Wydanie IV

Programowanie zorientowane obiektowo w Pythonie

Tworzenie solidnych i łatwych w utrzymaniu aplikacji i bibliotek

Steven F. Lott
Dusty Phillips

Helion
Packt

Tytuł oryginału: Python Object-Oriented Programming: Build robust and maintainable object-oriented Python applications and libraries, 4th Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-8949-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/przop4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/przop4.zip

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

Printed in Poland.

Spis treści

ROZDZIAŁ 2

Obiekty w Pythonie

Projekt leży już przed nami na biurku, a my jesteśmy gotowi, by przekształcić go w działający program! Oczywiście to się nie dzieje w taki sposób. W tej książce przedstawiamy przykłady i wskazówki dotyczące dobrych projektów oprogramowania, jednak koncentrujemy się na zagadnieniach programowania obiektowego. Dlatego też przyjrzymy się składni Pythona używanej do pisania programów obiektowych.

Po przeczytaniu tego rozdziału zrozumiesz:

- czym są podpowiedzi typów;
- jak tworzyć klasy i obiekty w Pythonie;
- jak organizować klasy w pakiety i moduły;
- jak sugerować, by inni programiści nie modyfikowali danych obiektu, ponieważ doprowadza to do błędów w jego stanie;
- jak używać pakietów innych twórców dostępnych w serwisie Python Package Index (PyPI).

W tym rozdziale będziemy także kontynuować prace w ramach naszego studium przypadku, a konkretnie zajmiemy się zaprojektowaniem wybranych klas.

Prezentacja podpowiedzi typów

Zanim będziemy mogli dokładniej przyjrzeć się tworzeniu klas, musimy zrozumieć, czym są klasy i jak możemy zyskać pewność, że będziemy ich prawidłowo używać. Kluczową rzeczą jest tutaj to, że w Pythonie wszystko jest obiektem.

Kiedy zapisujemy w kodzie literały, takie jak `"Witaj, świecie!"` czy `42`, to w rzeczywistości tworzymy instancje wbudowanych klas Pythona. Możemy uruchomić interaktywny interpreter Pythona i użyć wbudowanej funkcji `type()`, by poznać klasę definiującą właściwości tych obiektów:

```
>>> type("Witaj, świecie!")
<class 'str'>
>>> type(42)
<class 'int'>
```

Podstawowym celem *programowania obiektowego* jest rozwiązywanie problemów w wyniku interakcji pomiędzy obiektami. Kiedy zapisujemy wyrażenie `6*7`, mnożenie tych dwóch obiektów jest obsługiwane przez metodę wbudowanej klasy `int`. Aby obsłużyć bardziej złożone zachowania, niejednokrotnie będziemy musieli pisać nowe, unikalne klasy.

Pierwsze dwie podstawowe reguły dotyczące sposobu działania obiektów w Pythonie są następujące:

- W Pythonie wszystko jest obiektem.
- Każdy obiekt jest instancją przynajmniej jednej klasy, która definiuje jego cechy.

Te dwie reguły mają wiele interesujących konsekwencji. Definicja typu, którą zapisujemy w kodzie przy użyciu instrukcji `class`, tworzy nowy obiekt klasy `type`. Kiedy tworzymy **instancję** pewnej klasy, do jej utworzenia i zainicjowania zostaje użyty obiekt tej klasy.

Jaka zatem jest różnica pomiędzy klasą i typem? Instrukcja `class` pozwala nam definiować nowe typy. Ponieważ używamy właśnie instrukcji `class`, w tekście będziemy te typy nazywać klasami. W artykule Eliego Benderskiego *Python objects, types, classes, and instances — a glossary*[1], dostępnym na stronie *https://eli.thegreenplace.net/2012/03/30/python-objects-types-classes-and-instances-a-glossary*, można znaleźć bardzo pomocny cytat:

> *„Terminy »klasa« oraz »typ« są przykładami dwóch nazw odnoszących się do tego samego pojęcia”.*

Zgodnie z powszechnie stosowaną konwencją adnotacje będziemy nazywali **podpowiedziami typów** (ang. *type hints*).

Jest jeszcze jedna ważna reguła:

- Zmienna jest referencją do obiektu. Pomyśl o żółtej, samoprzylepnej karteczce z napisaną na niej nazwą, przylepionej do jakiegoś przedmiotu.

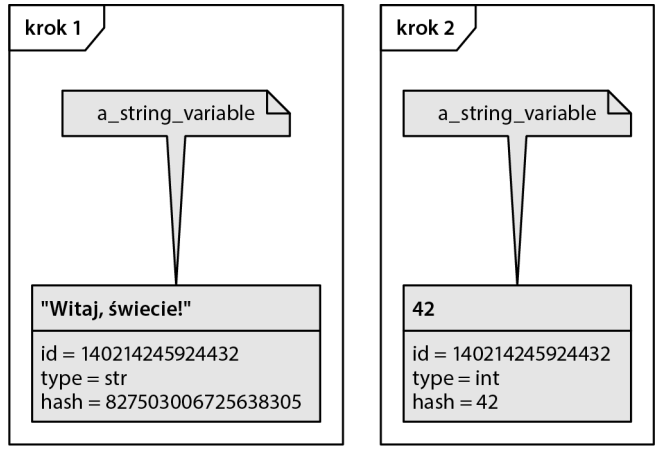
Ta reguła nie jest czymś wstrząsającym, ale na pewno jest dobra. Oznacza, że informacje o typie — czyli to, czym obiekt jest — są definiowane przez klasę (lub klasy) skojarzone z tym obiektem. Te informacje o typie nie są w żaden sposób powiązane ze *zmienną*. Z tego względu poniższy przykład jest całkowicie poprawnym kodem napisanym w Pythonie, lecz jednocześnie jest bardzo mylący:

```
>>> a_string_variable = "Witaj, świecie!"
>>> type(a_string_variable)
<class 'str'>
>>> a_string_variable = 42
>>> type(a_string_variable)
<class 'int'>
```

[1] *Obiekty, typy, klasy i instancje w Pythonie — słowniczek — przyp. tłum.*

W tym przykładzie utworzyliśmy obiekt jednej z wbudowanych klas Pytona — `str`. Z tym obiektem skojarzyliśmy długą nazwę, `a_string_variable`. Następnie utworzyliśmy obiekt innej wbudowanej klasy Pythona — `int`. Po czym skojarzyliśmy go z tą samą nazwą. (Utworzony wcześniej obiekt łańcucha znaków nie ma już żadnej referencji, która by się do niego odwoływała, więc zostaje usunięty).

Na rysunku 2.1 pokazaliśmy obok siebie dwa kroki przedstawiające, w jaki sposób zmienna jest przenoszona z jednego obiektu do drugiego.



Rysunek 2.1. Nazwy zmiennych i obiekty

Różne właściwości są elementami obiektu, a nie zmiennej. Kiedy przy użyciu funkcji `type()` sprawdzamy typ zmiennej, wyświetlany jest typ obiektu, do którego aktualnie ta zmienna się odwołuje. Sama zmienna nie ma żadnego własnego typu — jest ona jedynie nazwą. Analogicznie: wywołanie funkcji `id()` w celu uzyskania identyfikatora zmiennej zwróci identyfikator obiektu, do którego ta zmienna się odwołuje. Dlatego nazwa `a_string_variable` może być bardzo myląca, jeśli przypiszemy tej zmiennej obiekt będący liczbą całkowitą.

Sprawdzanie typów

Przyjrzyjmy się jeszcze bliżej związkowi pomiędzy obiektem i typem, a jednocześnie poznajmy kilka dodatkowych konsekwencji podanych wcześniej reguł. Poniżej przedstawiliśmy definicję funkcji:

```
>>> def odd(n):
...     return n % 2 != 0
...
>>> odd(3)
True
>>> odd(4)
False
```

Ta funkcja przeprowadza proste obliczenia na zmiennej parametru — `n`. Konkretnie rzecz biorąc, oblicza ona resztę z dzielenia, czyli modulo. Jeśli liczbę nieparzystą podzielimy przez dwa, to reszta z dzielenia wyniesie 1. Jeżeli natomiast przez dwa podzielimy liczbę parzystą, to resztą z dzielenia będzie 0. Ta funkcja zwraca logiczną prawdę (`True`), jeśli zostanie do niej przekazana jakakolwiek liczba nieparzysta.

A co się stanie, gdy do tej funkcji przekażemy coś, co nie będzie liczbą? No cóż, po prostu zróbmy tak i się przekonajmy (to powszechnie stosowany sposób uczenia się Pythona!). Jeśli spróbujemy to zrobić w interaktywnym interpreterze Pythona, uzyskamy wyniki podobne do tych przedstawionych w następnym przykładzie:

```
>>> odd("Witaj, świecie!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in odd
TypeError: not all arguments converted during string formatting
```

To ważna konsekwencja superelastycznych reguł Pythona: nic nie broni nam robić głupich rzeczy, które doprowadzą do zgłoszenia wyjątku. Pamiętaj o poniższej, ważnej wskazówce:

> **Wskazówka**
>
> Python nie zabrania nam używać nieistniejących metod obiektów.

W naszym przykładzie operator `%`, którego dostarcza klasa `str`, nie działa tak samo jak operator `%` dostarczany przez klasę `int` i to powoduje zgłoszenie wyjątku. W przypadku łańcuchów znaków operator `%` nie jest zbyt często używany; służy on do wykonywania interpolacji, czyli podstawienia, np. `"a=%d" % 113` zwróci łańcuch o postaci `"a=113"`. Jeśli po lewej stronie operatora nie pojawi się specyfikacja formatu o postaci podobnej do `%d`, zostanie zgłoszony wyjątek `TypeError`. Z kolei w przypadku liczb całkowitych operator ten zwraca resztę z dzielenia, np. `355 % 113` zwraca liczbę całkowitą, a konkretnie `16`.

W tej elastyczności widać wyraźny kompromis, który faworyzuje łatwość użycia, a nie zapobieganie potencjalnym problemom. Dzięki temu programiści mogą używać nazw zmiennych, nie zwracając na nie zbytniej uwagi.

Wbudowane operatory Pythona sprawdzają, czy zastosowane operandy spełniają wymagania danego operatora. Jednak przedstawiona wcześniej przykładowa definicja funkcji nie zawierała żadnych mechanizmów kontroli typów, która jest przeprowadzana w trakcie wykonywania kodu. Co więcej, wcale nie chcielibyśmy dodawać do niej kodu, który pozwalałby na taką kontrolę. Zamiast tego używamy narzędzi kontrolujących kod w ramach wykonywanych testów. Możemy dodać do kodu adnotacje, nazywane **podpowiedziami typów**, i użyć narzędzia, które będzie sprawdzać zgodność naszego kodu z tymi podpowiedziami.

W pierwszej kolejności przyjrzymy się adnotacjom. W kilku kontekstach za nazwą zmiennej można umieścić znak dwukropka (`:`), a po nim nazwę typu. Taki zapis możemy stosować na liście parametrów funkcji (i metod) oraz w instrukcjach przypisania. Co więcej, w definicji funkcji (lub metody klasy) możemy użyć zapisu `->`, aby określić oczekiwany typ wartości wynikowej.

Następny przykład pokazuje, jak wygląda podpowiedź typu:

```
>>> def odd(n: int) -> bool:
...     return n % 2 != 0
```

Do definicji naszej małej funkcji `odd()` dodaliśmy dwie podpowiedzi typów. Określiliśmy, że wartości argumentów dla parametru `n` powinny być liczbami całkowitymi. Oprócz tego wskazaliśmy także, że wynik zwracany przez funkcję będzie jedną z dwóch możliwych wartości typu logicznego (ang. *Boolean type*).

Choć te podpowiedzi zajmują trochę miejsca, to nie wywierają żadnego wpływu na wykonywanie kodu. Python bardzo uprzejmie je zignoruje; innymi słowy: podpowiedzi typu są opcjonalne. Jednak inni programiści czytający taki kod będą zachwyceni, kiedy je zobaczą. Podpowiedzi typów stanowią bowiem doskonały sposób informowania czytelników o naszych zamiarach. Możesz je pomijać podczas nauki, jednak później przekonasz się, że są nieocenione, i pokochasz je, kiedy będziesz musiał wrócić do jakiegoś kodu i go zmodyfikować.

Do sprawdzania spójności podpowiedzi typów bardzo często jest używane narzędzie *mypy*. Nie jest ono wbudowanym elementem Pythona, więc trzeba je osobno pobrać i zainstalować. Więcej informacji o wirtualnych środowiskach oraz instalowaniu narzędzi podamy w dalszej części tego rozdziału, w podrozdziale „Biblioteki innych twórców". Możemy je zainstalować, używając polecenia `python -m pip install mypy` lub — w razie korzystania z narzędzia *conda* — przy użyciu polecenia `conda install mypy`.

Załóżmy, że dysponujemy plikiem *bad_hints.py* zapisanym w katalogu *src*, który zawiera dwie funkcje oraz kod wywołujący funkcję `main()`:

```
def odd(n: int) -> bool:
    return n % 2 != 0

def main():
    print(odd("Witaj, świecie!"))

if __name__ == "__main__":
    main()
```

Możemy sprawdzić ten plik, wykonując z poziomu wiersza poleceń komendę `mypy`:

```
$ mypy --strict src/bad_hints.py
```

Narzędzie *mypy* wykryje kilka potencjalnych problemów, w tym trzy przedstawione poniżej:

```
src/bad_hints.py:12: error: Function is missing a return type annotation
src/bad_hints.py:12: note: Use "-> None" if function does not return a value
src/bad_hints.py:13: error: Argument 1 to "odd" has incompatible type "str";
↪expected "int"
```

Instrukcja `def main():` jest w *wierszu 12.* pliku, ponieważ na jego początku znajduje się blok komentarzy, których tu nie pokazaliśmy. W Twojej wersji pliku pierwszy błąd może występować w *wierszu 1*. Oto dwa problemy, które znalazło narzędzie *mypy*:

- Funkcja `main()` nie ma określonego typu wyniku; narzędzie sugeruje dodanie do niej fragmentu `-> None`, by wyraźnie zaznaczyć, że faktycznie funkcja niczego nie zwraca.
- Ważniejszy jest jednak błąd w *wierszu 13.*: Python będzie próbował przetwarzać funkcję `odd()`, do której próbujemy przekazać wartość typu `str`. Nie odpowiada to podpowiedzi typu podanej w definicji funkcji `odd()`, co może oznaczać następny potencjalny błąd.

W większości przykładów prezentowanych w tej książce będziemy używać podpowiedzi typów. Uważamy, że choć są opcjonalne, to zawsze pomagają, a zwłaszcza w kontekście edukacyjnym. Ponieważ Python zazwyczaj elastycznie podchodzi do zagadnienia typów, istnieje kilka przypadków, w których zachowanie kodu jest trudno opisać przy wykorzystaniu zwięzłych i sugestywnych podpowiedzi. Takich problematycznych przypadków będziemy unikać w tej książce.

Dokument Python Enhancement Proposals (PEP) 585 przedstawia kilka nowych możliwości języka, których celem jest uproszczenie podpowiedzi typów. My testowaliśmy wszystkie prezentowane w książce przykłady przy użyciu narzędzia *mypy* w wersji 0.812. Wszystkie starsze wersje tego programu będą miały problemy z analizowaniem nowszej składni i technik stosowania adnotacji.

Skoro już wyjaśniliśmy, w jaki sposób można opisywać parametry i atrybuty z wykorzystaniem podpowiedzi typów, spróbujmy napisać klasę.

Tworzenie klas w Pythonie

Nie trzeba napisać wiele kodu w Pythonie, by zauważyć, że jest to bardzo *przejrzysty* język. Kiedy chcemy coś zrobić, możemy po prostu to zrobić, bez implementowania rozbudowanego dodatkowego kodu. Klasyczny program „Witaj, świecie!" napisany w Pythonie ma tylko jeden wiersz długości.

Podobnie najprostsza możliwa klasa, którą można napisać w języku Python 3, ma następującą postać:

```
class MyFirstClass:
    pass
```

To nasz pierwszy program obiektowy! Definicja klasy zaczyna się od słowa kluczowego `class`. Za nim należy podać nazwę (dowolną) identyfikującą klasę, a na końcu wiersza umieścić dwukropek.

Wskazówka

Nazwa klasy musi odpowiadać standardowym zasadom określania nazw zmiennych stosowanym w Pythonie (musi zaczynać się od litery lub znaku podkreślenia i musi składać się wyłącznie z liter, znaków podkreślenia albo cyfr). Co więcej, zgodnie z wytycznymi dotyczącymi stylu pisania kodu w języku Python (wyszukaj w internecie frazę *PEP 8*) nazwy klas powinny być zapisywane według notacji **CapWords** (jak ją nazywa dokument PEP 8; zakłada ona, że nazwa ma się zaczynać od dużej litery, a wszystkie następne słowa na nią się składające też mają być zapisywane dużą literą).

