O'REILLY®

Wydanie III

Python

Nowoczesne programowanie w prostych krokach

Helion

Bill Lubanovic

Tytuł oryginału: Introducing Python: Modern Computing in Simple Packages, 3rd Edition

Tłumaczenie: Anna Mizerska, z wykorzystaniem fragmentów poprzedniego wydania
w przekładzie Andrzeja Watraka

ISBN: 978-83-289-4091-8

Authorized Polish translation of the English edition of *Introducing Python, 3E*
ISBN 9781098174408 © 2025 Bill Lubanovic

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/pythn3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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

Printed in Poland.

Spis treści

ROZDZIAŁ 11.
Obiekty

Żaden obiekt nie jest tajemniczy. Tajemnicą jest twoje oko.
— Elizabeth Bowen

Weź obiekt. Zrób coś z nim. Potem zrób coś innego.
— Jasper Johns

Jak już kilkakrotnie wspomniałem w tej książce, wszystko w Pythonie, od liczby po funkcję, jest obiektem. Jednak większość operacji wykonywanych na obiektach jest ukryta za specjalną składnią. Na przykład zapis `num = 7` powoduje utworzenie obiektu reprezentującego liczbę całkowitą i zawierającego wartość 7, a następnie przypisanie go zmiennej o nazwie `num`. W szczegóły obiektów zagląda się jedynie wtedy, gdy trzeba tworzyć własne obiekty lub modyfikować działanie istniejących. W tym rozdziale dowiesz się, jak się robi jedno i drugie.

Czym jest obiekt?

Obiekt jest niestandardową strukturą zawierającą zarówno dane (zmienne, zwane **atrybutami**) jak i kod (funkcje, zwane **metodami**). Obiekt reprezentuje instancję konkretnej rzeczy, a jego metody określają interakcje z innymi rzeczami. Obiekty można sobie wyobrazić jako rzeczowniki, a ich metody jako czasowniki.

Na przykład obiekt reprezentujący liczbę całkowitą 7 zawiera m.in. metody wykonujące operacje dodawania i mnożenia, które poznałeś w rozdziale 3. Liczba 8 jest innym obiektem. Oznacza to, że w języku Python jest zdefiniowana klasa liczb całkowitych, do której należą liczby 7 i 8. Ciągi `'kot'` i `'pies'` również są obiektami i mają metody opisane w rozdziale 4., m.in. `capitalize()` i `replace()`.

Program może zawierać wiele obiektów (często zwanych **instancjami**), w odróżnieniu od modułów (które będziemy omawiać w rozdziale 12.). Każdy obiekt może mieć inne atrybuty. Obiekty można porównać do super struktur danych z kodem w środku.

Proste obiekty

Zacznijmy od prostych klas. Dyskusję na temat **dziedziczenia** klas odłóżmy na później.

Definiowanie klasy

Aby utworzyć obiekt, należy najpierw zdefiniować **klasę** zawierającą informacje, co obiekt ma zawierać.

W rozdziale 2. porównałem obiekt do pudełka. Z kolei *klasę* można porównać do wzoru pudełka. Na przykład Python zawiera wbudowaną klasę, na podstawie której tworzy się ciągi znaków, takie jak `'kot'`, czy `'pies'`. Podobnie jest z danymi innych typów — listami, słownikami itp. Aby utworzyć własny, niestandardowy obiekt, najpierw trzeba zdefiniować klasę za pomocą instrukcji `class`. Przeanalizujmy kilka przykładów.

Załóżmy, że trzeba utworzyć obiekty reprezentujące koty. Każdy obiekt będzie reprezentował jednego zwierzaka. Najpierw należy zdefiniować klasę o nazwie `Cat`. W opisanych niżej przykładach wykorzystane są różne wersje tej klasy, od najprostszej do bardziej złożonej, z którą można zrobić coś pożytecznego.

W przykładach stosowana jest konwencja nazewnictwa opisana w dokumentacji PEP-8 (*https://oreil.ly/gAJOF*).

Nasza pierwsza klasa będzie najprostszą z możliwych, pustą klasą:

```
>>> class Cat():
...         pass
```

Klasę tę można również zdefiniować w następujący sposób:

```
>>> class Cat:
...         pass
```

Podobnie jak w przypadku funkcji, aby zdefiniować pustą klasę, należy użyć instrukcji `pass`.

