazwami metod. Nawet to jest zbyt naciąganym przypadkiem użycia, aby uzasadnić stosowanie tego mechanizmu.

Atrybuty o nazwach z podwójnym znakiem podkreślenia to podejście niepythoniczne. Jeśli chcesz zdefiniować atrybuty jako prywatne, użyj pojedynczego podkreślenia i przestrzegaj pythonicznej konwencji, że jest to atrybut prywatny.

> Nie definiuj atrybutów z wiodącymi podwójnymi podkreśleniami. Z tego samego powodu nie definiuj własnych metod „dunder" (metod, których nazwy są otoczone podwójnymi podkreśleniami).

Przyjrzyjmy się teraz odwrotnemu przypadku, to znaczy sytuacji, gdy chcemy uzyskać dostęp do niektórych atrybutów obiektu, które mają być publiczne. Zazwyczaj używamy do tego *właściwości*, które omówię w następnym punkcie.

Właściwości

Zazwyczaj w projektowaniu obiektowym tworzymy obiekty do reprezentowania abstrakcji nad podmiotem należącym do dziedziny problemu. W tym sensie obiekty mogą hermetyzować zachowania lub dane. Często o tym, czy obiekt może zostać utworzony, czy nie, decyduje dokładność danych. Oznacza to, że niektóre podmioty mogą istnieć tylko dla pewnych wartości danych, podczas gdy nieprawidłowe wartości nie powinny być dozwolone.

Dlatego właśnie tworzymy metody walidacji, zwykle używane w operacjach setterów. W Pythonie czasami jednak możemy hermetyzować metody setterów i getterów, by korzystać z nich bardziej kompaktowo, za pomocą *właściwości*.

Rozważmy przykład systemu geograficznego, który musi posługiwać się współrzędnymi. Istnieje tylko pewien zakres wartości, dla których szerokość i długość geograficzna mają sens. Poza tymi wartościami współrzędne nie mogą istnieć. Możemy utworzyć obiekt reprezentujący współrzędną, ale robiąc to, musimy zadbać o to, aby wartości szerokości geograficznej zawsze mieściły się w dopuszczalnych zakresach. Do tego celu możemy użyć właściwości:

```
class Coordinate:
    def __init__(self, lat: float, long: float) -> None:
        self._latitude = self._longitude = None
        self.latitude = lat
        self.longitude = long

    @property
    def latitude(self) -> float:
        return self._latitude

    @latitude.setter
    def latitude(self, lat_value: float) -> None:
        if lat_value not in range(-90, 90 + 1):
            raise ValueError(f"{lat_value} to nieprawidłowa wartość dla szerokości
            ↪geograficznej")
            self._latitude = lat_value

    @property
    def longitude(self) -> float:
        return self._longitude

    @longitude.setter
```

```
    def longitude(self, long_value: float) -> None:
        if long_value not in range(-180, 180 + 1):
            raise ValueError(f"{long_value} to nieprawidłowa wartość dla długości
            ↪geograficznej")
        self._longitude = long_value
```

Tutaj użyliśmy właściwości do zdefiniowania szerokości i długości geograficznej. W ten sposób ustaliliśmy, że pobranie dowolnego z tych atrybutów zwróci wewnętrzną wartość przechowywaną w zmiennych prywatnych. Co ważniejsze, gdy dowolny użytkownik zechce zmodyfikować wartości dowolnej z tych właściwości w następującej formie:

```
coordinate.latitude = <nowa-wartość-szerokości-geograficznej> # podobnie dla
↪właściwości longitude
```

to zostanie automatycznie (i przezroczyście) wywołana metoda walidacji zadeklarowana za pomocą dekoratora `@latitude.setter`. Do tej metody zostanie przekazana jako parametr (w powyższym kodzie `lat_value`) wartość po prawej stronie instrukcji (*<nowa-wartość-szerokości-geograficznej>*).

> Nie pisz niestandardowych metod `get_*` i `set_*` dla wszystkich atrybutów obiektów. W większości przypadków wystarczy pozostawić je jako zwykłe atrybuty. Jeśli chcesz zmodyfikować logikę podczas pobierania lub modyfikowania atrybutu, użyj *właściwości*.

Widzieliśmy przypadek, w którym obiekt musiał przechowywać wartości. Pokazałem, że właściwości pomagają zarządzać ich wewnętrznymi danymi w spójny i przejrzysty sposób. Czasami jednak potrzebujemy wykonać pewne obliczenia w oparciu o stan obiektu i jego wewnętrzne dane. W większości przypadków właściwości dobrze nadają się do tego celu.

Jeśli na przykład masz obiekt, który powinien zwrócić wartość w określonym formacie lub określonego typu danych, do wykonania tych obliczeń możesz użyć właściwości. Gdybyśmy w poprzednim przykładzie zdecydowali się, że chcemy zwrócić współrzędne z dokładnością do czterech miejsc po przecinku (niezależnie od tego, ile miejsc dziesiętnych zawierała podana liczba przekazana jako parametr), moglibyśmy wykonać obliczenia związane z zaokrąglaniem w metodzie `@property`, która odczytuje wartość.

Może się okazać, że właściwości są dobrym sposobem na uzyskanie separacji poleceń od zapytań (CC08). Zasada separacji poleceń od zapytań mówi, że metoda obiektu powinna albo odpowiadać na coś, albo coś robić, ale nie jedno i drugie. Jeśli metoda coś robi, a jednocześnie zwraca status odpowiadający na pytanie, jak przebiegała ta operacja, to robi więcej niż jedną rzecz, wyraźnie naruszając zasadę, która mówi, że funkcje powinny robić jedną rzecz i tylko jedną rzecz.

W zależności od nazwy metody może to spowodować jeszcze więcej zamieszania i utrudnić czytelnikom kodu zrozumienie, jaka jest jego rzeczywista intencja. Jeśli na przykład metoda nazywa się `set_email` i używamy jej w postaci `if self.set_email("a@j. com"):` ..., to co robi ten kod? Czy ustawia e-mail na *a@j.com*? A może sprawdza, czy wiadomość e-mail jest już ustawiona na tę wartość? A może robi jedno i drugie (ustawia e-mail, a następnie sprawdza, czy status jest poprawny)?

Dzięki właściwościom możemy uniknąć tego rodzaju zamieszania. Dekorator `@property` definiuje zapytanie, które na coś odpowiada, a `@<nazwa_właściwości>.setter` to polecenie, które coś robi.

Kolejna dobra rada płynąca z tego przykładu jest następująca — nie rób w metodzie więcej niż jednej rzeczy. Jeśli chcesz coś przypisać, a następnie sprawdzić wartość, podziel te działania na dwie lub więcej instrukcji.

Aby zilustrować, co to oznacza, skorzystamy z poprzedniego przykładu. Mielibyśmy w kodzie jedną metodę settera lub gettera, które ustawiałyby adres e-mail użytkownika, a następnie inną metodę właściwości, która zwracałaby adres e-mail. To dlatego, że ogólnie rzecz biorąc, za każdym razem, gdy pytamy obiekt o jego obecny stan, powinien on zwrócić go bez skutków ubocznych (bez zmiany jego wewnętrznej reprezentacji). Być może jedynym wyjątkiem od tej reguły, jaki przychodzi mi na myśl, jest przypadek tak zwanej leniwej właściwości: atrybutu, który chcemy wstępnie obliczyć tylko raz, a następnie używać wartości obliczonej. W pozostałych przypadkach staraj się, aby właściwości były idempotentne i definiuj metody, które mogą zmieniać wewnętrzną reprezentację obiektu, ale nie mieszaj jednych z drugimi.

> Metody powinny robić tylko jedną rzecz. Jeśli musisz uruchomić działanie, a następnie sprawdzić jego stan, zrób to w dwóch oddzielnych metodach, które są wywoływane w różnych instrukcjach.