Za wierszem definicji klasy podawana jest treść klasy, przy czym jej kod musi być odpowiednio wcięty. W klasach, podobnie jak we wszystkich innych instrukcjach w Pythonie, do wyznaczania granic bloków kodu są używane wcięcia, a nie nawiasy klamrowe stosowane w wielu innych językach. Co więcej, zgodnie z wytycznymi stylu pisania w Pythonie, wcięcia te powinny składać się z czterech spacji, chyba że mamy ważkie powody, by stosować inny sposób formatowania (np. musimy dostosować się do kodu innego programisty, który zamiast spacji do tworzenia wcięć używał znaków tabulacji).

Ponieważ w naszej pierwszej klasie nie ma ani żadnych danych, ani zachowań, zastosowaliśmy słowo kluczowe `pass`, które informuje, że w przypadku tej klasy żadne dodatkowe czynności nie są potrzebne.

Moglibyśmy sądzić, że z tą najprostszą z możliwych klas nie możemy już nic więcej zrobić. Okazuje się jednak, że możemy: możemy tworzyć instancje tej klasy. Możemy wczytać tę klasę do interpretera Pythona 3, co pozwoli nam na przeprowadzanie z nią interaktywnych eksperymentów. W tym celu zapisz definicję klasy w pliku *first_class.py*, a następnie wykonaj polecenie: `python -i first_class.py`. Argument `-i` informuje, że Python ma *wykonać wskazany plik, a następnie przejść do interaktywnego interpretera*. Poniższa sesja interpretera przedstawia podstawowe interakcje z naszą pierwszą klasą:

```
>>> a = MyFirstClass()
>>> b = MyFirstClass()
>>> print(a)
<__main__.MyFirstClass object at 0x000001F6EF40DCC0>
>>> print(b)
<__main__.MyFirstClass object at 0x000001F6EF40DC30>
```

W tym fragmencie kodu tworzymy dwa obiekty, które są instancjami naszej nowej klasy, i przypisujemy je zmiennym `a` i `b`. Utworzenie instancji klasy sprowadza się do zapisana jej nazwy i dodania do niej pary nawiasów. Ten zapis wygląda bardzo podobnie do wywołania funkcji: **wywołanie** klasy spowoduje utworzenie nowego obiektu. Kiedy wyświetlimy obiekt, zobaczymy na ekranie nazwę jego klasy oraz adres pamięci, pod którym obiekt został zapisany. W kodzie pisanym w języku Python adresy nie są używane zbyt często, jednak w tym przykładzie pokazują one, że mamy do czynienia z dwoma odrębnymi obiektami.

O tym, że obiekty są odrębne, możemy się także przekonać, używając operatora `is`:

```
>>> a is b
False
```

Operator ten może nam pomóc w sytuacjach, kiedy utworzymy kilka obiektów, zapiszemy je w kilku zmiennych, a potem stracimy rozeznanie, co i gdzie zapisaliśmy.

Dodawanie atrybutów

Dysponujemy zatem prostą klasą, choć na razie jest ona raczej mało przydatna — nie zawiera żadnych danych ani niczego nie robi. W jaki sposób przypisać konkretnemu obiektowi jakiś atrybut?

Okazuje się, że aby dodać atrybut do obiektu, nie musimy robić niczego szczególnego z definicją klasy. Dowolne atrybuty można dodawać bezpośrednio do konkretnych obiektów; wystarczy w tym celu zastosować notację z kropką, którą przedstawiliśmy w poniższym przykładzie:

```
class Point:
    pass

p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

Kiedy wykonamy ten kod, dwie instrukcje `print` umieszczone na jego końcu wyświetlą wartości atrybutów obu utworzonych obiektów:

```
5 4
3 6
```

W tym przykładzie tworzymy pustą klasę `Point`, w której nie definiujemy żadnych danych ani zachowań. Następnie tworzymy dwie instancje tej klasy i dla każdej z nich określamy wartości współrzędnych `x` i `y`, by identyfikować punkt w przestrzeni dwuwymiarowej. Aby przypisać wartość atrybutowi obiektu, wystarczy użyć zapisu `<obiekt>.<atrybut> = <wartość>`. Taka postać zapisu jest często nazywana **notacją z kropką**. Wartość przypisywana atrybutowi może być dowolna: może to być wartość typu podstawowego czy jakiegoś wbudowanego typu danych lub inny obiekt. Może to być nawet funkcja albo inna klasa!

Jednak tworzenie atrybutów w taki sposób jest bardzo mylące dla narzędzia *mypy*. Nie istnieje żaden łatwy sposób dodania do takiej definicji klasy `Point` podpowiedzi typów. Możemy dodawać podpowiedzi w instrukcji przypisania, np. `p1.x: float = 5`. Ogólnie rzecz biorąc, istnieje o wiele lepszy sposób stosowania atrybutów i podpowiedzi typów, który przedstawiamy nieco dalej, w punkcie „Inicjalizacja obiektów”. Jednak zanim do niego dojdziemy, pokażemy, jak można dodawać do definicji klasy zachowania.

Zapewnianie możliwości działania

Choć możliwość dodawania do obiektów atrybutów jest fantastyczna, to główną ideą programowania obiektowego jest tworzenie i stosowanie interakcji pomiędzy obiektami. Interesuje nas zatem wywoływanie akcji, które sprawią, że z atrybutami obiektów coś się będzie dziać. Dane już mamy, nadszedł zatem czas, by zacząć dodawać do klas zachowania.

Spróbujmy zamodelować w naszej klasie `Point` parę akcji. Zaczniemy od **metody** o nazwie `reset`, która będzie przenosić dany punkt do początku układy współrzędnych (czyli do miejsca, w którym współrzędne `x` i `y` mają wartość `0`). Taka akcja doskonale nadaje się do przedstawienia na początku, ponieważ nie wymaga żadnych parametrów:

```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

Instrukcja `print` wyświetli dwa zera, będące wartościami atrybutów `x` i `y` obiektu:

```
0 0
```

W Pythonie metody są zapisywane dokładnie tak samo jak funkcje. Zaczynają się od słowa kluczowego `def`, po którym następuje spacja oraz nazwa metody. Za nazwą można podać zbiór parametrów zawierający listę parametrów metody (parametrem `self`, nazywanym czasami zmienną instancyjną, zajmiemy się już za moment), po którym należy zapisać znak dwukropka. Następne wiersze kodu są dodatkowo wcięte i zawierają instrukcje będące zawartością metody. Instrukcje te mogą być dowolnym kodem w Pythonie, który metoda chce wykonać, operującym na samym obiekcie oraz na przekazanych parametrach.

W definicji metody `reset()` pominęliśmy podpowiedzi typów, ponieważ nie jest ona przykładem metody, w której takie podpowiedzi będą powszechnie stosowane. Znacznie lepsze okazje do stosowania podpowiedzi typów przedstawiamy nieco dalej, w punkcie „Inicjalizacja obiektów”.

Rozmowy z samym sobą

Jedną z kluczowych różnic pomiędzy metodami klas i funkcjami — pod względem syntaktycznym — jest to, że metody mają jeden wymagany argument. Zwyczajowo nosi on nazwę `self` — nie udało się nam spotkać żadnego programisty Pythona, który nadawałby tej zmiennej inną nazwę (niektóre konwencje naprawdę mają moc!). W rzeczywistości, z technicznego punktu widzenia, nic nie stoi na przeszkodzie, by nadać jej dowolną inną nazwę, np. `this` lub `Marta`; choć najlepiej będzie ulec presji społeczności Pythona, mającej swój wyraz w dokumencie PEP 8, i stosować nazwę `self`.

Argument `self` przekazywany do metody jest referencją do obiektu, na rzecz którego metoda ta została wywołana. Obiekt jest instancją klasy, dlatego też `self` jest czasami nazywany zmienną instancyjną.

Za pomocą zmiennej `self` możemy odwoływać się do atrybutów i metod tego obiektu. I właśnie to robimy w kodzie metody `reset` — używamy `self`, by ustawić wartości atrybutów `x` i `y` obiektu.

> **Wskazówka**
>
> Zwróć uwagę na różnicę pomiędzy **obiektem** i **klasą** w tych rozważaniach. **Metodę** możemy sobie wyobrażać jako funkcję skojarzoną z klasą. Parametr `self` odwołuje się do konkretnej instancji klasy. Kiedy dwa razy wywołujemy metodę na rzecz dwóch różnych obiektów, dwukrotnie wywołujemy tę samą metodę, jednak w każdym z tych wywołań parametr `self` będzie odnosić się do innego **obiektu**.

Zważ, że kiedy wywołujemy metodę `p.reset()`, nie przekazujemy do niej jawnie argumentu `self`. Python automatycznie robi to za nas. Wie, że wywołujemy metodę na rzecz obiektu `p`, więc automatycznie przekazuje ten obiekt do metody klasy `Point`.

Dla niektórych pewnym ułatwieniem może być wyobrażenie sobie, że funkcja jest częścią klasy. Zamiast wywoływać metodę na rzecz obiektu, możemy wywołać funkcję tak, jak została zdefiniowana w klasie, i jawnie przekazać do niej obiekt jako argument `self`:

```
>>> p = Point()
>>> Point.reset(p)
>>> print(p.x, p.y)
```

Otrzymany w tym przykładzie wynik jest dokładnie taki sam jak wcześniej, ponieważ wewnętrznie realizowany jest dokładnie ten sam proces. Takie rozwiązanie nie jest uznawane za dobrą praktykę programistyczną, jednak może pomóc ugruntować rozumienie argumentu `self`.

A co się stanie, jeśli w definicji metody zapomnimy podać argument `self`? Otóż w takim przypadku Python wyświetli komunikat o błędzie:

```
>>> class Point:
...     def reset():
...         pass
```

```
...
>>> p = Point()
>>> p.reset()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Point.reset() takes 0 positional arguments but 1 was given
```

Komunikat błędu nie jest tak przejrzysty, jak mógłby być (komunikat taki jak: „Hej, głupku, zapomniałeś zdefiniować w metodzie parametr `self`", byłby znacznie bardziej zrozumiały).

Wystarczy, byś zapamiętał, że jeśli zobaczysz komunikat błędu informujący o brakujących argumentach, to pierwszą rzeczą, jaką powinieneś sprawdzić, jest to, czy w definicji metody nie zapomniałeś parametru `self`.

Więcej argumentów

A jak przekazywać do metody więcej argumentów? Dodajmy do naszej przykładowej klasy nową metodę, która pozwoli nam przesuwać punkt w dowolnie wybrane miejsce, a nie jedynie do początku układu współrzędnych. Możemy także dodać do klasy metodę pobierającą inny obiekt `Point` i zwracającą odległość pomiędzy oboma punktami.

```
import math

class Point:
    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def reset(self) -> None:
        self.move(0, 0)

    def calculate_distance(self, other: "Point") -> float:
        return math.hypot(self.x - other.x, self.y - other.y)
```

Zdefiniowaliśmy klasę z dwoma atrybutami, `x` i `y`, oraz trzema odrębnymi metodami: `move()`, `reset()` i `calculate_distance()`.

Metoda `move()` oczekuje przekazania dwóch argumentów, `x` i `y`, i używa ich do ustawienia atrybutów `x` i `y` obiektu `self`. Metoda `reset()` wywołuje metodę `move()`, ponieważ jej działanie sprowadza się do przesunięcia punktu w konkretne, znane miejsce.

Metoda `calculate_distance()` oblicza euklidesową odległość pomiędzy dwoma punktami. (Jest także kilka innych sposobów rozumienia odległości. Kilka alternatyw dla odległości euklidesowej przedstawiliśmy w studium przypadku zamieszczonym w rozdziale 3. „Kiedy obiekty są do siebie podobne"). Mamy nadzieję, że pamiętasz jeszcze co nieco z matematyki. Definicja odległości wygląda następująco: $\sqrt{(x_s - x_o)^2 + (y_s - y_o)^2}$ — z tego wzoru korzysta funkcja `math.hypot`. W kodzie Pythona będziemy używać zapisu `self.x`, lecz matematycy będą preferować zapis x_s.

Poniżej przedstawiliśmy przykład zastosowania definicji naszej klasy `Point`. Pokazuje on, w jaki sposób można wywoływać metody z argumentami: wystarczy zapisać te argumenty wewnątrz nawiasów i użyć standardowej notacji z kropką, by odwołać się do nazwy metody konkretnej instancji obiektu. Aby przetestować metody, wybraliśmy dwa dowolne punkty. Testowy kod wywołuje każdą metodę i wyświetla wyniki w konsoli.

```
>>> point1 = Point()
>>> point2 = Point()
>>>
>>> point1.reset()
```

```
>>> point2.move(5, 0)
>>> print(point2.calculate_distance(point1))
5.0
>>> assert point2.calculate_distance(point1) == point1.calculate_distance(point2)
>>> point1.move(3,4)
>>> print(point1.calculate_distance(point2))
4.47213595499958
>>> print(point1.calculate_distance(point1))
0.0
```

Instrukcja `assert` jest niesamowitym narzędziem do testowania; działanie programu zostanie natychmiast przerwane, jeśli okaże się, że wyrażenie podane za `assert` przyjmuje wartość `False` (lub `0`, `None` lub wartość pustą). W tym przypadku zastosowaliśmy tę instrukcję, by upewnić się, że odległość będzie taka sama niezależnie od tego, którego punktu użyliśmy do wywołania metody `calculate_distance()`. Znacznie więcej przykładów jej stosowania przedstawiliśmy w rozdziale 13. „Testowanie oprogramowania obiektowego", w którym zajmujemy się m.in. pisaniem bardziej rygorystycznych testów.

Inicjalizacja obiektów

Jeśli nie określimy współrzędnych `x` i `y` obiektu `Point`, czy to przy wykorzystaniu metody `move`, czy poprzez bezpośrednie przypisanie im wartości, to będziemy dysponować bezużytecznym punktem, który nie będzie wskazywał żadnego realnego miejsca. Co się stanie, kiedy spróbujemy odwołać się do takiego punktu?

No cóż, spróbujmy i przekonajmy się. Takie *eksperymenty* są niezwykle użytecznym narzędziem podczas nauki języka Python. Otwórz interaktywny interpreter Pythona i wpisz w nim odpowiedni kod (ten interaktywny interpreter był jednym z narzędzi, których używaliśmy podczas pisania tej książki).

Poniższa sesja pokazuje, co się dzieje, kiedy spróbujemy odwołać się do brakującego atrybutu. Jeśli zapisałeś wcześniejszy przykład w pliku lub jeśli używasz przykładów dołączonych do książki, możesz wczytać ten plik do interaktywnego interpretera Pythona, wykonując polecenie `python -i more_arguments.py`:

```
>>> point = Point()
>>> point.x = 5;
>>> print(point.x)
5
>>> print(point.y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'y'
```

No cóż, przynajmniej Python zgłosił całkiem przydatny wyjątek. Wyjątki opisaliśmy dokładniej w rozdziale 4. „Oczekując nieoczekiwanego". Prawdopodobnie już wcześniej spotkałeś się z wyjątkami (a zwłaszcza z wszechobecnym wyjątkiem `SyntaxError`, oznaczającym, że coś nieprawidłowo wpisaliśmy). Jak na razie po prostu powinieneś zapamiętać, że wyjątek oznacza, że coś poszło źle.

Wyświetlający się komunikat wyjątku przydaje się podczas debugowania i poprawiania kodu. W przedstawionym przykładzie komunikat informuje, że błąd znajduje się w *wierszu 1.*, co tylko częściowo jest zgodne z prawdą (w interaktywnej sesji Pythona instrukcje są wykonywane pojedynczo). Gdybyśmy jednak wykonywali kod zapisany w pliku, to komunikat zawierałby numer wiersza, w którym wystąpił problem. Co więcej, z komunikatu dowiadujemy się, że wystąpił wyjątek `AttributeError`, a dodatkowo otrzymujemy informację, co dany błąd oznacza.

Możemy przechwycić błąd i rozwiązać problem, jednak w tym przypadku powinniśmy raczej określić jakiegoś rodzaju wartości domyślne atrybutów. Być może każdy nowy obiekt powinien domyślnie wywoływać metodę `reset()`. Albo może byłoby dobrze, gdybyśmy mogli wymusić na użytkowniku określanie położenia każdego tworzonego obiektu.

Co ciekawe, narzędzie *mypy* nie jest w stanie określić, czy `y` ma być atrybutem obiektu `Point`. Atrybuty z definicji są dynamiczne, więc nie ma żadnej prostej listy wskazującej, co jest elementem definicji klasy. Jednak w Pythonie są stosowane pewne popularne konwencje, które ułatwiają określanie oczekiwanego zbioru atrybutów.