Tak zdefiniowana klasa stanowi niezbędne minimum do utworzenia obiektu. Obiekt tworzy się przez wywołanie klasy tak jak funkcji:

```
>>> a_cat = Cat()
>>> another_cat = Cat()
```

W tym przykładzie zostały utworzone na podstawie klasy `Cat` dwa osobne obiekty, przypisane następnie zmiennym `a_cat` i `another_cat`. Ponieważ klasa nie zawiera żadnego kodu, obiekty zajmują jedynie miejsce w pamięci i nic nie robią.

Chociaż coś jednak mogą robić.

Atrybuty

Atrybut to zmienna zawarta w klasie lub obiekcie. Atrybutom można przypisywać wartości w definicji klasy lub w obiekcie po jego utworzeniu. Atrybut może zawierać dowolny inny obiekt. Utwórzmy ponownie dwa obiekty reprezentujące koty:

```
>>> class Cat:
...         pass
...
>>> a_cat = Cat()
>>> a_cat
<__main__.Cat object at 0x100cd1da0>
>>> another_cat = Cat()
>>> another_cat
<__main__.Cat object at 0x100cd1e48>
```

Definicja klasy `Cat` nie określa, jak ma być wyświetlany obiekt utworzony na jej podstawie. Na ekranie pojawia się informacja `<__main__.Cat object at 0x100cd1da0>`. W podrozdziale „Magiczne metody” dowiesz się, jak zmienić to domyślne działanie.

Teraz pierwszemu obiektowi przypiszmy kilka atrybutów:

```
>>> a_cat.age = 3
>>> a_cat.name = "Filemon"
>>> a_cat.nemesis = another_cat
```

Czy można odwoływać się do tak utworzonego atrybutu? Oczywiście, że tak:

```
>>> a_cat.age
3
>>> a_cat.name
'Filemon'
>>> a_cat.nemesis
<__main__.Cat object at 0x100cd1e48>
```

Ponieważ `nemesis` jest atrybutem odwołującym się do innego obiektu typu `Cat`, można użyć nazwy `a_cat.nemesis`. Obiekt ten jednak nie ma jeszcze atrybutu `name`:

```
>>> a_cat.nemesis.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Cat' object has no attribute 'name'
```

Nadajmy nazwę drugiemu obiektowi:

```
>>> a_cat.nemesis.name = "Bonifacy"
>>> a_cat.nemesis.name
'Bonifacy'
```

Nawet najprostszy obiekt, taki jak ten, może mieć wiele atrybutów. Zatem różne wartości można przechowywać za pomocą obiektu, a nie na przykład listy czy słownika.

Termin **atrybut** zazwyczaj oznacza atrybut obiektu. Istnieją również atrybuty klasy. Różnicę pomiędzy obydwoma pojęciami poznasz w podrozdziale „Atrybuty klas i obiektów”.

Metody

Metoda to funkcja zawarta w klasie lub obiekcie. Wygląda tak samo jak zwykła funkcja, ale można ją wykorzystywać w specjalny sposób opisany w podrozdziałach „Dostęp do atrybutów za pośrednictwem właściwości" i „Rodzaje metod".

Inicjacja obiektu

Instancja obiektu jest najpierw tworzona za pomocą metody __new__(), ale to się dzieje w tle i zwykle nawet tego nie widzisz. Następnym krokiem jest **inicjalizacja**, obsługiwana przez `__init__()`:

```
>>> class Cat:
...         def __init__(self):
...             pass
```

Z tego rodzaju kodem będziesz się spotykał w definicjach rzeczywistych klas. Przyznaję, że nazwy `__init__()` i `self` wyglądają dziwnie. Pierwsza z nich oznacza specjalną metodę inicjującą obiekt utworzony na podstawie danej klasy[1]. Argument `self` zawiera zaś odwołanie do bieżącego obiektu.

`self` musi być pierwszym argumentem metody `__init__()`. Choć nie jest to zarezerwowana nazwa, przyjęło się właśnie taką stosować. Jeżeli będziesz przestrzegał tej konwencji, inny programista (a być może nawet Ty sam) nie będzie musiał się domyślać, co oznacza ten argument.