Tworzenie klas o bardziej zwartej składni

Kontynuując ideę, że czasami potrzebujemy obiektów do przechowywania wartości, chciałbym zaprezentować powszechnie stosowany w Pythonie szablon dotyczący inicjowania obiektów. Polega on na zadeklarowaniu w metodzie `__init__` wszystkich atrybutów, którymi obiekt będzie się posługiwał, a następnie ustawieniu ich na wewnętrzne zmienne. Zwykle ma to następującą formę:

```
def __init__(self, x, y, … ):
    self.x = x
    self.y = y
```

Od opublikowania Pythona 3.7 możemy ten szablon uprościć za pomocą modułu `dataclasses`. Mechanizm ten wprowadzono w dokumencie PEP-557. Z tym modułem zetknęliśmy się w poprzednim rozdziale, w kontekście używania adnotacji w kodzie. W tym podrozdziale krótko go przejrzymy pod kątem wspomagania pisania bardziej zwięzłego kodu.

Wspomniany moduł udostępnia dekorator `@dataclass`, który po zastosowaniu do klasy pobiera wszystkie atrybuty klasy z adnotacjami i traktuje je jak atrybuty egzemplarza, tak jakby zostały zadeklarowane w metodzie `__init__`. Jeśli skorzystamy z tego dekoratora, metoda `__init__` klasy zostanie wygenerowana automatycznie, więc nie musimy tego robić sami.

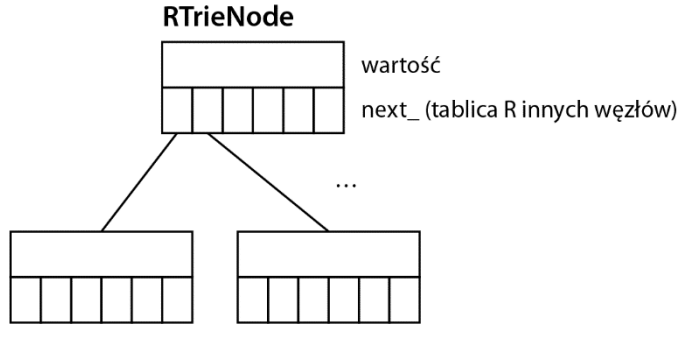
Dodatkowo moduł ten udostępnia obiekt `field`, który pomaga zdefiniować poszczególne cechy niektórych atrybutów. Jeśli na przykład jeden z atrybutów, którego potrzebujesz, musi być mutowalny (na przykład jest listą), to — jak zobaczymy dalej w tym rozdziale (w punkcie poświęconym unikaniu haczyków Pythona) — nie możemy przekazać domyślnej pustej listy w metodzie `__init__`. Zamiast tego powinniśmy przekazać `None` i ustawić atrybut na domyślną listę wewnątrz metody `__ init__`.

Gdy posługujemy się obiektem `field`, zamiast tego sposobu możemy użyć argumentu `default_ factory` i przekazać do niego klasę `list`. Ten argument ma być używany z obiektem wywoływalnym, który nie przyjmuje argumentów i zostanie wywołany w celu skonstruowania obiektu, gdy dla wartości tego atrybutu nie zostanie podana żadna wartość.

Co zrobić w ramach walidacji, skoro nie ma metody `__init__` do zaimplementowania? Albo co zrobić, jeśli jakieś atrybuty mają być obliczone lub wyprowadzone z innych? Odpowiem na to drugie pytanie: możemy skorzystać z *właściwości*, które omówiłem w poprzednim podrozdziale. Jeśli chodzi o pierwsze pytanie, klasy danych pozwalają na zdefiniowanie metody `__post_init__`, która zostanie automatycznie wywołana przez metodę `__init__`, więc jest dobrym miejscem do napisania logiki postinicjacji.

Aby zaprezentować te pojęcia w praktyce, rozważmy przykład modelowania węzła dla struktury danych R-Trie (gdzie R to promień — ang. *radix*, co oznacza, że jest indeksowanym drzewem nad jakąś bazą R). Szczegóły tej struktury danych i algorytmów z nią związanych wykraczają poza zakres tej książki, ale na potrzeby przykładu wspomnę, że jest to struktura danych zaprojektowana do odpowiadania na zapytania dotyczące tekstu lub ciągów znaków (takich jak prefiksy lub znajdowanie słów podobnych bądź powiązanych). W bardzo podstawowej formie ta struktura danych zawiera wartość (ze znakiem, która może być na przykład jego reprezentacją całkowitą) oraz tablicę o długości R z odwołaniami do następnych węzłów (jest to rekurencyjna struktura danych, w tym samym sensie, co na przykład lista powiązana lub drzewo). Chodzi o to, że każda pozycja tablicy definiuje niejawnie odwołanie do następnego węzła. Przykładowo wyobraź sobie, że wartość 0 jest mapowana na znak `'a'`. Jeśli następny węzeł zawiera wartość inną niż `None` w swojej pozycji 0, oznacza to, że istnieje odwołanie do `'a'`, które wskazuje na inny węzeł R-Trie.

Graficznie struktura danych może wyglądać mniej więcej tak:



Rysunek 2.1. Ogólna struktura węzła R-Trie

Reprezentację powyższej struktury moglibyśmy zapisać za pomocą zamieszczonego poniżej bloku kodu. W tym kodzie atrybut o nazwie `next_` zawiera końcowe podkreślenie, aby odróżnić go od wbudowanej funkcji `next`. Moglibyśmy się spierać, że w tym przypadku nie ma kolizji, ale gdybyśmy musieli użyć funkcji `next()` w klasie `RTrieNode`, mogłoby to stanowić problem (zwykle takie kolizje prowadzą do powstania trudnych do wychwycenia, subtelnych błędów):

```
from typing import List
from dataclasses import dataclass, field

R = 26

@dataclass
class RTrieNode:
    size = R
    value: int
    next_: List["RTrieNode"] = field(
        default_factory=lambda: [None] * R)

    def __post_init__(self):
        if len(self.next_) != self.size:
            raise ValueError(f"Nieprawidłowa długość listy next")
```

Powyższy przykład zawiera kilka różnych kombinacji. Najpierw definiujemy strukturę R-Trie z `R = 26`, w celu reprezentacji znaków w alfabecie angielskim (nie jest to ważne dla samego zrozumienia kodu, ale podaje szerszy kontekst). Idea jest taka, że jeśli chcemy przechowywać słowo, tworzymy węzeł dla każdej litery, zaczynając od pierwszej. Gdy istnieje link do następnego znaku, zapisujemy go w pozycji tablicy `next_` odpowiadającej temu znakowi, innemu węzłowi dla tego znaku i tak dalej.

Zwróć uwagę na pierwszy atrybut w klasie, czyli `size`. Nie ma adnotacji, więc jest to zwykły atrybut klasy (współdzielony dla wszystkich obiektów węzła), a nie coś, co należy wyłącznie do obiektu. Alternatywnie moglibyśmy to zdefiniować, ustawiając `field(init= False)`, ale ta forma jest bardziej zwarta. Gdybyśmy jednak chcieli dodać adnotację do zmiennej, ale nie chcieli uwzględniać jej w metodzie `__init__`, to ta składnia byłaby jedyną realną alternatywą.

Dalej są dwa inne atrybuty, z których oba mają adnotacje, ale z różnych powodów. Pierwszy z nich, `value`, jest liczbą całkowitą, ale nie ma argumentu domyślnego, więc kiedy tworzymy nowy węzeł, zawsze musimy podać wartość jako pierwszy parametr. Drugi jest argumentem mutowalnym (listą złożoną z węzłów tej samej klasy) i ma domyślną fabrykę: w tym przypadku funkcję lambda, która tworzy nową listę rozmiaru `R`, zainicjowaną samymi elementami `None`. Zauważ, że gdybyśmy użyli do tego celu adnotacji `field(default_ factory=list)`, nadal konstruowalibyśmy nową listę dla każdego obiektu podczas tworzenia, ale utracilibyśmy kontrolę nad rozmiarem tej listy. I, na koniec, chcieliśmy sprawdzić, czy nie tworzymy węzłów, które zawierają listę następnych węzłów o niewłaściwej długości, więc weryfikujemy to w metodzie `__post_init__`. Każda próba utworzenia takiej listy zostanie uniemożliwiona poprzez zgłoszenie w czasie inicjowania listy wyjątku `ValueError`.