W większości obiektowych języków programowania występuje pojęcie **konstruktora** (ang. *constructor*) — specjalnej metody, która inicjuje obiekt podczas jego tworzenia. W Pythonie wygląda to nieco inaczej: w tym języku mamy do czynienia nie tylko z konstruktorem, lecz także z inicjalizatorem. Metoda konstruktora, `__new__()`, jest używana sporadycznie, chyba że robimy coś bardzo egzotycznego. Dlatego też w pierwszej kolejności zajmiemy się znacznie częściej stosowaną metodą inicjalizującą: `__init__()`.

Metoda inicjalizująca Pythona niczym nie różni się od innych metod, z tym że ma ściśle określoną nazwę: `__init__()`. Sekwencje dwóch znaków pokreślenia przed i za słowem `init` oznaczają, że jest to specjalna metoda, którą interpreter Pythona będzie traktował jako przypadek szczególny.

> **Wskazówka**
>
> Nigdy samemu nie definiuj własnych metod, których nazwy będą zaczynać się i kończyć sekwencją dwóch znaków podkreślenia. Obecnie takie metody mogą nie mieć żadnego znaczenia dla interpretera Pythona, jednak zawsze istnieje prawdopodobieństwo, że projektanci Pythona w przyszłości dodadzą do niego funkcję o specjalnym znaczeniu i takiej nazwie. A jeśli tak zrobią, Twój kod przestanie działać.

Dodajmy zatem do naszej klasy `Point` funkcję inicjalizującą, która będzie wymagać, by użytkownik podał współrzędne `x` i `y` podczas tworzenia nowych instancji tej klasy:

```
class Point:
    def __init__(self, x: float, y: float) -> None:
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        self.x = x
        self.y = y
```

```
    def reset(self) -> None:
        self.move(0, 0)

    def calculate_distance(self, other: "Point") -> float:
        return math.hypot(self.x - other.x, self.y - other.y)
```

Teraz obiekty klasy `Point` należy tworzyć w następujący sposób:

```
point = Point(3, 5)
print(point.x, point.y)
```

Nie będzie już można utworzyć obiektu `Point` bez określania współrzędnych `x` i `y`! Jeśli spróbujemy utworzyć instancję klasy `Point` bez dostarczenia prawidłowych parametrów inicjalizacyjnych, zostanie zgłoszony wyjątek podobny do napotkanego wcześniej błędu o „braku wymaganej liczby argumentów", który był zgłaszany, kiedy zapomnieliśmy dodać parametr `self` do definicji metody.

W większości przypadków w funkcji `__init__()` umieszczamy kod związany z inicjalizacją obiektu. Bardzo duże znaczenie ma zadbanie o to, by w tej funkcji określić wartości wszystkich atrybutów obiektu. W ten sposób możemy znacząco ułatwić pracę narzędziu *mypy*, ponieważ wszystkie atrybuty są ustawiane w jednym, oczywistym miejscu. Ułatwia to także zadanie osobom analizującym nasz kod: nie muszą przeszukiwać kodu całej aplikacji, by znaleźć tajemnicze atrybuty ustawiane poza definicją klasy.

Warto także do parametrów tej metody oraz jej wyniku dodać podpowiedzi typów, choć są one opcjonalne. Za nazwą każdego parametru zapisaliśmy dwukropek i określiliśmy oczekiwany typ wartości. Z kolei za listą parametrów umieściliśmy operator `->`, a za nim podaliśmy typ wartości zwracanej przez metodę.

Podpowiedzi typów i wartości domyślne

Jak już kilkukrotnie zaznaczaliśmy w tekście, podpowiedzi typów są opcjonalne. Ich stosowanie nie ma żadnego wpływu na sposób wykonywania kodu. Istnieją jednak narzędzia, które analizują te podpowiedzi i sprawdzają spójność typów danych używanych w kodzie. Jednym z takich narzędzi, powszechnie stosowanym do sprawdzania zgodności kodu z podpowiedziami typu, jest *mypy*.

Jeśli będziemy chcieli, by oba argumenty nie były wymagane, to możemy zastosować tę samą składnię, która jest używana do definiowania domyślnych wartości argumentów funkcji. Składnia ta wymaga zapisania za parametrem znaku równości i wartości. Jeśli kod wywołujący nie przekaże wartości danego argumentu, zostanie użyta wartość domyślna. Zmienne wciąż będą dostępne wewnątrz funkcji, jednak będą miały wartości określone na liście argumentów. Oto przykład:

```
class Point:
    def __init__(self, x: float = 0, y: float = 0) -> None:
        self.move(x, y)
```

Definicje poszczególnych parametrów mogą być dość długie i sprawiać przez to, że także definicja samej metody będzie bardzo długa. W niektórych przykładach przekonasz się, że wiersz stanowiący jedną logiczną całość będzie zapisywany w kilku fizycznych wierszach kodu. Jest to możliwe dzięki temu, że Python łączy wszystkie fizyczne wiersze kodu umieszczone wewnątrz par nawiasów: (i). Jeśli wywołanie metody jest zbyt długie, możemy ją zapisać w następujący sposób:

```
class Point:
    def __init__(
        self,
        x: float = 0,
        y: float = 0
    ) -> None:
        self.move(x, y)
```

Taki styl zapisu definicji metod nie jest stosowany zbyt często, niemniej jednak jest on prawidłowy i pozwala na skrócenie poszczególnych wierszy kodu oraz ich łatwiejsze czytanie.

Podpowiedzi typów oraz wartości domyślne są bardzo wygodne, ale to nie wszystko, co możemy zrobić, by nasza klasa była łatwa w użyciu i prosta do rozszerzania, gdyby to było konieczne. Dlatego też dodamy do niej dokumentację w formie tzw. napisów dokumentujących (ang. *docstrings*).

Podawanie wyjaśnień w napisach dokumentujących

Python może być językiem programowania, którego kod jest niezwykle łatwy do czytania i analizy; niektórzy mogliby nawet stwierdzić, że sam kod mógłby stanowić dokumentację. Niemniej jednak podczas tworzenia oprogramowania obiektowego niezwykle ważne jest pisanie dokumentacji do interfejsów programowania aplikacji (ang. *Application Programming Interface*, API), zawierającej opisy przeznaczenia wszystkich obiektów oraz działania poszczególnych metod. Zadbanie o to, by ta dokumentacja była aktualna, jest trudnym zadaniem, a najlepszym sposobem na to jest pisanie jej bezpośrednio w kodzie.

W tym celu Python udostępnia tzw. **napisy dokumentujące**. Na początku każdej klasy, funkcji lub metody, poniżej wiersza kończącego się dwukropkiem, można zapisać zwykły łańcuch znaków, poprzedzając go odpowiednim wcięciem.

Napisy dokumentujące są zwyczajnymi łańcuchami znaków w języku Python, umieszczonymi pomiędzy znakami apostrofu (') lub cudzysłowu ("). Często zdarza się, że napisy te są dość długie i mogą zajmować nawet kilka wierszy kodu (wytyczne dotyczące stylu kodu pisanego w języku Python zalecają, by długość jednego wiersza nie przekraczała 80 znaków). W takim przypadku można je sformatować jako wielowierszowe łańcuchy znaków zapisywane pomiędzy sekwencjami trzech znaków apostrofu (''') lub trzech znaków cudzysłowu (""").

Napis dokumentujący powinien w przejrzysty i zwięzły sposób podsumowywać przeznaczenie opisywanej klasy lub metody. Powinien wyjaśniać wszelkie parametry, których znaczenie i zastosowanie nie jest oczywiste; a dobra praktyka zaleca, by w napisach dokumentujących

zamieszczać także krótkie przykłady użycia danego API. Należy w nich także opisać wszelkie kruczki oraz problemy, o których powinien wiedzieć niczego niepodejrzewający użytkownik.

Jedną z najlepszych rzeczy, które można umieszczać w napisach dokumentujących, są konkretne przykłady. Narzędzia takie jak *doctest* potrafią je odnajdywać i potwierdzać, że są one wykonywane prawidłowo. Wszystkie przykłady zamieszczone w tej książce zostały sprawdzone przy użyciu takiego narzędzia.

Aby pokazać zastosowanie napisów dokumentujących, zakończymy ten podrozdział, przedstawiając wyczerpująco skomentowaną klasę `Point`:

```
class Point:
    """
    Reprezentuje punkt w geometrycznej przestrzeni dwuwymiarowej.

    >>> p_0 = Point()
    >>> p_1 = Point(3, 4)
    >>> p_0.calculate_distance(p_1)
    5.0
    """

    def __init__(self, x: float = 0, y: float = 0) -> None:
        """
        Metoda inicjalizuje nowy punkt i umieszcza go w określonym miejscu.
        Można podać współrzędne x i y tego punktu. Jeśli zostaną pominięte,
        punkt zostanie umieszczony w początku układu współrzędnych

        :param x: float współrzędna x
        :param y: float współrzędna y
        """
        self.move(x, y)

    def move(self, x: float, y: float) -> None:
        """
        Metoda przenosi punkt w nowe miejsce w przestrzeni dwuwymiarowej

        :param x: float współrzędna x
        :param y: float współrzędna y
        """
        self.x = x
        self.y = y
    def reset(self) -> None:
        """
        Resetuje położenie punktu, przenosząc go do początku układu
        współrzędnych (punktu 0, 0)
        """
        self.move(0, 0)

    def calculate_distance(self, other: "Point") -> float:
        """
        Metoda oblicza odległość euklidesową danego punktu oraz punktu
        przekazanego jako parametr

        :param other: Point drugi punkt
```

```
        :return: float odległość między punktami
        """
        return math.hypot(self.x - other.x, self.y - other.y)
```

Spróbuj wpisać kod lub wczytać (pamiętaj o możliwości użycia polecenia `python -i point.py`) ten plik w interaktywnym interpreterze Pythona. Następnie w wierszu poleceń interpretera wpisz `help(Point)` i naciśnij klawisz *Enter*. W efekcie powinna zostać wyświetlona atrakcyjnie sformatowana dokumentacja klasy, przedstawiona w poniższym przykładzie:

```
Help on class Point in module __main__:

class Point(builtins.object)
 |  Point(x: float = 0, y: float = 0) -> None
 |
 |  Reprezentuje punkt w geometrycznej przestrzeni dwuwymiarowej.
 |
 |  >>> p_0 = Point()
 |  >>> p_1 = Point(3, 4)
 |  >>> p_0.calculate_distance(p_1)
 |  5.0
 |
 |  Methods defined here:
 |
 |  __init__(self, x: float = 0, y: float = 0) -> None
 |      Metoda inicjalizuje nowy punkt i umieszcza go w określonym miejscu.
 |      Można podać współrzędne x i y tego punktu. Jeśli zostaną pominięte,
 |      punkt zostanie umieszczony w początku układu współrzędnych
 |
 |      :param x: float współrzędna x
 |      :param y: float współrzędna y
 |
 |  calculate_distance(self, other: 'Point') -> float
 |      Metoda oblicza odległość euklidesową danego punktu oraz punktu
 |      przekazanego jako parametr
 |
 |      :param other: Point drugi punkt
 |      :return: float odległość między punktami
 |
 |  move(self, x: float, y: float) -> None
 |      Metoda przenosi punkt w nowe miejsce w przestrzeni dwuwymiarowej
 |
 |      :param x: float współrzędna x
 |      :param y: float współrzędna y
 |
 |  reset(self) -> None
 |      Resetuje położenie punktu, przenosząc go do początku układu
 |      współrzędnych (punktu 0, 0)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
```

```
|
|  __weakref__
|      list of weak references to the object (if defined)
```

Nasza dokumentacja jest w każdym szczególe równie dopracowana, jak dokumentacja wbudowanych funkcji Pythona. Możemy także wykonać polecenie `python -m doctest point_4.py`, by potwierdzić prawdziwość przykładu znajdującego się w napisach dokumentujących.

Dodatkowo możemy też wykorzystać narzędzie *mypy*, by sprawdzić podpowiedzi typów. Użyj polecenia `mypy --strict src/*.py`, żeby sprawdzić poprawność wszystkich plików zapisanych w katalogu *src*. Jeśli narzędzie *mypy* nie znajdzie żadnych problemów, nie wyświetli także żadnych wyników. (Pamiętaj, że *mypy* nie jest elementem standardowej instalacji Pythona — musisz je osobno zainstalować).

Moduły i pakiety

Teraz wiemy już, jak tworzyć klasy i instancje obiektów. Wcale nie trzeba napisać wielu klas, by zacząć tracić nad nimi kontrolę (choć podobnie jest z kodem, w którym nie są stosowane obiekty). W przypadku niewielkich programów wszystkie klasy zazwyczaj umieszczamy w jednym pliku, a na jego końcu dodajemy prosty skrypt, który rozpocznie interakcje. Jednak wraz z powiększaniem się projektów coraz trudniej będzie znaleźć pośród wszystkich zdefiniowanych klas tę jedną, którą chcemy edytować. To właśnie w takich sytuacjach przydają się **moduły** (ang. *modules*). Moduły to nic innego jak zwyczajne pliki Pythona. Pojedynczy plik w naszym niewielkim programie także jest modułem. Dwa pliki źródłowe to dwa moduły. Jeśli w jednym katalogu umieścimy dwa pliki, to klas zdefiniowanych w jednym module będziemy mogli używać w drugim.

W języku Python nazwa modułu odpowiada nazwie pliku bez rozszerzenia *.py*. Plik o nazwie *model.py* jest modułem o nazwie `model`. Pliki modułów znajduje się poprzez przeszukiwanie ścieżki zawierającej katalog lokalny oraz zainstalowanych pakietów.

Do importowania całych modułów oraz konkretnych klas lub funkcji należących do modułów służy instrukcja `import`. Przykład jej użycia przedstawiliśmy już w poprzednim podrozdziale przy okazji prezentowania klasy `Point`. Zastosowaliśmy ją, by zaimportować wbudowany moduł `math` Pythona i użyć dostępnej w nim funkcji `hypot()`, która była nam potrzeba do obliczania odległości w metodzie `calculate_distance()`. Zacznijmy od przedstawienia nowego przykładu.

Załóżmy, że piszemy system do handlu elektronicznego. W takim przypadku zapewne będziemy przechowywali wiele informacji w bazie danych. Wszystkie klasy i funkcje związane z zapewnianiem dostępu do bazy możemy umieścić w odrębnym pliku (nadamy mu jakąś sensowną nazwę, taką jak *database.py*). Dzięki temu inne moduły (np. model klienta, informacje o produktach, magazyn) mogą importować klasy z modułu `database`, by zapewnić sobie możliwość korzystania z bazy danych.

Zacznijmy od modułu o nazwie `database`. Będzie nim plik o nazwie *database.py*, zawierający klasę `Database`. Drugi moduł, o nazwie `products`, będzie odpowiedzialny za zapytania związane z produktami. Klasy umieszczone w tym module muszą tworzyć instancję klasy `Database` z modułu `database`, by móc wykonywać w bazie danych zapytania związane z produktami.

Istnieje kilka różnych wersji składni instrukcji `import`, której możemy użyć, by odwołać się do klasy `Database`. Jedną z tych wersji jest wczytanie całego modułu:

```
>>> import database
>>> db = database.Database("ścieżka/do/bazydanych")
```

Ta wersja instrukcji importuje cały moduł `database` i tworzy przestrzeń nazw `database`. Do każdej klasy i funkcji zaimplementowanych w module `database` możemy się odwołać, używając zapisu `database.<element>`.

Ewentualnie klasę, której potrzebujemy, możemy zaimportować, używając składni `from … import`:

```
>>> from database import Database
>> db = Database("ścieżka/do/bazydanych")
```

Ta wersja instrukcji zaimportowała z modułu `database` tylko klasę `Database`. W przypadku gdy potrzebujemy jedynie paru klas lub funkcji z kilku modułów, to ta wersja instrukcji może być wygodnym uproszczeniem, pozwalającym uniknąć dłuższych, pełnych nazw o postaci `database.Database`. Jednak importowanie wielu elementów z wielu różnych modułów może potencjalnie stanowić źródło pomyłek, kiedy będziemy pomijać w odwołaniach nazwy przestrzeni nazw.