Jednak nowa definicja klasy `Cat` też nie sprawia, że utworzony na jej podstawie obiekt cokolwiek robi. Trzecia próba polega na utworzeniu prostego obiektu, w którym jego atrybutowi jest przypisywana wartość. Tym razem metoda inicjująca obiekt zawiera argument `name`:

```
>>> class Cat():
...         def __init__(self, name):
...             self.name = name
...
>>>
```

Można więc utworzyć obiekt przez umieszczenie w argumencie ciągu znaków:

```
>>> furball = Cat('Filemon')
```

W powyższym wierszu kodu wykonywane są następujące operacje:

- wyszukanie definicji klasy `Cat`,
- utworzenie nowego obiektu (instancji),
- wywołanie metody `__init__()` z argumentem `self`, zawierającym nowo utworzony obiekt, oraz argumentem `name`, zawierającym ciąg znaków `'Filemon'`,
- przypisanie atrybutowi wartości argumentu `name`,
- zwrócenie nowego obiektu,
- zapisanie obiektu w zmiennej `furball`.

[1] Z podwójnymi znakami podkreślenia w nazwach będziesz się spotykał bardzo często.

Utworzony w ten sposób obiekt jest taki sam jak każdy inny. Można go użyć jako elementu listy, krotki, słownika lub zbioru, jak również umieszczać w argumentach funkcji i zwracanych przez nie wynikach.

Co się dzieje z wartością argumentu `name`? Jest zapisywana w atrybucie obiektu, którego wartość można odczytać wprost:

```
>>> print('Wartość argumentu:', furball.name)
'Wartość argumentu: Filemon
```

Pamiętaj, że wewnątrz definicji klasy do atrybutu należy się odwoływać z użyciem składni `self.name`. Po utworzeniu obiektu i przypisaniu go zmiennej, na przykład `furball`, należy w tym celu stosować zapis `furball.name`.

Metody `__init__()` nie trzeba definiować w każdej klasie. Wykorzystuje się ją do wykonywania operacji niezbędnych do późniejszego odróżniania obiektów utworzonych na podstawie tej samej klasy. Metoda ta nie jest odpowiednikiem tzw. konstruktora klasy, stosowanego w innych językach. W Pythonie obiekt jest tworzony automatycznie, a metoda `__init__()` służy do *inicjowania* go.

Na podstawie jednej klasy można tworzyć wiele obiektów. Pamiętaj, że w Pythonie dane implementuje się jako obiekty, więc klasa również jest obiektem. W programie może jednak istnieć tylko jeden obiekt klasy. W naszym przypadku można zdefiniować tylko jedną klasę o nazwie `Cat`.

Dziedziczenie klas

Często podczas pisania kodu okazuje się, że istniejąca klasa wykonuje większość potrzebnych operacji, ale nie wszystkie. Co można wtedy zrobić?

Można zmodyfikować istniejącą klasę, ale w ten sposób bardziej się ją komplikuje, a ponadto można w ten sposób zakłócić jej dotychczasowe, poprawne działanie. Można również zdefiniować nową klasę, skopiować kod istniejącej klasy i scalić go z nowym. Oznaczałoby to jednak więcej kodu do utrzymywania. Poza tym, w miarę upływu czasu i wprowadzania kolejnych zmian, pomiędzy obiema częściami kodu, które powinny być takie same, mogłyby się zacząć pojawiać różnice.

Innym rozwiązaniem jest **odziedziczenie klasy**, czyli utworzenie nowej klasy *na podstawie* istniejącej, a następnie wprowadzenie w niej pewnych zmian i dodatków. Jest to dobry sposób na wykorzystanie istniejącego kodu. Dzięki dziedziczeniu można w nowej klasie wykorzystywać cały kod istniejącej klasy bez konieczności powielania go.

Obecnie przyjmuje się, że w większości sytuacji lepsza jest **kompozycja** niż dziedziczenie. Wkrótce powiem o tym więcej, ale i tak warto wiedzieć, jak działa dziedziczenie.

Dziedziczenie klasy nadrzędnej

W nowej klasie można utworzyć nowy kod lub zmienić istniejący. Wprowadzone zmiany mają priorytet w stosunku do istniejącej implementacji. Oryginalna klasa jest nazywana klasą **rodzicielską**, **bazową** lub **nadklasą**, a nowa klasa **potomną**, **pochodną** lub **podklasą**. Powyższe terminy są w programowaniu obiektowym stosowane zamiennie.