> Klasy danych zapewniają bardziej kompaktowy sposób pisania klas. Pozwalają na uniknięcie konieczności ustawiania wszystkich zmiennych o tej samej nazwie w metodzie `__init__`.

Jeśli w aplikacji masz obiekty, które nie wykonują wielu złożonych walidacji lub przekształceń danych, rozważ tę alternatywę. Pamiętaj jednak o ważnej rzeczy. Adnotacje są świetne, ale nie wymuszają konwersji danych. Oznacza to, że jeśli na przykład zadeklarujesz atrybut, który musi być liczbą zmiennoprzecinkową (`float`) lub całkowitą (`integer`), musisz wykonać tę konwersję w metodzie `__init__`. Zapisanie tego jako klasy danych nie spowoduje konwersji i może ukryć subtelne błędy. Dotyczy to przypadków, gdy walidacje nie są ściśle wymagane i dozwolone jest rzutowanie typów. Przykładowo całkowicie prawidłowe jest zdefiniowanie obiektu, który można utworzyć z wielu innych typów, i tak możliwa jest konwersja liczby `float` z ciągu znaków reprezentujących liczbę (ostatecznie takie działanie wykorzystuje naturę dynamicznego typowania w Pythonie), pod warunkiem że wartość zostanie poprawnie skonwertowana na wymagany typ danych wewnątrz metody `__init__`.

Dobrym przypadkiem użycia klas danych są wszystkie miejsca, w których musimy używać obiektów jako kontenerów danych lub opakowań, czyli sytuacje, w których używaliśmy nazwanych krotek lub prostych przestrzeni nazw. Podczas rozważania różnych opcji w kodzie klasy danych należy traktować jak kolejną możliwość dla nazwanych krotek lub przestrzeni nazw.

Obiekty iterowalne

W Pythonie istnieją obiekty, po których można domyślnie iterować. Przykładowo listy, krotki, zbiory i słowniki mogą nie tylko przechowywać dane w żądanej strukturze, ale także być iterowane w pętli `for` w celu pobierania kolejnych wartości.

Jednak wbudowane obiekty iterowalne nie są jedynymi, które możemy wykorzystywać w pętli `for`. Możemy również utworzyć własny obiekt iterowalny z logiką zdefiniowaną dla iteracji.

W tym celu po raz kolejny polegamy na metodach magicznych.

Iteracja działa w Pythonie według własnego protokołu (protokołu iteratora). Kiedy próbujesz iterować po obiekcie w formie `for e in myobject:...`, to na bardzo wysokim poziomie Python sprawdza w kolejności następujące dwie rzeczy.

- Czy obiekt zawiera jedną z metod iteratora — `__next__` bądź `__iter__`?
- Czy obiekt jest sekwencją i zawiera metody `__len__` oraz `__getitem__`?

Dlatego, jako mechanizm rezerwowy, sekwencje mogą być iterowane. Z tego powodu istnieją dwa sposoby dostosowywania naszych obiektów w taki sposób, aby mogły być wykorzystywane w pętlach `for`.

Tworzenie obiektów iterowalnych

Gdy próbujemy iterować po obiekcie, Python wywołuje dla niego funkcję `iter()`. Jedną z pierwszych rzeczy, które sprawdza ta funkcja, jest obecność metody `__iter__` w tym obiekcie. Jeśli obiekt zawiera tę metodę, zostanie ona wykonana.

Poniższy kod tworzy obiekt, który umożliwia iterowanie po zakresie dat. W każdej iteracji pętli tworzy jeden dzień:

```
from datetime import timedelta

class DateRangeIterable:
    """Obiekt iterowalny zawierający własny obiekt iteratora."""

    def __init__(self, start_date, end_date):
        self.start_date = start_date
        self.end_date = end_date
        self._present_day = start_date

    def __iter__(self):
        return self

    def __next__(self):
        if self._present_day >= self.end_date:
            raise StopIteration()
        today = self._present_day
        self._present_day += timedelta(days=1)
        return today
```

Ten obiekt został zaprojektowany do utworzenia z wykorzystaniem pary dat. W przypadku iterowania po obiekcie będzie generował poszczególne dni z przedziału podanych dat, co widać w poniższym kodzie:

```
>>> from datetime import date
>>> for day in DateRangeIterable(date(2018, 1, 1), date(2018, 1, 5)):
...     print(day)
...
2018-01-01
2018-01-02
2018-01-03
2018-01-04
```

W tym przykładzie pętla `for` rozpoczyna nową iterację po naszym obiekcie. W tym momencie Python wywoła na tym obiekcie funkcję `iter()`, która z kolei wywoła magiczną metodę `__iter__`. Metoda ta zwraca `self`, co wskazuje, że obiekt jest sam w sobie iterowalny, więc w tym momencie w każdym kroku pętli zostanie wywołana na tym obiekcie funkcja `next()`, która deleguje sterowanie do metody `__next__`. Ta metoda decyduje o sposobie generowania elementów i zwracania ich po kolei. Gdy nie ma dalszych elementów do wygenerowania, musimy zasygnalizować to Pythonowi poprzez zgłoszenie wyjątku `StopIteration`.

A zatem to, co faktycznie się dzieje, jest podobne do działania Pythona polegającego na wywołaniu na naszym obiekcie w każdej iteracji metody `next()`, dopóki nie pojawi się wyjątek `StopIteration`. Wtedy wiadomo, że pętla `for` powinna być zatrzymana:

```
>>> r = DateRangeIterable(date(2018, 1, 1), date(2018, 1, 5))
>>> next(r)
datetime.date(2018, 1, 1)
>>> next(r)
datetime.date(2018, 1, 2)
>>> next(r)
datetime.date(2018, 1, 3)
>>> next(r)
datetime.date(2018, 1, 4)
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ... __next__
    raise StopIteration
StopIteration
>>>
```

Powyższy przykład działa, ale istnieje w nim pewien problem — po wyczerpaniu elementów obiekt iterowalny będzie ciągle pusty, co będzie powodowało zgłaszanie wyjątku `StopIteration`. Oznacza to, że jeśli użyjemy go w dwóch lub większej liczbie kolejnych pętli `for`, tylko pierwsza z nich będzie działać, podczas gdy druga będzie próbowała sięgać do pustej sekwencji:

```
>>> r1 = DateRangeIterable(date(2018, 1, 1), date(2018, 1, 5))
>>> ", ".join(map(str, r1))

'2018-01-01, 2018-01-02, 2018-01-03, 2018-01-04'
>>> max(r1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
>>>
```

Wynika to ze sposobu działania protokołu iteracji — obiekt iterowalny konstruuje iterator i to nad nim są wykonywane iteracje. W naszym przykładzie metoda `__iter__` po prostu zwróciła `self`, ale możemy sprawić, że metoda ta utworzy nowy iterator za każdym razem, gdy zostanie wywołana. Jednym ze sposobów naprawienia tego problemu mogłoby być utworzenie nowych egzemplarzy klasy `DateRangeIterable`, co nie jest strasznym problemem. Możemy jednak wykorzystać w metodzie `__iter__` generator (będący obiektem iteratora), który zostanie utworzony za każdym razem:

```
class DateRangeContainerIterable:
    def __init__(self, start_date, end_date):
        self.start_date = start_date
        self.end_date = end_date

    def __iter__(self):
        current_day = self.start_date
        while current_day < self.end_date:
            yield current_day
            current_day += timedelta(days=1)
```

Tym razem wszystko działa:

```
>>> r1 = DateRangeContainerIterable(date(2018, 1, 1), date(2018, 1, 5))
>>> ", ".join(map(str, r1))
'2018-01-01, 2018-01-02, 2018-01-03, 2018-01-04'
>>> max(r1)
datetime.date(2018, 1, 4)
>>>
```

Różnica polega na tym, że każda pętla `for` wywołuje metodę `__iter__` ponownie, a każde wywołanie ponownie tworzy generator.

Nazywa się to kontenerem obiektów iterowalnych.

> Ogólnie rzecz biorąc, korzystanie z kontenerów obiektów iterowalnych podczas posługiwania się generatorami to dobry pomysł.