Gdyby z jakiegoś powodu moduł `products` już zawierał klasę `Database` i gdybyśmy nie chcieli, by te dwie klasy były ze sobą mylone, możemy zmienić nazwę klasy importowanej z modułu `products`:

```
>>> from database import Database as DB
>>> db = DB("ścieżka/do/bazydanych")
```

Można także w jednej instrukcji zaimportować kilka elementów z jednego modułu. Gdyby nasz moduł `database` zawierał także klasę `Query`, to poniższa instrukcja pozwoliłaby nam zaimportować obie klasy jednocześnie:

```
from database import Database, Query
```

Aby zaimportować wszystkie klasy i funkcje z modułu `database`, należałoby użyć instrukcji o postaci:

```
from database import *
```

Wskazówka

Nie rób tego! Bardziej doświadczeni programiści Pythona na pewno Ci powiedzą, że nie powinieneś nigdy używać tej składni (nieliczni stwierdzą, że istnieją pewne bardzo szczególne sytuacje, kiedy taka postać instrukcji może się przydać, jednak my się z tym nie zgadzamy). Jednym ze sposobów, by przekonać się, dlaczego nie należy jej stosować, jest zrobić to, a następnie po upływie dwóch lat spróbować zrozumieć, co robił nasz kod. Zaoszczędzimy Ci jednak sporo czasu i dwóch lat borykania się z kiepsko napisanym kodem, podając już teraz krótkie uzasadnienie tej rady!

Oto powody, dla których należy unikać stosowania ostatniej składni instrukcji `import`:

- Kiedy jawnie importujemy klasę `Database`, umieszczając na początku pliku instrukcję `from database import Database`, dokładnie wiadomo, skąd ta klasa pochodzi. Nawet kilkaset wierszy dalej możemy użyć wywołania `db = Database()`, a i tak wystarczy, że szybko rzucimy okiem na sekcję instrukcji importu, by zorientować się, skąd pochodzi klasa `Database`. Jeśli później będziemy potrzebowali dodatkowego wyjaśnienia co do sposobu działania tej klasy, będziemy mogli zajrzeć do pierwotnego pliku modułu (ewentualnie będziemy mogli wczytać ten plik do interaktywnego interpretera i użyć polecenia `help(database.Database)`). Jednak gdybyśmy zastosowali składnię `from database import *`, to określenie, skąd pochodzi konkretna klasa, zajęłoby nam znacznie więcej czasu. Utrzymanie kodu w takim przypadku byłoby prawdziwym koszmarem.
- Jeśli wystąpią jakiekolwiek konflikty nazw, jesteśmy zgubieni. Załóżmy, że dysponujemy dwoma modułami, z których każdy zawiera klasę o nazwie `Database`. Zastosowanie instrukcji `from module_1 import *` oraz `from module_2 import *` oznacza, że druga instrukcja importu nadpisze nazwę `Database` utworzoną przez pierwszą instrukcję. Gdybyśmy natomiast zastosowali instrukcje `import module_1` oraz `import module_2`, musielibyśmy odwoływać się do klasy `Database`, używając pełnych nazw, czyli `module_1.Database` oraz `module_2.Database`.
- Co więcej, w przypadku stosowania normalnej postaci instrukcji `import` większość edytorów kodu może udostępniać dodatkowe narzędzia, takie jak niezawodne uzupełnianie kodu, możliwość przechodzenia do definicji klasy, czy też wyświetlanie dokumentacji. Składnia `import *` może utrudniać działanie tych mechanizmów.
- I w końcu: stosowanie instrukcji `import *` może sprawić, że do naszej lokalnej przestrzeni nazw trafią nieoczekiwane obiekty. Oczywiście instrukcja ta zaimportuje wszystkie klasy i funkcje z podanego modułu, jednak jeśli w module nie została zdefiniowana specjalna lista `__all__`, to instrukcja ta spowoduje także zaimportowanie wszystkich klas i modułów, które są importowane do niego!

Każda nazwa używana w module powinna pochodzić z precyzyjnie zdefiniowanego miejsca, niezależnie, czy została zdefiniowana bezpośrednio w nim, czy też do niego jawnie zaimportowana z innego modułu. Nie powinno być żadnych magicznych zmiennych, które biorą się nie wiadomo skąd. *Zawsze* powinniśmy być w stanie od razu określić, skąd pochodzą poszczególne

nazwy używane w naszej obecnej przestrzeni nazw. Obiecujemy, że jeśli zdecydujesz się użyć tej złej składni, nadejdzie kiedyś moment, gdy pełen frustracji zaczniesz zadawać sobie pytanie: *Skąd, do diabła, pochodzi ta klasa?*

Wskazówka

Dla zabawy spróbuj wykonać w interaktywnym interpreterze Pythona instrukcję `import this`. Powoduje ona wyświetlenie niedługiego wiersza (zawierającego kilka ukrytych żartów) podsumowującego niektóre zasady, jakie starają się praktykować zagorzali zwolennicy Pythona. W kontekście prezentowanych tu informacji zwróć uwagę na wiersz „Explicit is better then implicity"[2]. Jawne importowanie nazw do naszej przestrzeni nazw sprawia, że nasz kod staje się znacznie łatwiejszy do przeanalizowania niż w przypadku stosowania instrukcji `from <moduł> import *`.

Organizowanie modułów

Kiedy tworzony projekt będzie się rozrastał w kolekcję coraz to większej liczby modułów, możemy dojść do wniosku, że przydałby się nam dodatkowy poziom abstrakcji, coś przypominającego zagnieżdżoną hierarchię modułów. Nie możemy jednak umieszczać jednych modułów w innych — w końcu jeden plik może zawierać tylko jeden plik, a przecież moduły są plikami.

Jednak pliki można umieszczać w katalogach i podobnie jest z modułami. **Pakiet** (ang. *package*) to kolekcja modułów umieszczonych w jednym katalogu. Nazwą pakietu jest nazwa katalogu. Musimy jednak zaznaczyć, że dany katalog jest pakietem, by Python mógł odróżnić go do innych katalogów w projekcie. W tym celu w katalogu pakietu należy umieści (zazwyczaj pusty) plik *__init__.py*. Jeśli go pominiemy, nie będziemy mogli importować modułów z tego katalogu.

Zapiszmy zatem parę modułów w pakiecie `ecommerce` znajdującym się w naszym katalogu roboczym zawierającym plik *main.py*, który uruchamia program. Dodatkowo w tym pakiecie `ecommerce` umieścimy jeszcze jeden pakiet, implementujący różne sposoby płatności.

Tworząc głębokie hierarchie pakietów, musimy zachować ostrożność. W społeczności programistów Pythona znana jest następująca dobra rada: „płaska (hierarchia) jest lepsza od zagnieżdżonej". W tym przykładzie utworzenie zagnieżdżonego pakietu jest konieczne, ponieważ wszystkie formy płatności mają pewne wspólne cechy.

Naszą hierarchię pakietów przedstawiliśmy poniżej, jej korzeniem jest katalog *src* umieszczony w katalogu projektu:

```
src/
 +-- main.py
 +-- ecommerce/
```

[2] Jawne jest lepsze od niejawnego — *przyp. tłum.*

```
    +-- __init__.py
    +-- database.py
    +-- products.py
    +-- payments/
    |   +-- __init__.py
    |   +-- common.py
    |   +-- square.py
    |   +-- stripe.py
    +-- contact/
        +-- __init__.py
        +-- email.py
```

Katalog *src* będzie elementem nadrzędnego katalogu projektu. Oprócz katalogu *src* projekty często będą zawierać katalogi o takich nazwach jak *docs* i *tests*. W głównym katalogu projektu zazwyczaj znajdują się także różnego rodzaju pliki konfiguracyjne, np. dla narzędzia *mypy* i podobnych. Do tego zagadnienia wrócimy w rozdziale 13. „Testowanie oprogramowania obiektowego".

Podczas importowania modułów lub klas pomiędzy pakietami musimy zwracać baczną uwagę na strukturę tych pakietów. W języku Python 3 dostępne są dwa sposoby importowania modułów: import bezwzględny i względny.

Import bezwzględny

W przypadku **importu bezwzględnego** (ang. *absolute import*) określamy pełną ścieżkę do elementu (modułu, funkcji lub klasy), który chcemy zaimportować. Jeśli musimy skorzystać z klasy `Product` zdefiniowanej w module `products`, możemy użyć każdej z poniższych składni, by zaimportować ją z wykorzystaniem importu bezwzględnego:

```
>>> import ecommerce.products
>>> product = ecommerce.products.Product("nazwa1")
```

Możemy także pobrać jedną, konkretną klasę z modułu należącego do określonego pakietu:

```
>>> from ecommerce.products import Product
>>> product = Product("nazwa2")
```

Ewentualnie możemy zaimportować cały moduł z podanego pakietu:

```
>>> from ecommerce import products
>>> product = products.Product("nazwa3")
```

W instrukcjach `import` używamy operatora kropki, by oddzielać pakiety od modułów. Pakiet jest przestrzenią nazw zawierającą nazwy modułów, podobnie jak obiekt jest przestrzenią nazw zawierającą nazwy atrybutów.

Te instrukcje będą działać zawsze i w każdym module. Możemy ich użyć do utworzenia klasy `Product` w pliku *main.py*, w module `database`, czy też w każdym z dwóch modułów opłat. Zakładając, że pakiety będą dostępne dla Pythona, interpreter będzie mógł je zaimportować. Pakiety można także zainstalować np. w katalogu Pythona o nazwie *site-packages*, można także użyć zmiennej środowiskowej `PYTHONPATH`, by określić listę katalogów, które Python ma przeglądać w poszukiwaniu pakietów i modułów do zaimportowania.

A którą z tych różnych opcji powinniśmy wybrać? To zależy od użytkowników i samej aplikacji. Jeśli moduł `products` zawiera dziesiątki klas i funkcji, to zazwyczaj będziemy importować nazwę modułu, używając instrukcji o postaci `from ecommerce import products`, a następnie odwoływać się do poszczególnych klas, wykorzystując notację z kropką: `products.Product`. Jeśli jednak potrzebujemy z modułu `products` tylko jednej klasy lub dwóch, to możemy zaimportować je bezpośrednio, używając instrukcji o postaci `from ecommerce.products import Product`. Powinniśmy wybrać taką wersję instrukcji, która w największym stopniu uprości analizę i rozszerzanie kodu innym programistom.

Import względny

W przypadku korzystania z powiązanych ze sobą modułów wchodzących w skład zagnieżdżonej struktury pakietu podawanie pełnych ścieżek może się wydawać nieco nadmiarowe; w końcu dokładnie wiemy, jak nazywa się nasz moduł nadrzędny. Właśnie w takich sytuacjach przydaje się drugi sposób importowania: **import względny** (ang. *relative import*). W tym przypadku położenie klasy, funkcji lub modułu jest określane względem bieżącego modułu. Stosowanie importu względnego ma sens wyłącznie w plikach modułów, a co więcej: wyłącznie w przypadku pakietów o złożonej strukturze.

Jeśli np. w kodzie modułu `products` będziemy chcieli zaimportować klasę `Database` z modułu `database` umieszczonego w tym samym katalogu, to używając importu względnego, możemy to zrobić w następujący sposób:

```
from .database import Database
```

Znak kropki umieszczonej przed nazwą modułu `database` oznacza, że *należy użyć modułu `database` umieszczonego w tym samym pakiecie*. W tym przypadku bieżącym pakietem będzie pakiet zawierający aktualnie edytowany plik *products.py*, czyli pakiet `ecommerce`.

Gdybyśmy edytowali moduł `stripe` w pakiecie `ecommerce.payments`, to moglibyśmy chcieć *użyć pakietu `database` wchodzącego w skład nadrzędnego pakietu*. Moglibyśmy to zrobić w prosty sposób, używając dwóch kropek, jak pokazaliśmy w następnym przykładzie:

```
from ..database import Database
```

W ten sposób możemy używać kropek, by przechodzić na wyższe poziomy hierarchii, jednak w pewnym momencie musi do nas dotrzeć, że tych pakietów jest dużo. Oczywiście możemy zarówno wchodzić na wyższe poziomy hierarchii pakietów, jak i schodzić na niższe. Następny przykład przedstawia prawidłową instrukcję importu funkcji `send_mail` z pakietu `ecommerce.contact` zawierającego moduł `email`, gdybyśmy chcieli użyć tej funkcji w module `payments.stripe`:

```
from ..contact.email import send_mail
```

Ta instrukcja używa dwóch kropek, by odwołać się do *pakietu nadrzędnego względem pakietu `payments.stripe`*, a następnie wykorzystuje normalny zapis z kropką (`pakiet.<moduł>`), by zejść na niższy poziom hierarchii — do pakietu `contact` i umieszczonego w nim modułu `email`.

Import względny nie jest jednak tak użyteczny, jak można by sądzić. Jak już wspomnieliśmy wcześniej, z wiersza *Zen of Python* (możesz go przeczytać po wykonaniu instrukcji `import this`) dowiadujemy się, że „płaska (hierarchia) jest lepsza od zagnieżdżonej". Hierarchia standardowej biblioteki Pythona jest dość płaska — składa się z paru pakietów i jeszcze mniejszej liczby pakietów zagnieżdżonych. Jeśli znasz język Java, to zapewne pamiętasz, że w jej bibliotece standardowej pakiety są głęboko zagnieżdżone; jest to coś, czego społeczność programistów Pythona stara się unikać. Import względny jest potrzebny, by rozwiązywać szczególny problem — występujący wtedy, gdy te same nazwy modułów są używane w wielu pakietach. Ten sposób importowania może się zatem przydać w kilku sytuacjach. Konieczność użycia więcej niż dwóch znaków kropki, by dotrzeć do wspólnego nadrzędnego pakietu, sugeruje, że używaną hierarchię pakietów należy spłaszczyć.

Pakiety jako całość

Możemy importować kod, który będzie sprawiać wrażenie, jakby pochodził bezpośrednio z pakietu, a nie z modułu w tym pakiecie. Jak się przekonasz, także w tym przypadku jest używany moduł, jednak ma on specjalną nazwę, która zostanie ukryta. W naszym przykładzie dysponujemy pakietem `ecommerce` zawierającym dwa pliki modułów o nazwach *database.py* oraz *products.py*. Moduł `database` zawiera zmienną `db`, która jest używana w wielu miejscach kodu. Czy nie byłoby wygodnie, gdybyśmy mogli używać instrukcji importu o postaci `from ecommerce import db`, zamiast musieć za każdym razem pisać `from ecommerce.database import db`?

Czy pamiętasz plik *__init__.py*, który definiuje, że katalog, w którym został umieszczony, jest pakietem? Ten plik może zawierać deklaracje dowolnie wybranych zmiennych i klas, które będą dostępne jako elementy pakietu. W naszym przykładzie, jeśli umieścimy w pliku *ecommerce/__init__.py* następujący wiersz kodu:

```
from .database import db
```

to w pliku *main.py*, jak również w dowolnym innym pliku, będziemy mogli zapewnić sobie dostęp do zmiennej `db`, używając poniższej instrukcji:

```
from ecommerce import db
```

Przydatne jest wyobrażenie sobie, że plik *ecommerce/__init__.py* jest jakby odpowiednikiem pliku *ecommerce.py*. Pozwala on spojrzeć na pakiet `ecommerce` w taki sposób, jakby realizował on nie tylko protokół pakietu, lecz także modułu. Plik *__init__.py* nowego pakietu wciąż może stanowić główny punkt odwołań dla innych modułów używających tego pakietu, jednak kod w tym pakiecie może być podzielony na kilka różnych modułów lub podpakietów.

Radzimy jednak, by w plikach *__init__.py* nie umieszczać zbyt dużo kodu. Programiści przeważnie spodziewają się, że te pliki nie będą zawierać faktycznej logiki działania, i podobnie jak w przypadku, gdy stosowana jest instrukcja importu `from x import *`, mogą czuć się zagubieni, jeśli poszukując deklaracji określonego fragmentu kodu, znajdą ją dopiero w pliku *__init__.py*.

Skoro już przyjrzeliśmy się ogólnie modułom, nadszedł czas, by dowiedzieć się, co można w nich umieszczać. Reguły określające zawartość modułów w Pythonie (w odróżnieniu od wielu innych języków programowania) są dość elastyczne. Jeśli znasz język Java, to przekonasz się, że Python zapewnia nam swobodę łączenia różnych elementów kodu w sposób zrozumiały i treściwy.