Szczegóły dotyczące generatorów wyjaśnię w rozdziale 7., „Generatory, iteratory i programowanie asynchroniczne”.

Tworzenie sekwencji

Być może obiekt nie definiuje metody `__iter__()`, a pomimo to chcemy mieć możliwość iterowania po tym obiekcie. Jeśli w obiekcie nie zostanie zdefiniowana metoda `__iter__`, funkcja `iter()` wyszuka obecność metody `__getitem__`, a jeśli ta także nie zostanie znaleziona, zgłosi wyjątek `TypeError`.

Sekwencja jest obiektem, który implementuje metody `__len__` i `__getitem__`. Oczekuje się od niej, że będzie można uzyskać zawarte w niej elementy, po kolei, zaczynając od zera jako pierwszego indeksu. Oznacza to, że należy zachować ostrożność w logice, aby poprawnie zaimplementować metodę `__getitem__` w celu właściwej interpretacji indeksów. W przeciwnym razie iteracja nie będzie działać.

Przykład z poprzedniego punktu ma tę zaletę, że zużywa mniej pamięci. Oznacza to, że obiekt przechowuje tylko jedną datę na raz i wie, jak generować kolejne. Ma jednak tę wadę, że jeśli chcemy uzyskać *n*-ty element, nie ma innego sposobu, aby to zrobić, niż wykonanie iteracji *n* razy, dopóki do niego nie dotrzemy. Jest to typowy kompromis w informatyce między wykorzystaniem pamięci a wykorzystaniem procesora.

Implementacja z obiektem iterowalnym zużywa mniej pamięci, ale złożoność uzyskania elementu jest rzędu *O(n)*, podczas gdy implementacja sekwencji zużywa więcej pamięci (ponieważ sekwencja musi przechowywać wszystkie elementy naraz), ale złożoność indeksowania jest rzędu *O(1)*.

Notacja wspomniana w poprzednim akapicie (na przykład *O(n)*) to tak zwana notacja asymptotyczna (lub notacja Big-O). Opisuje ona rzędy złożoności algorytmu. Na bardzo wysokim poziomie oznacza to stosunek liczby operacji, jakie algorytm musi wykonać, w funkcji rozmiaru

danych wejściowych (n). Aby uzyskać więcej informacji na ten temat, poszukaj w (ALGO01) — pozycji wymienionej na końcu rozdziału — gdzie znajdziesz szczegółowe informacje na temat notacji asymptotycznej.

Nowa implementacja może wyglądać następująco:

```
class DateRangeSequence:
    def __init__(self, start_date, end_date):
        self.start_date = start_date
        self.end_date = end_date
        self._range = self._create_range()

    def _create_range(self):
        days = []
        current_day = self.start_date
        while current_day < self.end_date:
            days.append(current_day)
            current_day += timedelta(days=1)
            return days

    def __getitem__(self, day_no):
        return self._range[day_no]

    def __len__(self):
        return len(self._range)
```

Oto jak zachowuje się obiekt:

```
>>> s1 = DateRangeSequence(date(2018, 1, 1), date(2018, 1, 5))
>>> for day in s1:
...     print(day)
...
2018-01-01
2018-01-02
2018-01-03
2018-01-04
>>> s1[0]
datetime.date(2018, 1, 1)
>>> s1[3]
datetime.date(2018, 1, 4)
>>> s1[-1]
datetime.date(2018, 1, 4)
```

W powyższym kodzie widać, że w sekwencji można wykorzystać również indeksy ujemne. To dlatego, że obiekt `DateRangeSequence` deleguje wszystkie operacje do obiektu opakowanego (listy), co jest najlepszym sposobem zapewnienia zgodności i spójnego zachowania.

> Podczas podejmowania decyzji o wykorzystaniu jednej z dwóch możliwych implementacji oceń kompromis pomiędzy użyciem pamięci i procesora. Ogólnie rzecz biorąc, iterowanie jest lepsze (a jeszcze lepsze są generatory), ale należy wziąć pod uwagę wymagania każdego przypadku.

Obiekty kontenerowe

Kontenery to obiekty, które implementują metodę `__contains__` (zwykle zwracającą wartość logiczną). Wspomniana metoda jest wywoływana w obecności słowa kluczowego `in` Pythona. Przykładowo:

```
element in kontener
```

w rzeczywistym kodzie Pythona przyjmuje następującą postać:

```
container.__contains__(element)
```

Wyobraź sobie, o ile bardziej czytelny (i pythoniczny!) może być kod, gdy ta metoda zostanie poprawnie zaimplementowana.

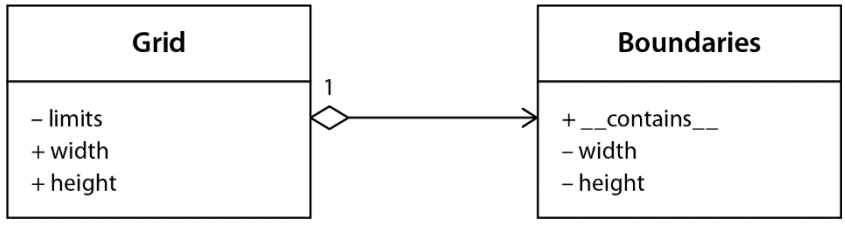
Załóżmy, że musimy zaznaczyć kilka punktów na mapie gry, w której wykorzystywane są dwuwymiarowe współrzędne. Moglibyśmy oczekiwać utworzenia funkcji podobnej do następującej:

```
def mark_coordinate(grid, coord):
    if 0 <= coord.x < grid.width and 0 <= coord.y < grid.height:
        grid[coord] = MARKED
```

Teraz część kodu, która sprawdza stan pierwszej instrukcji `if`, wydaje się zawiła; nie ujawnia intencji kodu, nie jest czytelna, a co najgorsze, stwarza warunki sprzyjające powielaniu kodu (w każdym fragmencie kodu, w którym trzeba będzie najpierw sprawdzić granice, by kontynuować, będzie musiała powtórzyć taką samą instrukcję `if`).

Co by było, gdyby sama mapa (w kodzie zmienna `grid`) mogła odpowiedzieć na to pytanie? Co więcej, co by było, gdyby mapa mogła delegować to działanie do jeszcze mniejszego (a tym samym bardziej spójnego) obiektu?

Moglibyśmy rozwiązać ten problem w bardziej elegancki sposób dzięki projektowi obiektowemu oraz z wykorzystaniem metody magicznej. W tym przypadku możemy utworzyć nową abstrakcję reprezentującą granice siatki, która sama może stać się obiektem. Ideę tę pokazałem na rysunku 2.2.



Rysunek 2.2. Przykład użycia kompozycji w celu podziału obowiązków w różnych klasach i użycia magicznych metod kontenerowych

Wspomnę, iż prawdą jest, że ogólnie rzecz biorąc, nazwy klas odnoszą się do rzeczowników i zwykle są to rzeczowniki w liczbie pojedynczej. Tak więc istnienie klasy o nazwie `Boundaries` może wydawać się dziwne. Jeśli jednak się nad tym zastanowimy, być może w tym konkretnym

przypadku sensowne jest stwierdzenie, że mamy obiekt reprezentujący wszystkie granice siatki, szczególnie ze względu na sposób, w jaki ten obiekt jest używany (w tym przypadku używamy go do zweryfikowania, czy określona współrzędna mieści się w tych granicach).

Dzięki temu projektowi możemy zapytać mapę, czy zawiera współrzędną, a sama mapa może zawierać informacje o swoich ograniczeniach i przekazać zapytanie do wewnętrznego współpracownika:

```
class Boundaries:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def __contains__(self, coord):
        x, y = coord
        return 0 <= x < self.width and 0 <= y < self.height

class Grid:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.limits = Boundaries(width, height)

    def __contains__(self, coord):
        return coord in self.limits
```

Sam ten kod jest znacznie lepszą implementacją. Po pierwsze, implementuje prostą kompozycję i używa delegowania do rozwiązania problemu. Oba obiekty są spójne, zawierają minimalną możliwą logikę; metody są krótkie, a logika mówi sama za siebie — `coord in self.limits` jest w zasadzie deklaracją problemu do rozwiązania, wyrażającą intencję kodu.

Z zewnątrz również możemy zobaczyć korzyści. To prawie tak, jakby Python rozwiązywał problem za nas:

```
def mark_coordinate(grid, coord):
    if coord in grid:
        grid[coord] = MARKED
```

Dynamiczne atrybuty obiektów

Możliwe jest kontrolowanie sposobu, w jaki z obiektów są odczytywane atrybuty z pomocą magicznej metody `__getattr__`. Kiedy użyjemy wywołania postaci *<mójobiekt>.<mójatrybut>*, Python poszuka atrybutu *<mójatrybut>* w słowniku obiektu i wywoła na nim metodę `__getattribute__`. Jeśli atrybut nie zostanie znaleziony (bo obiekt nie ma atrybutu, którego szukamy), wywoływana zostanie dodatkowa metoda `__getattr__`, do której nazwa atrybutu (*mójatrybut*) będzie przekazana jako parametr.

Po otrzymaniu tej wartości możemy decydować o sposobie zwracania danych do obiektów. Możemy nawet tworzyć nowe atrybuty i tak dalej.

Przykład metody `__getattr__` pokazałem na poniższym listingu:

```
class DynamicAttributes:

    def __init__(self, attribute):
        self.attribute = attribute

    def __getattr__(self, attr):
        if attr.startswith("fallback_"):
            name = attr.replace("fallback_", "")
            return f"[awaryjnie] {name}"
        raise AttributeError(
            f"{self.__class__.__name__} nie ma atrybutu {attr}"
        )
```

Oto kilka wywołań do obiektu tej klasy:

```
>>> dyn = DynamicAttributes("wartość")
>>> dyn.attribute
'wartość'

>>> dyn.fallback_test
'[awaryjnie] test'

>>> dyn.__dict__["fallback_new"] = "nowa wartość"
>>> dyn.fallback_new
'nowa wartość'

>>> getattr(dyn, "coś", "domyślnie")
'domyślnie'
```

Pierwsze wywołanie jest proste — po prostu żądamy atrybutu, który obiekt posiada, i w rezultacie otrzymujemy jego wartość. Drugie to demonstracja wywołania metody `__getattr__`. Ponieważ obiekt nie ma atrybutu o nazwie `fallback_test`, więc wywołuje metodę `__getattr__` z tą wartością. Wewnątrz tej metody umieściliśmy kod, który zwraca ciąg znaków, zatem otrzymujemy ten ciąg.

Trzeci przykład jest interesujący, ponieważ tworzymy nowy atrybut o nazwie `fallback_new` (w rzeczywistości to wywołanie jest równoważne z wywołaniem `dyn. fallback_new = "nowa wartość"`), zauważmy więc, że kiedy zażądamy tego atrybutu, nie zadziała logika umieszczona w metodzie `__getattr__`. To dlatego, że kod tej metody nigdy nie zostanie wywołany.

Ostatni przykład jest najciekawszy. Jest w nim subtelny szczegół, który sprawia ogromną różnicę. Spójrz jeszcze raz na kod w metodzie `__getattr__`. Zwróć uwagę na wyjątek `AttributeError`, który jest zgłaszany w przypadku, gdy nie można odczytać wartości. Zgłoszenie tego wyjątku ma na celu nie tylko zapewnienie spójności (jak również wyświetlenie komunikatu w wyjątku), ale jest również wymagane przez wbudowaną funkcję `getattr()`. Gdyby metoda zgłosiła jakikolwiek inny wyjątek, zostałby on przekazany w górę stosu, a wartość domyślna nie zostałaby zwrócona.

> Podczas implementacji metody tak dynamicznej jak `__getattr__` należy zachować ostrożność. Pamiętaj, że metoda `__getattr__` powinna zgłaszać wyjątek `AttributeError`.

Magiczna metoda `__getattr__` jest przydatna w wielu sytuacjach. Może być używana do tworzenia pośredników do innych obiektów. Jeśli na przykład tworzysz obiekt będący opakowaniem innego obiektu, korzystając z kompozycji, i chcesz oddelegować większość metod do obiektu opakowanego, zamiast kopiować i definiować wszystkie te metody, możesz zaimplementować metodę `__getattr__`, która wewnętrznie wywoła tę samą metodę na obiekcie opakowanym.

Inny przykład to sytuacja, gdy wiesz, że potrzebujesz atrybutów, które będą obliczane dynamicznie. Sytuację tę wykorzystałem w jednym z moich projektów do współpracy biblioteki GraphQL (*https://graphql.org/*) z Graphene (*https://graphene-python.org/*). Działanie biblioteki opierało się na wykorzystaniu metod resolwera. Ogólnie rzecz biorąc, żądanie właściwości `X` powodowało wywoływanie metod o nazwie `resolve_X`. Ponieważ istniały obiekty dziedzinowe zdolne do rozpoznania właściwości `X` w klasie obiektu `Graphene`, zaimplementowano metodę `__getattr__`, która wiedziała, skąd należy pobrać każdą właściwość, bez konieczności pisania rozbudowanego kodu narzędziowego.

Z magicznej metody `__getattr__` warto skorzystać w sytuacji, gdy widzisz okazję do uniknięcia powielonego i narzędziowego kodu, ale nie należy jej nadużywać, ponieważ może to utrudnić zrozumienie kodu i jego czytanie. Warto pamiętać, że posługiwanie się atrybutami, które nie są jawnie zadeklarowane, ale są obliczane dynamicznie, utrudnia zrozumienie kodu. Gdy korzystasz z metody `__getattr__`, zawsze powinieneś brać pod uwagę kompromis pomiędzy zwięzłością kodu a łatwością jego utrzymania.

Obiekty wywoływalne

Istnieje możliwość (często jest to bardzo wygodne) definiowania obiektów, które mogą działać jako funkcje. Jednym z najczęstszych zastosowań tego mechanizmu jest tworzenie lepszych dekoratorów, ale nie jest ono jedyne.

Gdy spróbujemy uruchomić obiekt tak, jakby był zwykłą funkcją, zostanie wywołana magiczna metoda `__call__`. Każdy argument przekazany do obiektu będzie przekazany do metody `__call__`.

Główną zaletą implementacji funkcji poprzez obiekty jest to, że obiekty mają stany, dzięki którym możemy zapisywać i utrzymywać informacje pomiędzy różnymi wywołaniami. Oznacza to, że jeśli trzeba utrzymywać wewnętrzny stan pomiędzy różnymi wywołaniami, użycie obiektu wywoływalnego może być wygodniejszym sposobem implementacji funkcji. Przykładami mogą być funkcje wykorzystywane do zapamiętywania wartości lub wewnętrzne pamięci podręczne.

Jeśli zdefiniujemy obiekt `obiekt`, to Python przetłumaczy instrukcję w postaci `obiekt(*args, **kwargs)` na `obiekt.__call__(*args, **kwargs)`.

Ta metoda jest przydatna, gdy chcemy utworzyć obiekty wywoływalne, które będą działać jako sparametryzowane funkcje lub w niektórych przypadkach do wywoływania funkcji z pamięcią.

W kodzie z poniższego listingu użyłem tej metody do skonstruowania obiektu, który po wywołaniu z parametrem zwraca liczbę wywołań obiektu z tą samą wartością:

```
from collections import defaultdict

class CallCount:

    def __init__(self):
        self._counts = defaultdict(int)

    def __call__(self, argument):
        self._counts[argument] += 1
        return self._counts[argument]
```

Oto kilka przykładów wykorzystania tej klasy:

```
>>> cc = CallCount()
>>> cc(1)
1
>>> cc(2)
1
>>> cc(1)
2
>>> cc(1)
3
>>> cc("coś")
1
>>> callable(cc)
True
```

Dalej w książce pokażę zastosowanie metody `__call__` do tworzenia dekoratorów.

Podsumowanie metod magicznych

Pojęcia, które opisałem w poprzednich punktach, można podsumować w formie ściągawki, takiej jak przedstawiona poniżej. Dla każdego działania w Pythonie przedstawiono magiczną metodę wraz z mechanizmem, który reprezentuje.