Organizowanie kodu w moduły

W języku Python moduły mają duże znaczenie. Każda aplikacja lub serwis internetowy składa się przynajmniej z jednego modułu. Nawet pozornie „prosty" skrypt napisany w Pythonie jest modułem. Wewnątrz każdego modułu można umieszczać zmienne, klasy i funkcje. Moduły mogą zatem stanowić wygodny sposób gromadzenia globalnego stanu bez wywoływania konfliktów nazw. I tak np. importujemy klasę `Database` do różnych modułów i tworzymy jej instancje, jednak może się okazać, że bardziej sensownym rozwiązaniem byłoby globalne utworzenie jednego obiektu `Database`, a następnie udostępnianie go z modułu `database`. Moduł `database` mógłby mieć następującą postać:

```
class Database:
    """Implementacja obsługi bazy danych"""

    def __init__(self, connection: Optional[str] = None) -> None:
        """Tworzy połączenie z bazą danych"""
        self.connection = connection

database = Database("ścieżka/do/danych")
```

Teraz moglibyśmy użyć dowolnej z przedstawionych wcześniej metod importu, by zapewnić sobie dostęp do obiektu `database`, np.:

```
from ecommerce.database import database
```

Problem z przedstawioną klasą polega na tym, że obiekt `Database` jest tworzony natychmiast po pierwszym zaimportowaniu modułu, czyli zazwyczaj podczas uruchamiania programu. Takie rozwiązanie nie zawsze będzie optymalne, ponieważ nawiązanie połączenia z bazą danych może zająć trochę czasu, co z kolei może wydłużyć uruchamianie aplikacji; może się także zdarzyć, że informacje o połączeniu z bazą będą jeszcze niedostępne, ponieważ aplikacja dopiero musi wczytać plik konfiguracyjny. Moglibyśmy opóźnić moment utworzenia obiektu bazy danych aż do czasu, kiedy będzie on faktycznie potrzebny; w tym celu możemy wywołać funkcję `initialize_database()`, która utworzy zmienną w module:

```
db: Optional[Database] = None

def initialize_database(connection: Optional[str] = None) -> None:
    global db
    db = Database(connection)
```

Podpowiedź typu `Optional[Database]` informuje narzędzie *mypy*, że zmienna `db` może zawierać wartość `None` lub może być instancją klasy `Database`. Podpowiedź typu `Optional` została zdefiniowana w module `typing`. Ta podpowiedź może się przydać także w innych miejscach aplikacji, byśmy mogli się upewnić, że wartość zmiennej `db` będzie różna od `None`.

Słowo kluczowe `global` informuje, że zmienna `db`, reprezentująca bazę danych wewnątrz funkcji `initialize_database()`, jest zmienną zdefiniowaną na poziomie modułu, poza funkcją. Gdyśmy nie określili tej zmiennej jako globalnej, Python utworzyłby nową zmienną lokalną, której wartość zostałaby utracona po zakończeniu funkcji, przez co wartość zmiennej zdefiniowanej na poziomie moduły pozostałaby niezmieniona.

Musimy wprowadzić jeszcze jedną modyfikację: zaimportować moduł `database` jako całość. Nie możemy zaimportować obiektu `db` z modułu, ponieważ mogłoby się okazać, że w danej chwili nie został on jeszcze zainicjowany. Musimy się upewnić, że zanim zmienna `db` przyjmie jakąkolwiek sensową wartość, zostanie wywołana funkcja `database.initialize_database()`. Gdybyśmy potrzebowali bezpośredniego dostępu do obiektu bazy danych, moglibyśmy użyć odwołania o postaci `database.db`.

Powszechnie stosowanym rozwiązaniem jest funkcja zwracająca bieżący obiekt bazy danych. Moglibyśmy ją importować wszędzie tam, gdzie będziemy potrzebować dostępu do bazy danych:

```
def get_database(connection: Optional[str] = None) -> Database:
    global db
    if not db:
        db = Database(connection)
    return db
```

Te przykłady wyraźnie pokazują, że cały kod definiowany na poziomie modułu jest wykonywany od razu, gdy moduł zostanie zaimportowany. Instrukcje `class` i `def` tworzą obiekty kodu, które zostaną wykonane później, gdy zostanie wywołana funkcja. Takie rozwiązanie może być nieco problematyczne w przypadku skryptów, które coś wykonują, takich jak główny skrypt naszej przykładowej aplikacji do handlu elektronicznego. Czasami może się zdarzyć, że napiszemy użyteczny, działający program, a później będziemy chcieli zaimportować funkcję lub klasę z tego modułu do innego programu. Jednak jak tylko zaimportujemy moduł, kod umieszczony na jego głównym poziomie natychmiast zostanie wykonany. Jeśli nie zachowamy odpowiedniej ostrożności, może się okazać, że zamiast zaimportować parę funkcji z tego modułu, wykonamy cały program.

Aby uniknąć tego problemu, kod uruchamiający powinniśmy zawsze umieszczać wewnątrz funkcji (zwyczajowo nazywanej `main()`) i wywoływać tę funkcję wtedy, kiedy będziemy wiedzieć, że moduł jest wykonywany jako skrypt, a nie, gdy jest importowany do innego skryptu. Możemy to zrobić, **ochraniając** wywołanie funkcji `main`, czyli umieszczając je w instrukcji warunkowej w sposób przedstawiony w poniższym przykładzie:

```
class Point:
    """
    Reprezentuje punkt w geometrycznej przestrzeni dwuwymiarowej.
    """
    pass
def main() -> None:
    """
    Funkcja robi coś przydatnego

    >>> main()
```

```
    p1.calculate_distance(p2)=5.0
    """
    p1 = Point()
    p2 = Point(3, 4)
    print(f"{p1.calculate_distance(p2)=}")


if __name__ == "__main__":
    main()
```

Klasy `Point` (oraz funkcji `main()`) możemy używać bez obaw. Możemy także zaimportować zawartość tego modułu bez narażania się na to, że jakiś kod zostanie nieoczekiwanie wykonany. Jeśli natomiast wykonamy ten moduł jako program główny, to zostanie wywołana funkcja `main()`.

Przedstawione rozwiązanie działa, ponieważ każdy moduł dysponuje zmienną specjalną o nazwie `__name__` (pamiętasz zapewne, że Python używa sekwencji dwóch znaków podkreślenia do oznaczania elementów o specjalnym znaczeniu, takich jak metoda `__init__` klas). Zmienna ta określa nazwę modułu w momencie, kiedy zostaje on zaimportowany. Kiedy moduł jest wykonywany bezpośrednio, przy użyciu instrukcji `python <moduł>.py`, nie jest importowany, więc jego zmienna `__name__` przyjmuje ustaloną wartość `"__main__"`.

> **Wskazówka**
>
> Wyrób w sobie zwyczaj wykonywania wszystkiego, co skrypt ma realizować, w instrukcji warunkowej `if __name__ == "__main__":`, choćby tylko na wypadek, gdybyś w danym module zaimplementował funkcję, którą kiedyś w przyszłości mógłbyś chcieć importować do jakiegoś innego modułu.

Metody są zatem umieszczane w klasach, które są umieszczane w modułach, a te z kolei są umieszczane w pakietach. Czy to już wszystko?

Okazuje się, że nie. To typowa hierarchia organizacyjna programów pisanych w Pythonie, jednak nie jest jedyną możliwą. Klasy można definiować gdziekolwiek. Zazwyczaj są one definiowane na poziomie modułów, jednak można definiować je także wewnątrz funkcji lub metod; jak pokazaliśmy w tym przykładzie:

```
from typing import Optional

class Formatter:
    def format(self, string: str) -> str:
        pass

def format_string(string: str, formatter: Optional[Formatter] = None) -> str:
    """
    Formatuje łańcuch znaków, używając obiektu formatter, który
    ma dysponować metodą format() pobierającą łańcuch znaków
    """
```

```
    class DefaultFormatter(Formatter):
        """Formatuje łańcuch znaków, zapisując każde słowo dużą literą"""

        def format(self, string: str) -> str:
            return str(string).title()

    if not formatter:
        formatter = DefaultFormatter()

    return formatter.format(string)
```

W tym przykładzie zdefiniowaliśmy klasę `Formatter` jako pewną abstrakcję, wyjaśniając, czym powinna dysponować klasa formatująca. Nie skorzystaliśmy tutaj z definicji abstrakcyjnej klasy bazowej (abstrakcyjne klasy bazowe opisaliśmy szczegółowo w rozdziale 6. „Abstrakcyjne klasy bazowe i przeciążanie operatorów"). Zamiast tego zdefiniowaliśmy metodę, która praktycznie nie robi niczego użytecznego. Użyliśmy w niej natomiast pełnego zestawu podpowiedzi typów, aby mieć pewność, że narzędzie *mypy* będzie znało oczekiwaną postać formalnej definicji.

W ciele funkcji `format_string()` utworzyliśmy klasę wewnętrzną rozszerzającą klasę `Formatter`. Formalizuje ona oczekiwania co do zestawu metod, którymi powinna dysponować nasza klasa wewnątrz funkcji. To połączenie pomiędzy definicją klasy `Formatter`, parametrem `formatter` oraz konkretną definicją klasy `DefaultFormatter` gwarantuje, że przez przypadek o czymś nie zapomnimy lub czegoś nie dodamy.

Tej funkcji możemy używać w sposób, jaki pokazaliśmy na poniższym przykładzie:

```
>>> hello_string = "Witaj, świecie! Jak się masz dzisiaj?"
>>> print(f" input: {hello_string}")
 input: Witaj, świecie! Jak się masz dzisiaj?
>>> print(f"output: {format_string(hello_string)}")
output: Witaj, Świecie! Jak Się Masz Dzisiaj?
```

Funkcja `format_string` pobiera łańcuch znaków oraz opcjonalny obiekt `Formatter`, po czym przetwarza przekazany łańcuch przy użyciu obiektu formatującego. Jeśli w wywołaniu nie zostanie przekazany obiekt `Formatter`, funkcja tworzy instancję swojej lokalnej klasy formatującej — `DefaultFormatter`. Ponieważ klasa jest zdefiniowana wewnątrz zasięgu funkcji, nie będzie można uzyskać do niej dostępu poza tą funkcją. Analogicznie: istnieje także możliwość definiowania funkcji wewnątrz innych funkcji; ogólnie rzecz biorąc, instrukcje w Pythonie mogą być wykonywane w dowolnym miejscu kodu.

Klasy i funkcje wewnętrzne czasami przydają się do tworzenia jednorazowych elementów, których nie warto ani nie trzeba umieszczać na głównym poziomie modułu, bądź też do tworzenia elementów, które mają sens wyłącznie wewnątrz konkretnej metody. Niemniej jednak w kodzie w języku Python spotyka się je sporadycznie.

Pokazaliśmy już, jak tworzyć klasy oraz moduły. Dysponując znajomością tych dwóch technik, możemy zacząć myśleć o pisaniu użytecznego oprogramowania, które będzie w stanie rozwiązywać problemy. Jednak kiedy aplikacja lub usługa rozrośnie się do pokaźnych rozmiarów,

często mogą w niej zacząć występować problemy z granicami. Musimy zatem mieć pewność, że obiekty respektują prywatność innych obiektów, i unikać pisania zagmatwanego kodu, który potrafi zmienić złożone oprogramowanie w wielki splot wzajemnych powiązań, przypominający miskę spaghetti. Chcielibyśmy raczej, by każda klasa była elegancko hermetyzowanym ravioli. Przyjrzyjmy się zatem innemu aspektowi organizacji kodu, zapewniającemu możliwość tworzenia dobrze zaprojektowanego oprogramowania.

Kto ma dostęp do moich danych?

W większości obiektowych języków programowania występuje pojęcie **kontroli dostępu**. Jest ono powiązane z abstrakcją. Niektóre atrybuty i metody obiektu mogą zostać określone jako prywatne, co oznacza, że jedynie dany obiekt będzie miał do nich dostęp. Inne są oznaczane jako chronione — w takim przypadku jedynie dana klasa oraz jej klasy pochodne będą mieć do nich dostęp. Wszystkie pozostałe atrybuty i metody są uznawane za publiczne: wszystkie inne obiekty będą mogły uzyskać do nich dostęp.

Jednak w Pythonie tak nie jest. Python nie wierzy w narzucanie praw, które kiedyś mogą nam przeszkadzać. Zamiast tego Python udostępnia wytyczne i najlepsze praktyki, których stosowanie nie jest jednak w żaden sposób wymuszane. Z technicznego punktu widzenia wszystkie metody i atrybuty klas są publicznie dostępne. Jeśli chcemy zasugerować, że dana metoda nie powinna być używana publicznie, możemy umieścić odpowiednią informację w napisie dokumentującym i wyjaśnić, że dana metoda jest przeznaczona do użytku wewnętrznego (być może warto tę informację uzupełnić o wyjaśnienie, jak działa publiczny interfejs programowania aplikacji danej klasy).

Często przypominamy sobie o tym, mówiąc: „Wszyscy tu jesteśmy dorośli". Nie ma sensu deklarować zmiennej jako prywatnej, skoro i tak każdy ma dostęp do kodu źródłowego.

Zgodnie z konwencją nazwy wszystkich metod i atrybutów przeznaczonych do użytku wewnętrznego są poprzedzane znakiem podkreślenia (_). Programiści używający Pythona będą rozumieć, że znak podkreślenia na początku nazwy będzie oznaczać: *to jest zmienna wewnętrzna, zastanów się trzy razy, zanim spróbujesz odwołać się do niej bezpośrednio*. Jednak interpreter nie udostępnia żadnych mechanizmów, które uniemożliwią takie bezpośrednie użycie, jeśli ktoś uzna, że takie działanie leży w jego najlepiej pojętym interesie. W końcu jeśli ktoś tak właśnie uważa, to dlaczego Python miałby mu w tym przeszkadzać? Nie mamy żadnego pojęcia, w jaki sposób w przyszłości będą używane nasze klasy, a w przyszłych wersjach kodu znak podkreślenia może zostać usunięty. Jednak na razie stanowi on jasny sygnał: informuje, by unikać stosowania danego atrybuty lub metody.

Jest jeszcze jedna rzecz, którą możemy zrobić, by bardzo dobitnie zasugerować, że zewnętrzne obiekty nie powinny bezpośrednio odwoływać się do danej właściwości lub metody: możemy ją poprzedzić dwoma znakami podkreślenia: __. Użycie sekwencji dwóch znaków podkreślenia oznacza, że interpreter Python „udekoruje" nazwę danego atrybutu lub metody. **Dekorowanie nazw** (ang. *name mangling*) oznacza, że metodę wciąż będzie można wywołać spoza

obiektu, jeśli ktoś naprawdę będzie chciał to zrobić, jednak będzie to wymagało dodatkowej pracy; to dobitny sygnał, że żądamy, by dany atrybut pozostał **prywatny**.

W przypadku zastosowania dwóch znaków podkreślenia nazwa właściwości zostaje poprzedzona ciągiem `_<nazwa_klasy>`. Kiedy do takiej zmiennej odwołuje się kod umieszczony wewnątrz klasy, dekoracja jej metody jest automatycznie usuwana. Jeśli jednak będzie chciał odwołać się do niej zewnętrzny kod, sam będzie musiał zadbać o odpowiednie udekorowanie jej nazwy. A zatem dekorowanie nazw nie gwarantuje prywatności, jedynie ją mocno sugeruje. Technika ta jest stosowana bardzo rzadko, a kiedy się ją spotyka, to staje się przyczyną częstych problemów i zamieszania.

> **Wskazówka**
>
> Nie powinieneś tworzyć w swoim kodzie nowych nazw zaczynających się do dwóch znaków podkreślenia; przyniosą Ci one jedynie utrapienie i rozpacz. Przyjmij, że takie nazwy są zarezerwowane wyłącznie dla specjalnych nazw definiowanych wewnętrznie przez Pythona.

Najważniejsze jest to, że hermetyzacja, jako zasada projektowa, gwarantuje, że metody klas będą hermetyzować zmiany stanu atrybutów. To, czy atrybuty (lub metody) są prywatne, czy nie, nie zmienia dobrego projektu, który wypływa ze stosowania hermetyzacji.

Zasada hermetyzacji odnosi się do poszczególnych klas, jak również całych modułów zawierających wiele klas. Odnosi się ona także do pakietów składających się z wielu modułów. Jako programiści obiektowi używający języka Python dbamy o izolowanie odpowiedzialności i przejrzystą hermetyzację cech.

Oczywiście wykorzystujemy Pythona, by rozwiązywać problemy. Okazuje się, że jest dostępna ogromna biblioteka standardowa, która może nam pomóc w tworzeniu przydatnego oprogramowania. Ta ogromna biblioteka standardowa sprawia, że Pythona nazywamy językiem „z naładowanymi bateriami". Już zaraz po jego zainstalowaniu mamy do dyspozycji niemal wszystko, czego możemy potrzebować — nie musimy biec do sklepu, żeby kupić baterie.

Oprócz biblioteki standardowej istnieje jeszcze większy wszechświat pakietów przygotowywanych przez innych twórców. W następnym podrozdziale wyjaśnimy, jak rozszerzać instalację Pythona i dodawać do niej gotowe biblioteki.

Biblioteki innych twórców

Python jest dostarczany wraz ze wspaniałą biblioteką standardową, stanowiącą kolekcję pakietów i modułów — można jej używać na wszystkich komputerach, na których działa sam Python. Jednak bardzo szybko przekonasz się, że nie zawiera ona wszystkiego, czego będziesz potrzebował. Kiedy nastąpi taki moment, będziesz miał do wyboru dwie możliwości:

- Samodzielnie napisać potrzebny pakiet.
- Użyć kodu napisanego przez kogoś innego.