Najlepszym sposobem na poprawną implementację tych metod (i poznanie zestawu metod, które muszą być zaimplementowane razem) jest zadeklarowanie własnej klasy zgodnie z interfejsem abstrakcyjnych klas bazowych zdefiniowanych w module `collections.abc` (*https://docs.python.org/3/library/collections.abc.html#collections-abstract-base-classes*). Interfejsy tych klas udostępniają metody, które trzeba zaimplementować. Zapoznanie się z nimi ułatwi poprawne zdefiniowanie klasy, a także obsłuży prawidłowe tworzenie typu (takiego, który dobrze działa, gdy zostanie wywołana na nim funkcja `isinstance()`).

Tabela 2.1. Metody magiczne i ich zachowanie w Pythonie

Instrukcja	Metoda magiczna	Zachowanie

`obj[key]`
`obj[i:j]`
`obj[i:j:k]`
	`__getitem__(key)`	Obiekt indeksowalny

`with obj: ...`	`__enter__/__exit__`	Menedżer kontekstu
`for i in obj:`	`__iter__/__next__`	Obiekt iterowalny
`...`	`__len__/__getitem__`	Sekwencja
`obj.<atrybut>`	`__getattr__`	Dynamiczne pobieranie atrybutów
`obj(*args, **kwargs)`	`__call__(*args, **kwargs)`	Obiekt wywoływalny

Pokazałem najważniejsze cechy Pythona w odniesieniu do jego osobliwej składni. Po zapoznaniu się z nimi (menedżery kontekstu, obiekty wywoływalne, tworzenie własnych sekwencji i tym podobne) możesz pisać kod, który będzie się dobrze komponować z zastrzeżonymi słowami Pythona (na przykład możesz używać instrukcji `with` z własnymi menedżerami kontekstu lub operatora `in` z własnym kontenerem).

Praktyka i doświadczenie sprawią, że będziesz stawać się coraz bardziej biegły w posługiwaniu się tymi cechami Pythona, aż opakowywanie logiki, którą zapisujesz w abstrakcje z ładnymi i zwięzłymi interfejsami stanie się Twoją drugą naturą. Daj sobie wystarczająco dużo czasu, a nastąpi odwrotny efekt: Python zacznie programować Ciebie. Oznacza to, że w naturalny sposób zaczniesz myśleć o zwięzłych, czystych interfejsach w swoich programach, dzięki czemu nawet wtedy, gdy będziesz tworzyć oprogramowanie w innych językach, spróbujesz używać tych pojęć. Jeśli na przykład programujesz w Javie lub C (albo nawet w Bashu), będziesz potrafił zidentyfikować scenariusz, w którym może być przydatny menedżer kontekstu. Być może sam język nie obsługuje tego pojęcia, ale to nie powstrzyma Cię przed napisaniem własnej abstrakcji, która zapewni podobne mechanizmy. To dobrze. Oznacza to, że zinternalizowałeś dobre wzorce wykraczające poza konkretny język i możesz je zastosować w różnych sytuacjach.

We wszystkich językach programowania istnieją pewne haczyki i Python nie jest wyjątkiem. Zatem, aby lepiej zrozumieć Pythona, przyjrzymy się kilku z nich w następnym podrozdziale.

Haczyki Pythona

Umiejętność pisania idiomatycznego kodu, oprócz zrozumienia głównych cech języka, polega również na uświadomieniu sobie potencjalnych problemów związanych z niektórymi idiomami i poznaniu sposobów ich unikania. W tym podrozdziale omówię typowe problemy, które — gdybyś nie był na nie przygotowany — mogłyby być przyczyną długotrwałych sesji.

Większość punktów tego podrozdziału dotyczy problemów, których należy unikać. Ośmielę się powiedzieć, że nie istnieje żaden możliwy scenariusz, który uzasadniałby stosowanie antywzorca (lub w tym przypadku idiomu). Dlatego, jeśli znajdziesz taki antywzorzec w bazie kodu, nad którą pracujesz, zrefaktoryzuj kod w sugerowany sposób. Jeśli znajdziesz podobne antywzorce podczas przeglądu kodu, będzie to wyraźna wskazówka, że coś trzeba zmienić.

Mutowalne argumenty domyślne

Mówiąc najprościej, nie używaj mutowalnych obiektów jako domyślnych argumentów funkcji. Jeśli użyjesz takich obiektów jako argumentów domyślnych, otrzymasz nieoczekiwane wyniki.

Rozważmy następującą błędną definicję funkcji:

```
def wrong_user_display(user_metadata: dict = {"name": "Jan", "age": 30}):
    name = user_metadata.pop("name")
    age = user_metadata.pop("age")

    return f"{name} ({age})"
```

W pokazanym kodzie w istocie są dwa problemy. Oprócz tego, że domyślny argument jest mutowalny, w treści funkcji następuje jego modyfikacja, co tym samym tworzy efekt uboczny. Jednak głównym problemem jest domyślny argument `user_metadata`.

Argument domyślny w rzeczywistości zadziała tylko przy pierwszym wywołaniu metody bez argumentów. Gdy po raz drugi wywołamy ją bez wyraźnego przekazania czegoś w argumencie `user_metadata`, nastąpi zgłoszenie wyjątku `KeyError`, co pokazałem poniżej:

```
>>> wrong_user_display()
'Jan (30)'
>>> wrong_user_display({"name": "Janina", "age": 25})
'Janina (25)'
>>> wrong_user_display()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ... in wrong_user_display
    name = user_metadata.pop("name")
KeyError: 'name'
```

Wyjaśnienie jest proste — poprzez przypisanie w definicji funkcji słownika z domyślnymi danymi do argumentu `user_metadata` słownik ten jest faktycznie tworzony raz, a zmienna `user_metadata` na niego wskazuje. Gdy interpreter języka Python podejmie próbę parsowania pliku, odczyta funkcję i znajdzie w sygnaturze instrukcję tworzącą słownik i przypisującą go do parametru. Od tego momentu słownik, który został raz utworzony, jest taki sam przez cały okres istnienia programu.

Następnie, w treści funkcji ten obiekt, który pozostaje w pamięci tak długo, jak działa program, zostaje zmodyfikowany. Gdy przekażemy do niego wartość, zajmie ona miejsce domyślnego argumentu, który właśnie utworzyliśmy. Chociaż nie chcemy tego obiektu, jest on wywoływany

ponownie, a został zmodyfikowany od poprzedniego uruchomienia. Następnym razem, gdy go uruchomimy, nie będzie zawierał kluczy, ponieważ zostały usunięte podczas poprzedniego wywołania.

Poprawka jest również prosta — musimy użyć `None` jako domyślnej wartości-strażnika i przypisać wartość domyślną w treści funkcji. Ponieważ każda funkcja ma swój własny zakres i cykl życia, do argumentu `user_metadata` zostanie przypisany słownik za każdym razem, gdy będzie przekazana wartość `None`:

```
def user_display(user_metadata: dict = None):
    user_metadata = user_metadata or {"name": "John", "age": 30}

    name = user_metadata.pop("name")
    age = user_metadata.pop("age")

    return f"{name} ({age})"
```

Podrozdział zakończę omówieniem dziwactw związanych z rozszerzaniem typów wbudowanych.

Rozszerzanie typów wbudowanych

Prawidłowym sposobem rozszerzania typów wbudowanych, takich jak listy, ciągi i słowniki, jest wykorzystanie modułu `collections`.

Jeśli na przykład utworzysz klasę, która bezpośrednio rozszerza typ `dict`, uzyskasz wyniki, które prawdopodobnie nie są tym, czego oczekujesz. Powodem jest to, że w implementacji CPython (napisanej w języku C) metody klasy nie wywołują się nawzajem (tak jak powinny), więc jeśli zastąpisz jedną z nich, nie zostanie to odzwierciedlone przez resztę. To spowoduje nieoczekiwane wyniki. Przykładowo chcesz zastąpić `__getitem__`, a następnie podczas iterowania po obiekcie w pętli `for` zauważasz, że logika wprowadzona w tej metodzie nie jest stosowana.