Nie będziemy tu szczegółowo opisywać, jak przekształcać pakiety w biblioteki, jeśli jednak masz problem, który musisz rozwiązać, a jednocześnie nie masz ochoty, by go samodzielnie rozwiązywać (najlepsi programiści są bardzo leniwi i zamiast pisać własny kod, wolą stosować już istniejący i sprawdzony), to najprawdopodobniej wyszukasz odpowiednią bibliotekę w serwisie **Python Package Index**, na stronie *https://pypi.org*. Kiedy znajdziesz odpowiedni pakiet, wystarczy, że go zainstalujesz, używając polecenia `pip`.

Pakiety możemy instalować, wykorzystując polecenie systemowe, takie jak to przedstawione poniżej:

```
% python -m pip install mypy
```

Jeśli spróbujemy je wykonać bez żadnych wcześniejszych przygotowań, to wybrana biblioteka bądź to zostanie zainstalowana bezpośrednio w systemowym katalogu Pythona, bądź to, co jest bardziej prawdopodobne, nie zostanie zainstalowana wcale — zostanie wyświetlony komunikat o braku uprawnień do modyfikacji systemowego Pythona.

W środowisku użytkowników Pythona panuje powszechna zgoda co do tego, że nie należy w żaden sposób zmieniać Pythona stanowiącego element systemu operacyjnego. Wcześniejsze wersje systemu Mac OS X były dostarczane z Pythonem 2.7. Jednak praktycznie nie był on dostępny dla użytkowników końcowych. Najlepiej jest myśleć o nim jako o elemencie systemu operacyjnego, zignorować go i zainstalować nową wersję języka.

Python jest dostarczany wraz z narzędziem o nazwie `env` — programem użytkowym pozwalającym na tworzenie w bieżącym katalogu roboczym **wirtualnego środowiska** Pythona. Kiedy aktywujemy to środowisko, polecenia związane z Pythonem będą używały Pythona z tego środowiska wirtualnego, a nie wersji systemowej. Oznacza to, że jeśli wykonamy polecenie `pip` lub `python`, w ogóle nie będą one operować na Pythonie zainstalowanym w systemie operacyjnym. Poniżej pokazaliśmy, jak można używać programu `env`:

```
cd katalog_projektu
python -m venv env
source env/bin/activate     # w systemie Linux lub macOS
env/Scripts/activate.bat    # w systemie Windows
```

(Informacje na temat stosowania tego programu w innych systemach operacyjnych, w tym wszelkie informacje o sposobach aktywacji wirtualnego środowiska Pythona, można znaleźć na stronie *https://docs.python.org/3/library/venv.html*).

Po aktywowaniu wirtualnego środowiska możemy mieć pewność, że polecenie `python -m pip` zainstaluje nowe pakiety w tym środowisku, pozostawiając wersję systemową Pythona bez zmian. Spróbuj zatem użyć polecenia `python -m pip install mypy`, by zainstalować narzędzie *mypy* w swoim bieżącym środowisku wirtualnym.

Na domowym komputerze, na którym mamy dostęp do plików wymagających wyższych uprawnień dostępowych, czasami będziemy w stanie zainstalować i używać jednej, centralnej, systemowej wersji Pythona. Jednak w korporacyjnym środowisku roboczym, w którym dostęp do pewnych katalogów wymaga specjalnych uprawnień, korzystanie ze środowiska

wirtualnego jest konieczne. Ponieważ środowiska wirtualne zawsze działają (w odróżnieniu od scentralizowanych instalacji systemowych), ogólnie rzecz biorąc, lepszym rozwiązaniem jest stosowanie właśnie środowisk wirtualnych.

Typowe rozwiązanie zakłada tworzenie odrębnego środowiska wirtualnego dla każdego projektu. Środowiska wirtualne można przechowywać gdziekolwiek, jednak dobra praktyka sugeruje, by zapisywać je w tym samym katalogu, w którym znajduje się reszta plików projektu. W razie korzystania z systemów kontroli wersji, takich jak Git, można użyć pliku *.gitignore*, by środowisko nie było umieszczane w repozytorium.

Kiedy zaczynamy prace nad nowym projektem, często tworzymy dla niego katalog i przechodzimy do niego przy użyciu polecenia `cd`. Następnie wykonujemy polecenie `python -m venv env`, by utworzyć środowisko wirtualne, które przeważnie ma prostą nazwę, taką jak `env`, choć czasami używamy bardziej złożonych nazw, takich jak `StudiumPrzypadku39`.

I w końcu możemy użyć odpowiedniego polecenia z dwóch ostatnich wierszy kodu z poprzedniego przykładu (zależnie od używanego systemu operacyjnego), by aktywować środowisko.

Za każdym razem, kiedy będziemy rozpoczynać prace nad projektem, możemy przejść do katalogu i wykonać polecenie `source` (lub `activate.bat`), by skorzystać z wybranego środowiska wirtualnego. W momencie zmiany projektu używamy polecenia `deactivate`, by wyłączyć aktywne środowisko.

Środowiska wirtualne mają kluczowe znaczenie dla oddzielania bibliotek innych twórców od standardowej biblioteki Pythona. Bardzo często zdarzy się, że będziemy pracować nad różnymi projektami, z których każdy będzie używał innej wersji tej samej biblioteki (np. starsza wersja witryny może wykorzystywać bibliotekę Django 1.8, a nowa wersja — Django 2.1). Przechowywanie każdego projektu w odrębnym środowisku wirtualnym bardzo ułatwia korzystanie z jednej bądź drugiej wersji biblioteki. Co więcej, środowiska wirtualne eliminują konflikty, które mogą występować pomiędzy pakietami zainstalowanymi w systemie oraz tymi instalowanymi przy użyciu polecenia `pip`, w przypadku gdybyśmy próbowali zainstalować ten sam pakiet, używając różnych narzędzi. I w końcu: korzystanie ze środowisk wirtualnych pozwala omijać wszelkie ograniczenia związane z uprawnieniami, które system operacyjny może narzucać na systemową wersję Pythona.

Wskazówka

Dostępnych jest kilka różnych narzędzi innych twórców, które umożliwiają efektywne zarządzanie środowiskami wirtualnymi. Należą do nich m.in. `virtualenv`, `pyenv`, `virtualenvwrapper` oraz `conda`. Jeśli pracujesz nad zagadnieniami związanymi z nauką o danych, to zapewne będziesz potrzebował ostatniego z nich, `conda`, by móc instalować bardziej złożone pakiety. Istnieje parę rzeczy, które sprawiają, że dostępnych jest wiele różnych sposobów rozwiązywania problemu zarządzania ogromnym ekosystemem pakietów dla programów pisanych w Pythonie.

Studium przypadku

W tym podrozdziale rozszerzymy nieco obiektowy projekt naszego realistycznego przykładu. Zaczniemy od diagramów **UML** — pomogą one nam przedstawić i podsumować kod, który chcemy napisać.

Rozważymy także różne uwarunkowania związane z implementowaniem definicji klas w języku Python. Zaczniemy od przejrzenia diagramów opisujących klasy, które mamy zdefiniować.

Widok logiczny

Na rysunku 2.2 zamieściliśmy przegląd klas, które mamy zaimplementować. Pochodzą one z modelu przygotowanego w poprzednim rozdziale (z wyjątkiem paru nowych metod).

Nasz podstawowy model danych składa się z czterech klas, przy czym w paru miejscach używamy także ogólnej klasy listy. Miejsca, w których jest ona używana, oznaczyliśmy podpowiedzią typu `List`. Poniżej opisaliśmy kluczowe cztery klasy:

- Klasa `TrainingData` jest pojemnikiem zawierającym dwie listy z próbkami danych: pierwsza z tych list jest używana do uczenia modelu, a druga do jego testowania. Na obu listach są przechowywane obiekty klasy `KnownSample`. Oprócz tego będziemy także używać listy zmieniających się wartości typu `Hyperparameter`. Ogólnie rzecz biorąc, są to wartości „dostrajające", używane do modyfikowania działania modelu. Chodzi o to, by przetestować różne wartości hiperparametrów, aby znaleźć model o najlepszej jakości.

  Umieściliśmy w tej klasie także element metadanych: nazwę danych, których używamy, datę i godzinę, kiedy zostały one przesłane po raz pierwszy, oraz datę i godzinę wykonania testów na modelu.

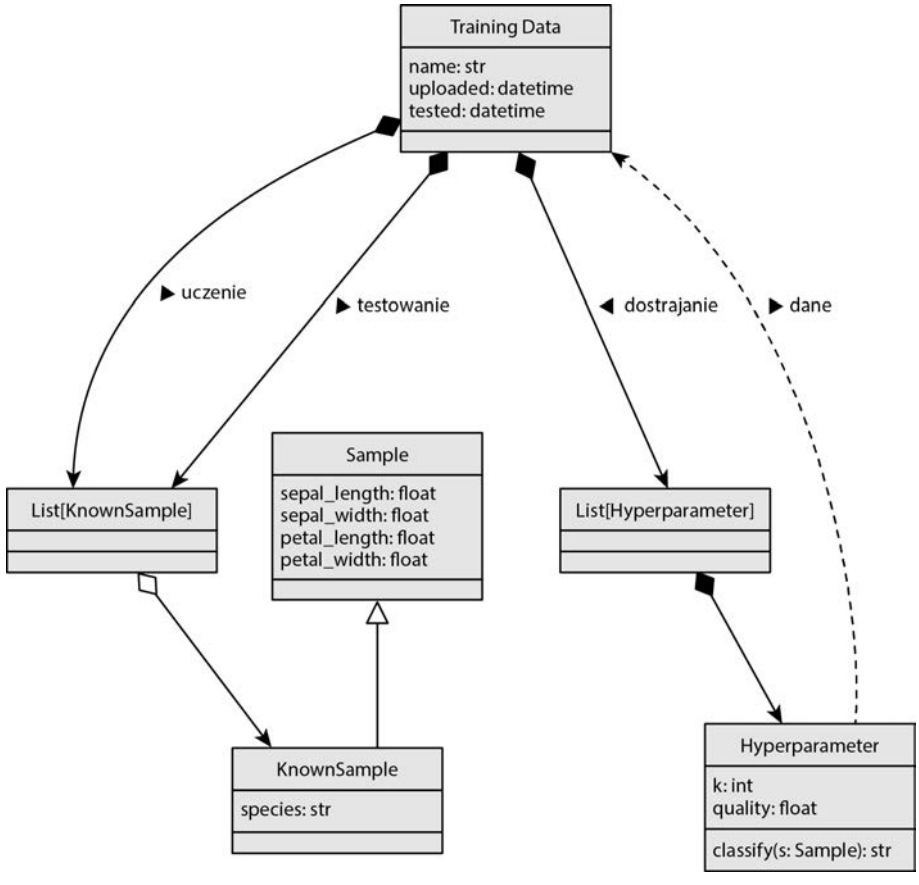
- Każda instancja klasy `Sample` jest kluczowym elementem danych roboczych. W naszym przykładzie tymi danymi są pomiary: długości działek kielicha oraz szerokości i długości płatków korony. Dokładni studenci botaniki uważnie zmierzyli bardzo wiele kwiatów, by zgromadzić te dane. Mamy nadzieję, że mieli choć trochę czasu, by zatrzymać się i powąchać trochę róż podczas pracy.

- Klasa `KnownSample` jest rozszerzoną klasą `Sample`. Ten fragment naszego projektu stanowi zapowiedź zagadnień opisanych w rozdziale 3. „Kiedy obiekty są do siebie podobne". Klasa `KnownSample` jest klasą `Sample` z dodanym jednym atrybutem — określającym nazwę gatunku. Tę informację podają wykształceni botanicy, którzy sklasyfikowali dane używane przez nas do uczenia modelu oraz jego testowania.

- Klasa `Hyperparameter` zawiera wartość *k* używaną do określania liczby uwzględnianych najbliższych sąsiadów. Zawiera ona także informację będącą podsumowaniem testowania modelu z użyciem wartości *k*. Testy jakości mówią nam, ile próbek testowych zostało poprawnie sklasyfikowanych. Oczekujemy,

że dla niewielkich wartości *k* (np. 1 i 3) efekty klasyfikacji nie będą zbyt dobre. Oczekujemy także, że wyniki uzyskiwane dla średnich wartości *k* będą lepsze, a bardzo duże wartości *k* nie będą dawać równie dobrych wyników.

Klasa `KnownSample` przedstawiona na diagramie z rysunku 2.2 nie musi stanowić odrębnej definicji klasy. Podczas dalszych prac przedstawimy alternatywne projekty dla każdej z tych klas.



Rysunek 2.2. Diagram widoku logicznego

Zaczniemy od klas `Sample` i `KnownSample`. Python udostępnia trzy główne sposoby definiowania nowych klas:

- Definicję `class`. To właśnie nią zajmiemy się w pierwszej kolejności.
- Definicję `@dataclass`. Udostępnia ona kilka wbudowanych możliwości. Choć jest użyteczna, to jednak nie jest najlepszym rozwiązaniem dla osób uczących się Pythona, ponieważ może ukrywać pewne ważne szczegóły implementacyjne. Opisaliśmy ją dokładniej w rozdziale 7. „Struktury danych w Pythonie”.

- Rozszerzenie klasy `typing.NamedTuple`. Kluczową cechą tej definicji jest to, że stan tworzonych obiektów jest niezmienny — wartości atrybutów obiektów tej klasy nie mogą się zmieniać. Niezmienne atrybuty mogą być użyteczne, kiedy będziemy chcieli mieć pewność, że błędy w aplikacji nie będą w stanie popsuć naszych danych testowych. Także klasę `typing.NamedTuple` opisaliśmy dokładniej w rozdziale 7.

Naszą pierwszą decyzją projektową jest tworzenie definicji klasy `Sample` oraz jej klasy pochodnej `KnownSample` przy użyciu instrukcji `class`. W przyszłości (np. w rozdziale 7.) będziemy mogli to zmienić i zamiast instrukcji `class` zastosować klasy danych lub klasę `NamedTuple`.

Próbki i ich stan

Klasę `Sample` oraz rozszerzającą ją klasę `KnownSample` przedstawiliśmy już wcześniej na diagramie z rysunku 2.2. Nie wygląda jednak na to, by te dwie klasy stanowiły wyczerpującą dekompozycję różnych rodzajów próbek. Podczas analizy opowieści użytkowników i widoków procesów można odnieść wrażenie, że w projekcie jest pewna luka, a konkretnie rzecz biorąc: „żądanie klasyfikacji" nadsyłane przez użytkownika wymaga przekazania nieznanej próbki. Zawiera ona te same atrybuty pomiarowe co `Sample`, lecz nie dysponuje atrybutem określającym nazwę gatunku, dostępnym w klasie `KnownSample`. Co więcej, nie ma żadnej zmiany stanu, która pozwoliłaby na dodanie wartości tego atrybutu. Ta nieznana próbka nigdy nie zostanie formalnie sklasyfikowana przez botanika — będzie ją klasyfikował nasz algorytm, ale to tylko SI, a nie botanik.

Możemy zatem uwzględnić dwie odrębne klasy pochodne klasy `Sample`:

- `UnknownSample`: Ta klasa będzie zawierać cztery początkowe atrybuty klasy `Sample`. Użytkownik dostarcza obiekty tej klasy, a nasz algorytm je klasyfikuje.
- `KnownSample`: Ta klasa będzie zawierać atrybuty klasy `Sample` oraz wynik klasyfikacji — nazwę gatunku. Obiektów tej klasy będziemy używać do uczenia i testowania modelu.

Ogólnie rzecz biorąc, definicje klas traktujemy jako sposób hermetyzowania stanu i zachowań. Instancje klasy `UnknownSample` dostarczane przez użytkownika nie mają informacji o nazwie gatunku. Później, kiedy algorytm klasyfikatora określi nazwę gatunku, obiekty `Sample` zmienią stan i będą zawierać nazwę gatunku określoną przez algorytm.

Przygotowując definicje klas, zawsze musimy zadać sobie jedno pytanie:

Czy zmiana stanu powoduje jakąś zmianę zachowań?

W naszym przypadku nie wydaje się, by określenie nazwy gatunku powodowało dodanie nowych lub zmianę wcześniejszych zachowań. Może moglibyśmy zatem zaimplementować tylko jedną klasę z jakimiś opcjonalnymi atrybutami?

Jednak istnieje możliwość jeszcze innej zmiany stanu. Obecnie nie ma żadnej klasy, na której spoczywałby obowiązek podzielenia obiektów `Sample` na zbiór uczący i testowy. A to także stanowi pewnego rodzaju zmianę stanu.

Musimy zatem odpowiedzieć sobie na następne ważne pytanie:

Do obowiązków której klasy należy wykonanie tej zmiany stanu?

W naszym przypadku wydaje się, że przygotowanie zbioru uczącego i testowego powinno należeć do odpowiedzialności klasy `TrainingData`.

Jedną z rzeczy, która może nam pomóc dokładnie przyjrzeć się projektowi klas, jest wyliczenie wszystkich możliwych stanów poszczególnych próbek. Ta technika ułatwia określanie nowych atrybutów, które należy dodać do klasy. Oprócz tego ułatwia ona identyfikację metod wprowadzających zmiany stanu w obiektach danej klasy.

Zmiany stanu próbek

Przyjrzyjmy się teraz cyklowi życia obiektów `Sample`. Cykl życia obiektu rozpoczyna się od jego utworzenia, następnie zmienia się jego stan i (w niektórych przypadkach) istnienie obiektu kończy się w momencie, gdy w programie nie ma już żadnych referencji do niego. W naszym przykładowym programie występują trzy scenariusze:

1. **Wczytanie początkowe:** Będziemy potrzebowali metody `load()`, która zapisze w obiekcie `TrainingData` informacje pochodzące z jakiegoś źródła nieprzetworzonych danych. Zajmiemy się tym w rozdziale 9. „Łańcuchy, serializacja i ścieżki do plików", w którym pokażemy, że wczytywanie plików CVS często zwraca sekwencje słowników. Możemy sobie wyobrazić, że metoda `load()` będzie używać czytnika CSV, by tworzyć obiekty `Sample` zawierające specjalną wartość i przekształcać je na obiekty `KnownSample`. Metoda `load()` będzie także dzielić obiekty `KnownSample` na dwie listy, uczącą i testową, co stanowi ważną zmianę stanu obiektu `TrainingData`.
2. **Testowanie hiperparametrów:** W klasie `Hyperparameter` będziemy potrzebowali metody `test()`. Będzie ona operować na próbkach zapisanych w obiekcie `TrainingData` powiązanym z hiperparametrem. Dla każdej próbki metoda ta będzie stosować klasyfikator i zliczać dopasowania pomiędzy gatunkami przypisanymi przez botanika oraz tymi wytypowanymi przez nasz algorytm sztucznej inteligencji. To pokazuje, że potrzebujemy także metody `classify()`, która będzie operować na pojedynczej próbce i będzie używana przez metodę `test()` operującą na grupach próbek. Metoda `test()` będzie aktualizować stan obiektu `Hyperparameter`, określając przechowywaną w im wartość oceny jakości.
3. **Klasyfikacja żądana przez użytkownika:** Internetowe aplikacje typu *RESTful* często są dzielone na odrębne funkcje widoków służące od obsługi żądań. Podczas obsługi żądania klasyfikacji nieznanej próbki funkcja widoku do przeprowadzenia klasyfikacji będzie używać obiektu `Hyperparameter`; obiekt ten zostanie wybrany

przez botanika tak, by zagwarantować najlepsze wyniki. Danymi przekazywanymi przez użytkownika będą instancje klasy `UnknownSample`. Funkcja widoku zastosuje metodę `Hyperparameter.classify()`, aby przygotować odpowiedź zawierającą gatunek irysa przypisany danej próbce przez nasz algorytm klasyfikujący. Ale czy zmiana stanu, która zachodzi, kiedy nasz algorytm klasyfikuje instancję `UnknownSample`, ma tak naprawdę jakieś znaczenie? Oto dwie możliwości:

- Każdy obiekt `UnknownSample` może mieć atrybut `classified`. Ustawienie jego wartości będzie zmianą stanu klasy `Sample`. Nie jest jasne, czy ta zmiana stanu pociąga za sobą jakąś zmianę zachowań.
- Wynik klasyfikacji w ogóle nie będzie elementem klasy `Sample`. Może być przechowywany w zmiennej lokalnej w funkcji widoku. Ta zmiana stanu w funkcji będzie używana do przygotowania odpowiedzi dla użytkownika, jednak w żaden sposób nie wpłynie na obiekt `Sample`.

A oto kluczowy wniosek, płynący z prezentacji tych wszystkich alternatyw:

Wskazówka

Nie ma żadnej „jedynie słusznej" odpowiedzi.

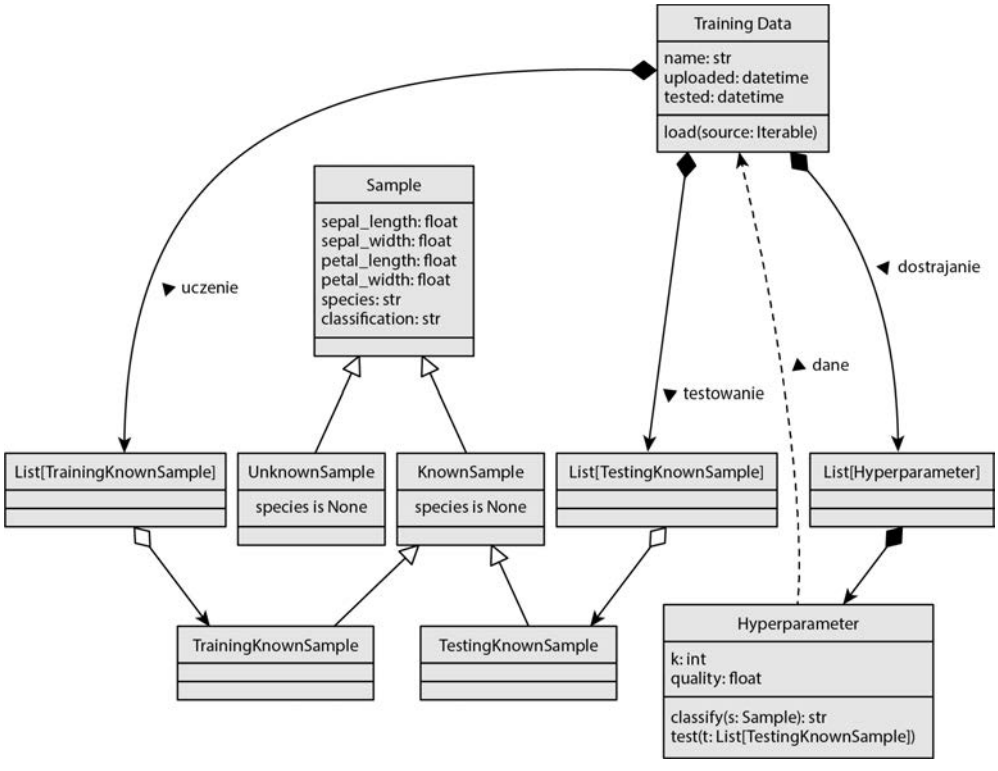
Niektóre decyzje projektowe są podejmowane na podstawie czynników, które nie mają ani charakteru funkcjonalnego, ani technicznego. Mogą one obejmować przewidywany okres użytkowania aplikacji, przyszłe przypadki użycia, dodatkowych użytkowników, którzy mogą zacząć ją wykorzystywać, aktualny terminarz i budżet, wartość edukacyjną, ryzyko techniczne, tworzenie własności intelektualnej oraz to, jak duże wrażenie na uczestnikach konferencji wywrze wersja demonstracyjna aplikacji.

W rozdziale 1. „Projektowanie obiektowe" zamieściliśmy wskazówkę sugerującą, że nasza aplikacja ma być wstępem do przygotowania mechanizmu do rekomendowania produktów. Napisaliśmy: „Docelowo użytkownicy oczekują podejmowania złożonych decyzji dotyczących wyboru produktów konsumenckich, zauważą jednak, że rozwiązywanie trudnych problemów nie jest dobrym sposobem budowania tego rodzaju aplikacji. Znacznie lepiej będzie zacząć od czegoś o niższym poziomie złożoności, a następnie dopracowywać i rozszerzać rozwiązanie aż do momentu, kiedy będzie robić wszystko, czego od niego oczekujemy."

Z tego względu uznajemy, że zmiana stanu z `UnknownSample` na `ClassifiedSample` ma bardzo duże znaczenie. Obiekty `Sample` będą przechowywane w bazie danych w celu używania ich w przyszłych kampaniach reklamowych lub w celu potencjalnego wykonania ponownej klasyfikacji, kiedy pojawią się nowe produkty, a dane uczące zostaną zmodyfikowane.

Zdecydujemy się zachować dane o klasyfikacji i gatunku w klasie `UnknownSample`.

Powyższe rozważania prowadzą do wniosku, że możemy scalić wszystkie szczegóły projektu klasy `Sample` w sposób przedstawiony na rysunku 2.3.



Rysunek 2.3. Zaktualizowany diagram UML

W tym diagramie strzałki z grotem w postaci pustego trójkąta pokazują różne klasy pochodne klasy `Sample`. Jednak nie będziemy ich implementować bezpośrednio jako klas pochodnych. Na diagramie dodaliśmy także strzałki, by pokazać unikalne przypadki użycia dla tych obiektów. Konkretnie rzecz biorąc, prostokąt `KnownSample` zawiera warunek **species in not None**, by pokazać, jaką unikalną cechę mają te obiekty `Sample`. Podobnie prostokąt `UnknownSample` zawiera warunek **species is None**, by pokazać nasze zamierzenia dotyczące obiektów `Sample`, w których atrybut `species` będzie mieć wartość `None`.

W prezentowanych diagramach UML ogólnie staraliśmy się unikać pokazywania „specjalnych" metod Pythona. Pozwala to zmniejszyć wizualny bałagan na diagramach. Jednak w niektórych przypadkach te specjalne metody mogą mieć absolutnie kluczowe znaczenie i warto będzie pokazywać je na diagramach. Implementacja klas niemal zawsze wymaga dodania metody `__init__()`.

Istnieje jeszcze jedna metoda specjalna, która może nam naprawdę pomóc: `__repr__()`. Metoda ta jest używana do przygotowywania reprezentacji obiektu. Reprezentacją tą jest łańcuch znaków, który zazwyczaj ma postać wyrażenia Pythona służącego do odtworzenia danego obiektu. W przypadku prostych liczb reprezentacją tą będzie liczba. W przypadku łańcucha będzie to ten sam łańcuch zapisany w cudzysłowach. W przypadku bardziej złożonych obiektów wyrażenie to będzie zawierało odpowiednią interpunkcję Pythona oraz

wszystkie szczegóły klasy i stanu obiektu. Całkiem często będziemy używali łańcuchów z mechanizmem interpolacji (ang. *f-strings*), podając w nich nazwę klasy oraz wartości atrybutów.

Poniżej przedstawiliśmy początek klasy `Sample`, która zawiera chyba wszystkie szczegóły pojedynczej próbki:

```
class Sample:
    def __init__(
        self,
        sepal_length: float,
        sepal_width: float,
        petal_length: float,
        petal_width: float,
        species: Optional[str] = None,
    ) -> None:
        self.sepal_length = sepal_length
        self.sepal_width = sepal_width
        self.petal_length = petal_length
        self.petal_width = petal_width
        self.species = species
        self.classification: Optional[str] = None

    def __repr__(self) -> str:
        if self.species is None:
            known_unknown = "UnknownSample"
        else:
            known_unknown = "KnownSample"
        if self.classification is None:
            classification = ""
        else:
            classification = f", classification={self.classification!r}"
        return (
            f"{known_unknown}("
            f"sepal_length={self.sepal_length}, "
            f"sepal_width={self.sepal_width}, "
            f"petal_length={self.petal_length}, "
            f"petal_width={self.petal_width}, "
            f"species={self.species!r}"
            f"{classification}"
            f")"
        )
```

Metoda `__repr__()` odzwierciedla stosunkowo złożony wewnętrzny stan obiektów klasy `Sample`. Stany niejawnie sugerowane przez obecność (lub brak) nazwy gatunku oraz obecność (lub brak) klasyfikacji prowadzą do nieznacznych zmian w jej działaniu. Jak na razie wszelkie zmiany w zachowaniu obiektu ograniczają się właśnie do metody `__repr__()`, używanej do wyświetlania bieżącego stanu obiektu.

To dość ważne, że zmiany stanu powodują (nieznaczne) zmiany zachowania obiektów.

W klasie `Sample` musimy także zaimplementować dwie metody związane z wykorzystaniem obiektów tej klasy. Przedstawiliśmy je w tym fragmencie kodu:

```
    def classify(self, classification: str) -> None:
        self.classification = classification

    def matches(self) -> bool:
        return self.species == self.classification
```

Metoda `classify()` definiuje zmianę stanu próbki z niesklasyfikowanej na sklasyfikowaną. Z kolei metoda `matches()` porównuje wynik klasyfikacji z gatunkiem określonym przez botanika. Metoda ta posłuży nam do wykonywania testów.

Poniższy przykład przedstawia, jak może wyglądać taka zmiana stanu:

```
>>> from model import Sample
>>> s2 = Sample(
...     sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_width=0.2,
    ↪species='Iris-setosa')
>>> s2
KnownSample(sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_width=0.2,
↪species='Iris-setosa')
>>> s2.classification = "wrong"
>>> s2
KnownSample(sepal_length=5.1, sepal_width=3.5, petal_length=1.4, petal_width=0.2,
↪species='Iris-setosa', classification='wrong')
```

Dysponujemy już zatem działającą definicją klasy `Sample`. Metoda `__repr__()` jest stosunkowo złożona, co sugeruje możliwość wprowadzenia jakichś usprawnień.

Przydatne może być także zdefiniowanie odpowiedzialności każdej z klas. Można nadać jej formę podsumowania koncentrującego się na atrybutach i metodach, rozszerzonego o jakieś wyjaśnienia dotyczące tego, dlaczego zdecydowaliśmy się zastosować taki sposób ich połączenia.

Odpowiedzialności klasy

Która klasa jest odpowiedzialna za wykonywanie testów? Czy to klasa `TrainingData` ma wywoływać klasyfikator na rzecz każdej z instancji `KnownSample` umieszczonych w zbiorze uczącym? A może ma przekazywać ten zbiór uczący do klasy `Hyperparameter`, delegując testowanie właśnie do niej? Klasa `Hyperparameter` jest odpowiedzialna za parametr *k* i dysponuje algorytmem znajdowania *k* najbliższych sąsiadów; wydaje się zatem sensowne, by to właśnie jej używać do wykonywania testów z wykorzystaniem konkretnej wartości *k* oraz przekazanej listy instancji `KnownSample`.

Wydaje się także jasne, że klasa `TrainingData` jest dobrym miejscem do przechowywania informacji o wynikach prób przeprowadzanych z różnymi instancjami `Hyperparameter`. Oznacza to, że klasa `TrainingData` może określać, która z instancji `Hyperparameter` zawiera wartość *k* pozwalającą na klasyfikowanie irysów z największą dokładnością.

Mamy tu do czynienia z wieloma, powiązanymi ze sobą, zmianami stanu. W tym przypadku obie klasy, `Hyperparameter` i `TrainingData`, będą wykonywały część pracy. System, pojmowany całościowo, będzie zmieniał stan, kiedy będą zmieniały stan jego poszczególne elementy. Taki sposób działania jest czasami określany jako **zachowanie emergentne** (ang. *emergent behavior*). Zamiast pisać gigantyczną klasę, która robi wiele rzeczy, będziemy pisać mniejsze klasy, które współdziałają ze sobą, by osiągnąć zamierzone cele.

Metody `test()` klasy `TrainingData` nie pokazywaliśmy wcześniej na diagramach UML. Dodaliśmy analogiczną metodę `test()` do klasy `Hyperparameter`, jednak nie wyglądało na to, by trzeba było ją definiować także w klasie `TrainingData`.

Następny fragment kodu przedstawia początek definicji klasy `Hyperparameter`:

```
class Hyperparameter:
    """Wartość hiperparametru i ogólna jakość klasyfikacji"""

    def __init__(self, k: int, training: "TrainingData") -> None:
        self.k = k
        self.data: weakref.ReferenceType["TrainingData"] = weakref.ref(training)
        self.quality: float
```

Zwróć uwagę na zapis podpowiedzi typów dla klas, które jeszcze nie zostały zdefiniowane. W przypadku gdy klasa jest definiowana w dalszej części pliku, wszystkie odwołania do takiej klasy są *odwołaniami do przodu*. Takie odwołania do jeszcze niezdefiniowanej klasy `TrainingData` są zapisywane jako łańcuchy znaków, a nie jako zwyczajna nazwa klasy. Kiedy narzędzie *mypy* analizuje taki kod, przekształca takie łańcuchy na prawidłowe nazwy klas.