Wszystko to można rozwiązać, na przykład poprzez zastosowanie klasy `collections.UserDict`, która dostarcza przezroczysty interfejs do właściwych słowników i jest bardziej elastyczna.

Załóżmy, że chcesz, aby elementy listy, która pierwotnie została utworzona z liczb, zostały skonwertowane na ciągi znaków z dodaniem prefiksu. Może się wydawać, że poniższe pierwsze podejście rozwiązuje problem, ale może ono prowadzić do błędów:

```
class BadList(list):
    def __getitem__(self, index):
        value = super().__getitem__(index)
        if index % 2 == 0:
            prefix = "parzysta"
        else:
            prefix = "nieparzysta"
        return f"[{prefix}] {value}"
```

Na pierwszy rzut oka może się wydawać, że obiekt zachowuje się tak, jak powinien. Jeśli jednak spróbujesz po nim iterować (w końcu to jest lista), przekonasz się, że nie otrzymujesz tego, czego chcesz:

```
>>> bl = BadList((0, 1, 2, 3, 4, 5))
>>> bl[0]
'[parzysta] 0'
>>> bl[1]
'[nieparzysta] 1'
>>> "".join(bl)
Traceback (most recent call last):
...
TypeError: sequence item 0: expected str instance, int found
```

Funkcja `join` próbuje iterować (uruchomić pętlę `for`) po liście, ale oczekuje wartości typu `string`. Można by się spodziewać, że iterowanie powiedzie się, ponieważ zmodyfikowaliśmy metodę `__getitem__` tak, aby zawsze zwracała ciąg. Na podstawie wyniku możemy jednak stwierdzić, że zmodyfikowana wersja metody `__getitem__` nie jest wywoływana. Problem ten wynika w istocie ze szczegółów implementacji interpretera CPython, podczas gdy na innych platformach, takich jak PyPy, tak się nie dzieje (możesz o tym przeczytać w artykule „Differences between PyPy and CPython" — patrz materiały referencyjne na końcu tego rozdziału).

Niezależnie od tego powinniśmy napisać kod, który jest przenośny i kompatybilny ze wszystkimi implementacjami, więc naprawimy go, rozszerzając nie typ `list`, ale `UserList`:

```
from collections import UserList

class GoodList(UserList):
    def __getitem__(self, index):
        value = super().__getitem__(index)
        if index % 2 == 0:
            prefix = "parzysta"
        else:
            prefix = "nieparzysta"
        return f"[{prefix}] {value}"
```

Teraz wszystko wygląda znacznie lepiej:

```
>>> gl = GoodList((0, 1, 2))
>>> gl[0]
'[parzysta] 0'
>>> gl[1]
'[nieparzysta] 1'
>>> "; ".join(gl)
'[parzysta] 0; [nieparzysta] 1; [parzysta] 2'
```

Nie należy rozszerzać bezpośrednio typu `dict`. Zamiast niego skorzystaj z typu `collections.UserDict`. W przypadku list użyj klasy `collections.UserList`, a dla ciągów znaków użyj klasy `collections.UserString`.

W tym momencie znasz wszystkie główne pojęcia Pythona. Nie tylko wiesz, jak pisać idiomatyczny kod, który dobrze komponuje się z samym Pythonem, ale także jak unikać pewnych pułapek. Następny podrozdział zawiera uzupełnienie tej wiedzy.

Na końcu rozdziału postanowiłem umieścić szybkie wprowadzenie do programowania asynchronicznego. Chociaż nie jest ono ściśle związane z czystym kodem *per se*, kod asynchroniczny staje się coraz bardziej popularny. Podążam więc za ideą, że aby skutecznie pracować z kodem, trzeba umieć go odczytać i zrozumieć. Dotyczy to również kodu asynchronicznego.

Krótkie wprowadzenie do kodu asynchronicznego

Programowanie asynchroniczne nie jest związane z czystym kodem. Dlatego własności Pythona opisane w tym podrozdziale nie ułatwią utrzymania bazy kodu. Ten podrozdział zawiera wprowadzenie do składni Pythona niezbędnej do pracy z podprogramami. Podprogramy mogą Ci się przydać, a przykłady korzystania z podprogramów mogą pojawić się dalej w tej książce.

Ideą programowania asynchronicznego jest zawieszenie działania pewnych fragmentów kodu, aby mogły działać inne fragmenty. Zazwyczaj, gdy uruchamiamy operacje wejścia-wyjścia, bardzo chcielibyśmy, aby ten kod działał, a w tym czasie chcemy używać procesora do czegoś innego.

Takie podejście zmienia model programowania. Zamiast stosowania wywołań synchronicznych chcemy napisać kod w sposób, który jest nazywany pętlą zdarzeń odpowiedzialną za szeregowanie podprogramów tak, aby wszystkie działały w tym samym procesie i wątku.

Chodzi o to, że tworzymy zbiór podprogramów, które są dodawane do pętli zdarzeń. Po uruchomieniu pętla zdarzeń wybiera spośród posiadanych podprogramów i planuje ich uruchomienie. W pewnym momencie, gdy jeden z podprogramów musi wykonać operację wejścia-wyjścia, może ją uruchomić i przekazać sygnał do pętli zdarzeń, aby ponownie przejęła kontrolę. Pętla zdarzeń może wtedy zaplanować działanie kolejnego podprogramu w czasie, gdy inny podprogram wykonuje operację wejścia-wyjścia. W pewnym momencie pętla zdarzeń wznawia podprogram w ostatnim punkcie, w którym się zatrzymała, i kontynuuje od tego momentu. Należy zapamiętać, że zaletą programowania asynchronicznego są nieblokujące operacje wejścia-wyjścia. Oznacza to, że kod może przejść do wykonywania innych działań w czasie, gdy inny podprogram realizuje operacje wejścia-wyjścia, a następnie wrócić do tego podprogramu, ale to nie oznacza, że istnieje wiele procesów uruchomionych jednocześnie. Model działania nadal jest jednowątkowy.

Aby osiągnąć taki model w Pythonie, można skorzystać z licznych frameworków. Jednak w starszych wersjach Pythona nie było konkretnej składni, która na to pozwalała, więc sposób działania frameworków był nieco skomplikowany lub nieoczywisty na pierwszy rzut oka. Począwszy od Pythona 3.5, do języka dodano specyficzną składnię do deklarowania podprogramów, co zmieniło sposób, w jaki możemy pisać asynchroniczny kod w tym języku. Nieco wcześniej w standardowej bibliotece wprowadzono domyślny moduł obsługi pętli zdarzeń `asyncio`.

Dzięki tym dwóm kamieniom milowym programowanie asynchroniczne w Pythonie stało się znacznie łatwiejsze.

Chociaż w tym podrozdziale używamy `asyncio` jako modułu do przetwarzania asynchronicznego, nie jest to jedyny moduł dostępny do tego celu. Kod asynchroniczny możesz pisać za pomocą dowolnej biblioteki (dostępnych jest wiele modułów poza biblioteką standardową, na przykład `trio` — *https://github.com/python-trio/trio* i `curio` — *https://github.com/dabeaz/curio*). Składnia, którą Python dostarcza w celu pisania podprogramów, może być uważana za interfejs API. Tak długo, jak wybrana biblioteka jest zgodna z tym interfejsem API, powinieneś z niej korzystać, bez konieczności zmiany sposobu deklarowania podprogramów.

Różnice składniowe w porównaniu z programowaniem asynchronicznym polegają na tym, że podprogramy są podobne do funkcji, ale definiowane za pomocą instrukcji `async def`. Kiedy jesteś wewnątrz podprogramu i chcesz wywołać inny (może to być podprogram, który sami zdefiniowaliśmy lub zdefiniowany w bibliotece zewnętrznej), zwykle możesz użyć słowa kluczowego `await` przed jego nazwą. Wywołanie `await` jest sygnałem dla pętli zdarzeń, aby przejęła kontrolę. W tym momencie pętla zdarzeń wznawia swoje działanie, a podprogram pozostaje w tym samym miejscu w oczekiwaniu na wznowienie jego nieblokującej operacji. W międzyczasie uruchamia się inna część kodu (pętla zdarzeń wywoła inny podprogram). W pewnym momencie pętla zdarzeń ponownie wywoła wcześniej zatrzymany podprogram i zostanie on wznowiony od punktu, w którym został wstrzymany (bezpośrednio za wierszem z instrukcją `await`).