Proces testowania jest zdefiniowany jako metoda `test()` przedstawiona w poniższym fragmencie kodu:

```
    def test(self) -> None:
        """Wykonuje cały zestaw testów"""
        training_data: Optional["TrainingData"] = self.data()
        if not training_data:
            raise RuntimeError("Uszkodzona słaba referencja")
        pass_count, fail_count = 0, 0
        for sample in self.data.testing:
            sample.classification = self.classify(sample)
            if sample.matches():
                pass_count += 1
            else:
                fail_count += 1
        self.quality = pass_count / (pass_count + fail_count)
```

Zaczynamy od wyznaczenia słabej referencji do zbioru uczącego. W razie wystąpienia jakiegoś problemu, instrukcja ta zgłosi wyjątek. Dla każdej próbki należącej do zbioru testowego wykonujemy klasyfikację i ustawiamy wartość parametru `classification` danej próbki. Metoda `matches()` określa, czy klasyfikacja wykonana przez model odpowiada znanym gatunkom. I w końcu określamy ogólną jakość, dzieląc liczbę poprawnie wykonanych klasyfikacji przez liczbę wszystkich wykonanych testów.

Samej metody odpowiedzialnej za wykonanie klasyfikacji nie będziemy przedstawiać w tym rozdziale, zajmiemy się nią dopiero w rozdziale 10. „Wzorzec Iterator". W dalszej części tego rozdziału przedstawimy klasę `TrainingData`, łączącą w sobie wszystkie przedstawione wcześniej elementy.

Klasa TrainingData

Klasa `TrainingData` zawiera listy z obiektami dwóch klas pochodnych klasy `Sample`. Klasy `KnownSample` oraz `UnknownSample` możemy zaimplementować jako rozszerzenia wspólnej klasy bazowej — `Sample`.

Temu rozwiązaniu przyjrzymy się dokładniej, i to z kilku różnych punktów widzenia, w rozdziale 7. Klasa `TrainingData` zawiera także listę instancji `Hyperparameter`. W tej klasie możemy używać prostych, bezpośrednich referencji do wcześniej zdefiniowanych klas.

Klasa `TrainingData` definiuje również dwie metody służące do rozpoczynania przetwarzania:

- Metoda `load()` wczytuje nieprzetworzone dane i dzieli je na zbiór uczący i testowy. Oba te zbiory będą tworzyć instancje `KnownSample`, jednak każdy z nich będzie służył do innych celów. Zbiór uczący służy do zastosowania algorytmu *k*-NN, natomiast zbiór testowy do określenia, jakie efekty daje użycie konkretnej wartości hiperparametru *k*.
- Metoda `test()` używa obiektu `Hyperparameter`, wykonuje test i zapisuje wynik.

Jeśli cofniemy się do rozdziału 1., znajdziemy w nim trzy opowieści użytkowników: *Dostarcza danych uczących*, *Ustawia parametry i testuje klasyfikator* oraz *Prosi o przeprowadzenie klasyfikacji*. Zapewne przyda się nam metoda, która będzie wykonywać klasyfikację, dysponując przekazaną instancją klasy `Hyperparameter`. To z kolei zmusi nas do dodania do klasy `TrainingData` metody `classify()`. Także tym razem na początku prac nie było żadnego jasno sprecyzowanego wymogu, by dodawać tę metodę, jednak teraz wydaje się, że jej implementacja będzie dobrym pomysłem.

Poniżej zamieściliśmy początek definicji klasy `TrainingData`:

```
class TrainingData:
    """Zbiór danych uczących oraz testowych wraz z metodami do ich wczytania
       oraz wykonania testu"""

    def __init__(self, name: str) -> None:
        self.name = name
        self.uploaded: datetime.datetime
        self.tested: datetime.datetime
        self.training: list[Sample] = []
        self.testing: list[Sample] = []
        self.tuning: list[Hyperparameter] = []
```

Do klasy dodaliśmy szereg atrybutów pozwalających na śledzenie wprowadzanych w niej zmian. Takimi informacjami historycznymi są np. czas wczytania danych oraz czas wykonania testu. Atrybuty `training`, `testing` oraz `tuning` zawierają listy obiektów `Sample` i `Hyperparameter`.

Nie będziemy implementować metod służących do ustawiania tych wszystkich atrybutów. W końcu używamy Pythona, a w złożonych aplikacjach bezpośredni dostęp do atrybutów stanowi ogromne ułatwienie. Odpowiedzialności są hermetyzowane w klasach, jednak — ogólnie rzecz biorąc — nie piszemy zbyt wielu metod, które odczytują wartości atrybutów oraz je ustawiają (określanych potocznie jako metody typu *getter* i *setter*).

W rozdziale 5. „Kiedy korzystać z programowania obiektowego" przedstawiliśmy kilka sprytnych technik, takich jak definiowanie właściwości, zapewniających dodatkowe możliwości obsługi atrybutów.

Metoda `load()` została zaprojektowana tak, by przetwarzać dane przekazywane przez inny obiekt. Moglibyśmy ją zaprojektować w taki sposób, by otwierała i wczytywała dane z pliku, jednak oznaczałoby to powiązanie jej z konkretnym formatem pliku i logicznym układem danych. Wydaje się, że bardziej sensowne będzie odseparowanie szczegółów związanych z formatem pliku od szczegółów związanych z zarządzaniem zbiorem danych używanych do uczenia i testowania. Szczegółowe informacje dotyczące wczytywania danych oraz ich walidacji podaliśmy w rozdziale 5. Dodatkowe rozważania na temat zagadnień związanych z formatem pliku zamieściliśmy także w rozdziale 9. „Łańcuchy, serializacja i ścieżki do plików".

Jak na razie do wczytywania danych będziemy używali metody o następujące strukturze:

```
def load(self, raw_data_source: Iterable[dict[str, str]]) -> None:
    """Wczytuje i dzieli nieprzetworzone dane na zbiory uczący i testowy"""
    for n, row in enumerate(raw_data_source):
        ... filtrowanie i pobranie podzbiorów danych (patrz rozdział 6.)
        ... utworzenie podzbiorów self.training oraz self.testing
    self.uploaded = datetime.datetime.now(tz=datetime.timezone.utc)
```

Konkretne czynności, jakie będziemy musieli wykonać, będą zależeć od źródła danych. Właściwości tego źródła danych opiszemy przy użyciu podpowiedzi typu o postaci `Iterable[dict[str, str]]`. Typ `Iterable` określa, że wyniki metody będą mogły być używane w instrukcji `for` lub w metodzie `list()`. Takie możliwości zapewniają kolekcje takie jak listy i pliki, jak również funkcje generatorów, które zostały dokładniej opisane w rozdziale 10. „Wzorzec Iterator".

Wynikami zwracanymi przez ten interator muszą być słowniki, które kojarzą łańcuch znaków z innym łańcuchem. To bardzo ogólna struktura, która pozwala nam zażądać użycia słownika o następującej postaci:

```
{
    "sepal_length": 5.1,
    "sepal_width": 3.5,
    "petal_length": 1.4,
    "petal_width": 0.2,
    "species": "Iris-setosa"
}
```

Ta wymagana struktura wydaje się na tyle elastyczna, że możemy przygotować obiekt, który będzie ją generował. Tym zagadnieniem ponownie zajmiemy się w rozdziale 9.

Pozostałe metody przekazują większość z wykonywanych operacji do klasy `Hyperparameter`. Zamiast samodzielnie wykonywać klasyfikację, ta klasa używa do tego celu innej klasy:

```
def test(self, parameter: Hyperparameter) -> None:
    """Testuje przekazaną wartość Hyperparameter"""
    parameter.test()
    self.tuning.append(parameter)
    self.tested = datetime.datetime.now(tz=datetime.timezone.utc)

def classify(self, parameter: Hyperparameter, sample: Sample) -> Sample:
    """Dokonuje klasyfikacji przekazanego obiektu Sample"""
    classification = parameter.classify(sample)
    sample.classify(classification)
    return sample
```

Do obu tych metod, jako parametr, przekazywany jest obiekt `Hyperparameter`. W przypadku testów jest to sensowne rozwiązanie, ponieważ każdy test powinien mieć unikalną wartość tego hiperparametru. Z kolei w przypadku klasyfikacji należy użyć obiektu `Hyperparameter` z wartością parametru zapewniającą najlepsze wyniki.

W ramach tej części naszego studium przypadku napisaliśmy definicje klas `Sample`, `KnownSample`, `TrainingData` oraz `Hyperparameter`. Te klasy obejmują pewien fragment całej aplikacji. Oczywiście nie jest ona jeszcze gotowa ani kompletna — pominęliśmy na razie pewne kluczowe algorytmy. Jednak prace warto zaczynać do tego, co jest oczywiste, zidentyfikować zachowania, zmiany stanu oraz zdefiniować odpowiedzialności. W następnym etapie prac można uzupełniać szczegóły i bazować na tej przygotowanej wcześniej podstawie.

Przypomnij sobie

Przypomnij sobie niektóre kluczowe zagadnienia zamieszczone w tym rozdziale:

- Python udostępnia opcjonalne podpowiedzi typów opisujące powiązania pomiędzy obiektami oraz określające, jakie parametry powinny mieć metody i funkcje.
- Do tworzenia klas w języku Python używamy instrukcji `class`. Atrybuty klas należy inicjować w specjalnej metodzie o nazwie `__init__()`.
- Moduły i pakiety są wyższymi poziomami hierarchii grupowania klas.
- Zawartość modułów należy uważnie zaplanować. Choć ogólna zasada mówi, że „płaska struktura jest lepsza od zagnieżdżonej”, to jednak mogą się pojawić przypadki, w których zastosowanie zagnieżdżonej struktury pakietów będzie pomocne.
- W Pythonie nie występuje pojęcie danych „prywatnych”. Często mówimy: „Wszyscy tu jesteśmy dorośli” — możemy przeglądać kod źródłowy, więc prywatne deklaracje nie są szczególnie przydatne. W żadnym stopniu nie zmienia to naszego projektu, jedynie eliminuje konieczność zapisywana w kodzie paru słów kluczowych.

- Dodatkowe pakiety, napisane przez innych twórców, można instalować przy użyciu narzędzia PIP. Środowiska wirtualne można tworzyć np. przy użyciu narzędzia `venv`.

Ćwiczenia

Napisz jakiś kod, używając w nim klas i obiektów. Chodzi o to, byś zastosował zasady i składnię, które poznałeś w tym rozdziale, i upewnił się, że rozumiesz opisane w nim zagadnienia. Jeśli już wcześniej pracowałeś nad jakimś projektem tworzonym w języku Python, wróć do niego i zastanów, czy są w nim jakieś obiekty, które mógłbyś stworzyć i dodać do nich jakieś właściwości i metody. Jeśli projekt jest duży, spróbuj podzielić go na kilka modułów, albo nawet pakietów, i pobawić się ze składnią instrukcji. Choć „prosty" skrypt może się trochę wydłużyć po dodaniu do niego klas i obiektów, to jednak powinien także zyskać na elastyczności i zapewniać większe możliwości rozszerzania.

Jeśli nie dysponujesz żadnym projektem do przerobienia na wersję obiektową, to utwórz nowy. Nie musi to być coś, co chciałbyś doprowadzić do końca, a jedynie miejsce, by poćwiczyć podstawowe elementy stosowane w projektach. Nie musisz nawet wszystkiego do końca implementować; czasami proste wywołanie `print("ta metoda będzie coś robić")` w zupełności wystarczy do przygotowania ogólnego projektu aplikacji. Takie postępowanie nazywamy **zstępującym** (ang. *top-down design*) i polega ono na tym, że najpierw opracowujemy kolejne interakcje i opisujemy, jak powinny działać, a dopiero potem implementujemy faktycznie wykonywane przez nie czynności. Podejście odwrotne, nazywane także **wstępującym** (ang. *bottom-up design*), polega na implementowaniu szczegółów i późniejszym łączeniu ich w całość. Oba te podejścia do tworzenia oprogramowania są przydatne w określonych sytuacjach, jednak podejście zstępujące znacznie lepiej nadaje się do poznawania i uczenia zasad programowania obiektowego.

Jeśli masz problem ze zrozumieniem prezentowanych tu koncepcji, spróbuj napisać aplikację listy rzeczy do zrobienia. Może ona rejestrować codzienne zadania do wykonania. Jej elementy mają obsługiwać zmianę stanu z „oczekujące" na „wykonane". Możesz także zastanowić się nad dodaniem stanu pośredniego: „rozpoczęte (ale jeszcze nie skończone)".

Następnie spróbuj zaprojektować coś większego. Interesującym wyzwaniem mogłoby być zamodelowanie gry w karty. Karty mają niewiele cech, jednak reguły mogą być bardzo różne. Klasa reprezentująca rękę gracza ma interesujące zmiany stanu związane z oddawaniem i zagrywaniem kart. Wybierz konkretną grę i napisz klasy reprezentujące karty, rękę gracza oraz samą rozgrywkę. (Nie zawracaj sobie głowy implementowaniem strategii wygrywającej; to może być zbyt trudne).

Gra *cribbage* ma interesujące zmiany stanu: każdy z dwóch graczy przekazuje po dwie karty, które tworzą rodzaj trzeciej ręki, określanej jako *crib*. Nie zapomnij poeksperymentować z pakietami i instrukcjami importu z innych modułów. Zaimplementuj funkcje w różnych modułach, a następnie próbuj importować je z innych modułów i pakietów. Staraj się używać

importowania bezwzględnego oraz względnego. Przyjrzyj się różnicom pomiędzy tymi dwoma sposobami importowania i spróbuj wyobrazić sobie scenariusze, w których mógłbyś chcieć wykorzystać każdy z nich.

Podsumowanie

W tym rozdziale pokazaliśmy, jak łatwo można w Pythonie tworzyć klasy i dodawać do nich właściwości i metody. W przeciwieństwie do wielu innych języków programowania Python wprowadza rozróżnienie pomiędzy konstruktorem oraz inicjalizatorem obiektu. Pokazaliśmy, że Python w dość luźny sposób podchodzi do kontroli dostępu. Dostępnych jest wiele różnych poziomów zasięgu, w tym zasięg pakietów, modułów, klas oraz funkcji. Wyjaśniliśmy różnice pomiędzy importem bezwzględnym i względnym, a na końcu pokazaliśmy, jak zarządzać pakietami innych twórców, które nie są domyślnie dostarczane wraz z Pythonem.

W następnym rozdziale dowiesz się, jak współdzielić implementację przy wykorzystaniu dziedziczenia.

Skorowidz

C

D

F

G

H

I

J

K

L

Ł

M

P

R

S

Ś

T

U

W

X

Y

Z

Chcesz tworzyć solidny kod? Zorientuj się na obiekty!

Python zasłużenie cieszy się ogromną popularnością. To język, który może służyć do wielu celów, szczególnie do szybkiego tworzenia niewielkich, wyspecjalizowanych programów. Projektowanie bardziej rozbudowanego, wyrafinowanego oprogramowania też jest możliwe, wymaga jednak zdobycia kilku ważnych umiejętności. Bardzo dobrym pomysłem okazuje się zastosowanie w programowaniu w Pythonie podejścia zorientowanego obiektowo. Tak tworzony kod jest czytelny, solidny, łatwy w rozbudowie i o wiele efektywniejszy w działaniu.

Oto przyjazny przewodnik dla programistów Pythona, wyczerpująco wyjaśniający wiele zagadnień programowania obiektowego, takich jak dziedziczenie, kompozycja, polimorfizm, tworzenie klas i struktur danych. W książce szczegółowo omówiono obsługę wyjątków, testowanie kodu i zastosowanie technik programowania funkcyjnego. Opisano też dwa potężne zautomatyzowane systemy testowe: unittest i pytest. Zaprezentowano tematykę utrzymania złożonego oprogramowania napisanego w sposób zorientowany obiektowo, a także podano wskazówki odnoszące się do jego rozbudowy. Ważną częścią przewodnika jest omówienie zasad programowania współbieżnego we współczesnym Pythonie. Co ważne, poszczególne zagadnienia zostały zilustrowane diagramami UML, czytelnymi przykładami i studiami przypadków.

W książce między innymi:

- kiedy korzystać z technik obiektowych
- implementacja obiektów i mechanizmu dziedziczenia w Pythonie
- stosowanie wyjątków, a także tworzenie testów jednostkowych i integracyjnych
- ważniejsze wzorce projektowe i ich implementacja w Pythonie
- statyczne typowanie dynamicznego kodu
- programowanie współbieżne przy użyciu asyncio

Steven F. Lott

programuje w Pythonie od ponad trzydziestu lat, używa tego języka do tworzenia różnego rodzaju narzędzi i aplikacji. Napisał kilka książek o programowaniu. Uważa się za koczownika i mieszka na łodzi.

Dusty Phillips

jest kanadyjskim programistą i autorem książek o programowaniu. Pracował dla rządów, startupów i sieci społecznościowych. Obecnie zajmuje się pisaniem powieści fantastycznych.

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 978-83-283-8949-6

9 788328 389496

Cena: 149,00 zł

Packt