Typowy podprogram, który możesz zdefiniować we własnym kodzie, ma następującą strukturę:

```
async def mojpodprogram(*args, **kwargs):
    # … logika
await zewnetrzna_biblioteka.podprogram(…)
    # … dalsza część logiki
```

Jak wspomniałem wcześniej, w nowych wersjach Pythona pojawiła się nowa składnia do definiowania podprogramów. Jedną z różnic, które wprowadza ta składnia, jest to, że w przeciwieństwie do zwykłych funkcji wywołanie funkcji zdefiniowanych w ten sposób nie uruchamia jej kodu. Zamiast tego tworzy obiekt podprogramu. Obiekt ten jest umieszczany w pętli zdarzeń i w pewnym momencie musi zostać wywołany z użyciem instrukcji `await` (w przeciwnym razie kod wewnątrz definicji nigdy nie zostanie uruchomiony):

```
result = await mojpodprogram(…) # wywołanie result = mojpodprogram() spowodowałoby błąd
```

> Nie zapomnij wywołać swoich podprogramów za pomocą słowa kluczowego `await`, ponieważ inaczej ich kod nigdy nie zostanie uruchomiony. Zwróć uwagę na ostrzeżenia generowane przez moduł `asyncio`.

Jak wspomniałem, w Pythonie istnieje kilka bibliotek do programowania asynchronicznego. Zawierają one pętle zdarzeń zdolne do uruchamiania podprogramów podobnych do zdefiniowanych wcześniej. W przypadku modułu `asyncio` istnieje wbudowana funkcja do uruchamiania podprogramu aż do jego zakończenia:

```
import asyncio
asyncio.run(mojpodprogram(…))
```

Szczegóły dotyczące działania podprogramów w Pythonie wykraczają poza zakres tej książki, ale to wprowadzenie powinno pomóc w przyswojeniu sobie niezbędnej składni. Warto pamiętać, że podprogramy, z technicznego punktu widzenia, są zaimplementowane na bazie generatorów, które szczegółowo omówię w rozdziale 7., „Generatory, iteratory i programowanie asynchroniczne".

Podsumowanie

W tym rozdziale omówiłem własności Pythona niezbędne do zrozumienia jego najbardziej charakterystycznych cech. Dzięki nim Python jest specyficznym językiem w porównaniu z innymi językami programowania. W tym celu omówiłem różne dostępne w Pythonie metody, protokoły oraz ich wewnętrzną mechanikę.

W przeciwieństwie do poprzedniego rozdziału, ten jest bardziej skoncentrowany na Pythonie. Kluczowym wnioskiem z tematów omawianych w tej książce jest to, że czysty kod to nie tylko przestrzeganie zasad formatowania (co oczywiście jest niezbędne dla zapewnienia dobrej jakości bazy kodu). Dobre formatowanie to warunek konieczny, ale niewystarczający. W najbliższych kilku rozdziałach zaprezentuję pomysły i reguły odnoszące się do kodu, mające na celu osiągnięcie lepszego projektu i implementacji programowego rozwiązania.

Dzięki pojęciom i tematom omówionym w tym rozdziale opisałem zasadniczą część Pythona: jego protokoły i metody magiczne. Powinno być już jasne, że najlepszym sposobem na tworzenie pythonicznego, idiomatycznego kodu jest nie tylko przestrzeganie konwencji formatowania, ale także pełne wykorzystanie wszystkich własności, jakie Python ma do zaoferowania. Oznacza to, że używanie określonej metody magicznej, menedżera kontekstu lub pisanie zwięźlejszych instrukcji z użyciem wyrażeń składanych i wyrażeń przypisania spowoduje, że napiszesz kod, który będzie łatwiejszy w utrzymaniu.

W tym rozdziale zapoznałeś się również z programowaniem asynchronicznym. Powinieneś teraz bez trudu czytać asynchroniczny kod w Pythonie. To ważne, ponieważ programowanie asynchroniczne staje się coraz bardziej popularne i będzie przydatne podczas analizy tematów, omówionych dalej w tej książce.

W następnym rozdziale pokażę wykorzystanie omówionych pojęć w praktyce. Zestawię ogólne koncepcje inżynierii oprogramowania ze sposobem, w jaki można je wykorzystywać w Pythonie.

Materiały referencyjne

Więcej informacji na temat niektórych tematów, które omówiłem w tym rozdziale, można znaleźć w poniższych materiałach. Sposób działania indeksów w Pythonie bazuje na (EWD831), gdzie przeanalizowano kilka możliwości dla zakresów w matematyce i językach programowania:

- EWD831: *Why numbering should start at zero* (*https://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html*).

- PEP-343: *The „with" statement* (*https://www.python.org/dev/peps/pep-0343/*).
- CC08: Robert C. Martin, „Clean Code: A Handbook of Agile Software Craftsmanship"[1].
- *The iter() function*: *https://docs.python.org/3/library/functions.html#iter*.
- *Differences between PyPy and CPython*: *https://pypy.readthedocs.io/en/latest/cpython_differences.html#subclasses-of-built-in-types*
- *The Art of Enbugging*: *http://media.pragprog.com/articles/jan_03_enbug.pdf*.
- ALGO01: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest i Clifford Stein, „Introduction to Algorithms", wydanie 3. (The MIT Press)[2].

[1] Wydanie polskie: „Czysty kod. Podręcznik dobrego programisty", Helion 2014 — przyp. tłum.

[2] Wydanie polskie: „Wprowadzenie do algorytmów", Wydawnictwo Naukowe PWN 2012 — przyp. tłum.

Czysty kod w Pythonie. Tylko taki warto pisać!

Popularność Pythona, ulubionego języka programistów i naukowców, stale rośnie. Jest on bowiem łatwy do nauczenia się: nawet początkujący programista może napisać działający kod. W efekcie, mimo że Python pozwala na pisanie kodu przejrzystego i prostego w konserwacji, zdarzają się przypadki kodu źle zorganizowanego, nieczytelnego i praktycznie nietestowalnego. Jedną z przyczyn tego stanu rzeczy jest tendencja niektórych programistów do pisania kodu bez czytelnej struktury. Zidentyfikowanie takich problemów i ich rozwiązywanie nie jest łatwym zadaniem.

Dzięki tej książce nauczysz się korzystać z kilku narzędzi służących do zarządzania projektami napisanymi w Pythonie. Dowiesz się, czym się charakteryzuje czysty kod i jakie techniki umożliwiają tworzenie czytelnego i wydajnego kodu. Przekonasz się, że do tego celu wystarczą standardowa biblioteka Pythona i zestaw najlepszych praktyk programistycznych. Opisano tu szczegóły programowania obiektowego w Pythonie wraz z zastosowaniem deskryptorów i generatorów. Zaprezentowano również zasady testowania oprogramowania i sposoby rozwiązywania problemów poprzez implementację wzorców projektowych w kodzie. Pokazano też, jak można podzielić monolityczną aplikację na mikrousługi, by otrzymać solidną architekturę aplikacji.

W książce między innymi:

- konfiguracja wydajnego środowiska programistycznego
- tworzenie zaawansowanych projektów obiektowych
- techniki eliminacji zdublowanego kodu i tworzenie rozbudowanych abstrakcji
- zastosowanie dekoratorów i deskryptorów
- skuteczna refaktoryzacja kodu
- budowa solidnej architektury opartej na czystym kodzie Pythona

Mariano Anaya

jest doświadczonym inżynierem oprogramowania. Tworzy oprogramowanie i wspiera innych programistów. Zajmuje się architekturą oprogramowania, programowaniem funkcyjnym i systemami rozproszonymi. Był prelegentem na konferencjach EuroPython w latach 2016 i 2017, a także FOSDEM w 2019 roku.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-8611-2
9 788328 386112
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 79,00 zł
Packt