Odziedziczmy więc coś. W poniższym przykładzie zdefiniowana jest pusta klasa o nazwie `Car` oraz podklasa `Yugo`[2]. Podklasę definiuje się podobnie jak nadklasę. Różnica polega jedynie na tym, że w definicji podklasy umieszcza się w nawiasach nazwę nadklasy (w tym przykładzie `class Yugo(Car)`):

```
>>> class Car():
...         pass
...
>>> class Yugo(Car):
...         pass
...
```

Za pomocą funkcji `issubclass()` można sprawdzić, czy dana klasa pochodzi od innej:

```
>>> issubclass(Yugo, Car)
True
```

Utwórzmy teraz obiekty na podstawie obu klas:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Klasa pochodna jest wyspecjalizowaną wersją klasy rodzicielskiej. Zmienna `give_me_a_yugo` zawiera instancję klasy `Yugo`, dziedziczącą cechy klasy `Car`. Zdefiniujmy teraz na nowo nasze klasy tak, aby coś robiły:

```
>>> class Car():
...         def exclaim(self):
...             print("Jestem klasą Car!")
...
>>> class Yugo(Car):
...         pass
...
```

Utwórzmy również obiekty na podstawie obu klas i wywołajmy metodę `exclaim()` każdego z nich:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
Jestem klasą Car!
>>> give_me_a_yugo.exclaim()
Jestem klasą Car!
```

[2] Niedrogi i niezbyt dobry samochód z lat 80.

Klasa `Yugo`, nie robiąc nic szczególnego, odziedziczyła po klasie `Car` metodę `exclaim()`. Dlatego „mówi”, że *jest* klasą `Car` (ogólnie prawdziwe, ale może wymagać specjalizacji), co może wprowadzać zamieszanie w identyfikacji obiektów. Sprawdźmy, co można z tym zrobić.

Dziedziczenie jest bardzo przydatną, ale często nadużywaną funkcjonalnością. Jak się okazuje, jej nadmierne stosowanie skutkuje uzyskaniem kodu, który trudno jest utrzymywać. Dlatego zalecane jest wykorzystywanie technik takich jak agregowanie i komponowanie klas, opisanych w dalszej części rozdziału, w podrozdziale „Agregowanie i komponowanie klas”.

Nadpisywanie metod

Jak już wiesz, nowa klasa dziedziczy wszystkie cechy klasy nadrzędnej. Teraz dowiesz się, jak zastępować, czyli nadpisywać, metody klasy nadrzędnej. Klasa `Yugo` powinna odróżniać się w jakiś sposób od klasy `Car`. Gdyby była taka sama, jaki byłby sens jej tworzenia? Zmieńmy więc metodę `exclaim()` klasy `Yugo`:

```
>>> class Car():
...         def exclaim(self):
...             print("Jestem klasą Car!")
...
>>> class Yugo(Car):
...         def exclaim(self):
...             print("Jestem klasą Yugo dziedziczącą cechy klasy Car!")
...
```

Następnie utwórzmy dwa obiekty na podstawie obu klas:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Jak działają ich metody?

```
>>> give_me_a_car.exclaim()
Jestem klasą Car!
>>> give_me_a_yugo.exclaim()
Jestem klasą Yugo dziedziczącą cechy klasy Car!
```

W powyższym przykładzie została nadpisana metoda `exclaim()`. Nadpisywać można wszystkie metody, włącznie z `__init__()`. Poniżej przedstawiony jest inny przykład, w którym użyta jest klasa `Person`. Zdefiniujmy na jej podstawie dwie podklasy: `MDPerson` i `JDPerson`:

```
>>> class Person():
...         def __init__(self, name):
...             self.name = name
...
>>> class MDPerson(Person):
...         def __init__(self, name):
...             self.name = "Dr " + name
...
>>> class JDPerson(Person):
...         def __init__(self, name):
...             self.name = "Mec. " + name
...
```

W obu klasach metody inicjujące `__init__()` mają takie same argumenty jak analogiczna metoda klasy `Person`, ale w każdej z nich wartość argumentu `name` jest zapamiętywana w inny sposób:

```
>>> person = Person('Nowak')
>>> doctor = MDPerson('Nowak')
>>> lawyer = JDPerson('Nowak')
>>> print(person.name)
Nowak
>>> print(doctor.name)
Dr Nowak
>>> print(lawyer.name)
Mec. Nowak
```

Dodawanie metod

Klasa pochodna może *zawierać* metody, których nie ma klasa rodzicielska. Wróćmy do klas `Car` i `Yugo` i w tej drugiej zdefiniujmy metodę `need_a_push()`:

```
>>> class Car():
...         def exclaim(self):
...             print("Jestem klasą Car!")
...
>>> class Yugo(Car):
...         def exclaim(self):
...             print("Jestem klasą Yugo dziedziczącą cechy klasy Car!")
...         def need_a_push(self):
...             print("Potrzebna pomoc?")
...
```

Utwórzmy obiekty na podstawie obu klas:

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Można wywołać metodę `need_a_push()` klasy `Yugo`:

```
>>> give_me_a_yugo.need_a_push()
Potrzebna pomoc?
```

Nie można jednak tego zrobić w przypadku nadrzędnej klasy `Car`:

```
>>> give_me_a_car.need_a_push()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'need_a_push'
```

Klasa `Yugo` robi coś, czego klasa `Car` nie potrafi, zatem obie klasy różnią się od siebie.

Odwołanie do klasy nadrzędnej za pomocą metody super()

Wiesz już, jak w klasie potomnej można definiować nowe metody i nadpisywać metody klasy rodzicielskiej. Ale co robić, gdy trzeba wywołać metodę rodzicielską? Tu pojawia się funkcja `super()`. W poniższym przykładzie zdefiniowana jest nowa klasa `EmailPerson`, pochodna od klasy `Person`. Od swojej klasy rodzicielskiej różni się tym, że zawiera atrybut z adresem e-mail. Najpierw przyjrzyjmy się klasie `Person`:

```
>>> class Person():
...         def __init__(self, name):
...             self.name = name
...
```

Zwróć uwagę, że metoda `__init__()` w klasie pochodnej ma dodatkowy argument `email`:

```
>>> class EmailPerson(Person):
...         def __init__(self, name, email):
...             super().__init__(name)
...             self.email = email
```

Metoda `__init__()` zdefiniowana w klasie pochodnej zastępuje metodę o tej samej nazwie zdefiniowaną w klasie rodzicielskiej. Od tej chwili oryginalna metoda nie jest automatycznie wywoływana. Aby ją wywołać, trzeba to zrobić jawnie. Oto co się dzieje w kodzie:

- Funkcja `super()` odczytuje definicję klasy rodzicielskiej `Person`.
- Metoda `__init__()` wywołuje metodę o tej samej nazwie, zdefiniowaną w klasie `Person`. W jej argumentach umieszcza obiekt `self` i opcjonalne wartości. W tym przykładzie jedynym takim argumentem jest `name`.
- Nowy wiersz `self.email = email` odróżnia klasę `EmailPerson` od `Person`.

Idźmy dalej i utwórzmy obiekt:

```
>>> bob = EmailPerson('Robert Nowak', 'robert@adres.com')
```

Można odwoływać się zarówno do atrybutu `name`, jak i `email` obiektu:

```
>>> bob.name
'Robert Nowak'
>>> bob.email
'robert@adres.com'
```

Dlaczego nie zdefiniowaliśmy klasy pochodnej w następujący sposób?

```
>>> class EmailPerson(Person):
...         def __init__(self, name, email):
...             self.name = name
...             self.email = email
```

Można było tak zrobić, ale przeczyłoby to idei dziedziczenia klas. Tutaj za pomocą funkcji `super()` wykorzystaliśmy kod klasy `Person`, ten sam, który byłby wykorzystany w przypadku utworzenia obiektu na podstawie tej klasy. Takie podejście ma jeszcze jedną zaletę: jeżeli w przyszłości zmieni się definicja klasy `Person`, za pomocą funkcji `super()` będzie można odwoływać się do zmodyfikowanych atrybutów i metod.

Dzięki funkcji `super()` klasa pochodna, która robi różne rzeczy po swojemu, może korzystać z funkcjonalności klasy rodzicielskiej (tak jak dziecko korzysta z pomocy rodzica).

Dziedziczenie wielu klas

Poznałeś przykłady klas, które nie miały klasy rodzicielskiej lub miały tylko jedną. W rzeczywistości można definiować klasy pochodne od wielu klas.

Jeżeli w klasie pochodnej znajduje się odwołanie do niezdefiniowanego atrybutu, przeszukiwane są wszystkie klasy rodzicielskie. Co się jednak stanie, gdy dany atrybut będzie zdefiniowany w kilku klasach rodzicielskich? Która z nich będzie miała priorytet?

W Pythonie, inaczej niż w naturze, gdzie zwycięża silniejszy gen niezależnie od tego, skąd pochodzi, ważna jest **kolejność dziedziczenia**. Każda klasa ma specjalną metodę `mro()`, która zwraca listę klas przeszukiwanych pod kątem wykorzystywanych metod i atrybutów. Atrybut o podobnej nazwie, `__mro__`, jest krotką zawierającą te same informacje. Pierwsza napotkana klasa, która zawiera żądany atrybut lub metodę, wygrywa.

W poniższym przykładzie zdefiniowana jest główna klasa `Animal`, dwie pochodne od niej klasy `Horse` i `Donkey` oraz dwie następne, pochodne od powyższych:

```
>>> class Animal:
...         def says(self):
                return 'Klasa Animal'
...
>>> class Horse(Animal):
...         def says(self):
...             return 'Klasa Horse'
...
>>> class Donkey(Animal):
...         def says(self):
...             return 'Klasa Donkey'
...
>>> class Mule(Donkey, Horse):
...         pass
...
>>> class Hinny(Horse, Donkey):
...         pass
...
```

W przypadku odwołania się do metody lub atrybutu wewnątrz obiektu utworzonego na podstawie klasy `Mule`, przeszukiwane są obiekty i klasy w następującej kolejności:

1. Sam obiekt (pochodzący od klasy `Mule`).
2. Klasa, na podstawie której został utworzony obiekt (`Mule`).
3. Pierwsza klasa rodzicielska (`Donkey`).
4. Druga klasa rodzicielska (`Horse`).
5. Następna w hierarchii klasa rodzicielska (`Animal`).

Poniższe wyniki, uzyskane dla klas `Mule` i `Hinny`, różnią się kolejnością przeszukiwania klas `Horse` i `Donkey`:

```
>>> Mule.mro()
[<class '__main__.Mule'>, <class '__main__.Donkey'>,
<class '__main__.Horse'>, <class '__main__.Animal'>,
<class 'object'>]
>>> Hinny.mro()
[<class '__main__.Hinny'>, <class '__main__.Horse'>,
<class '__main__.Donkey'>, <class '__main__.Animal'>,
class 'object'>]
```

Co zatem wyświetli każdy z obiektów?

```
>>> mule = Mule()
>>> hinny = Hinny()
>>> mule.says()
'Klasa Donkey'
>>> hinny.says()
'Klasa Horse'
```

Napis jest wyświetlany przez tę klasę rodzicielską, która jest umieszczona jako pierwsza w definicji klasy pochodnej.

Gdyby klasy `Horse` i `Donkey` nie miały metod `says()`, zostałaby wywołana metoda klasy `Animal` (znajdującej się najwyżej w hierarchii) wyświetlająca napis `'Klasa Animal'`.

Domieszki

W definicji klasy można umieszczać dodatkową, pomocniczą klasę rodzicielską, która nie ma takich samych metod jak inne klasy rodzicielskie. Dzięki temu unika się opisanych w poprzednim podrozdziale niejednoznaczności w wywoływaniu metod klas rodzicielskich.

Tego rodzaju klasy są nazywane **domieszkami** (ang. *mixin*). Wykorzystuje się je do wykonywania pobocznych operacji, na przykład zapisywania informacji w dziennikach. Poniżej pokazana jest domieszka wyświetlająca w czytelny sposób atrybuty obiektu:

```
>>> class PrettyMixin():
...         def dump(self):
...             import pprint
...             pprint.pprint(vars(self))
...
>>> class Thing(PrettyMixin):
...         pass
...
>>> t = Thing()
>>> t.name = "rzecz"
>>> t.feature = "niezwykła"
>>> t.age = "100"
>>> t.dump()
{'age': '100', 'feature': 'niezwykła', 'name': 'rzecz'}
```

Argument self

Jedną z krytykowanych cech Pythona (oprócz wcięć w kodzie) jest konieczność definiowania w metodach instancji (szczególnego rodzaju metodach, które poznałeś w opisanych przykładach) pierwszego argumentu o nazwie `self`, wykorzystywanego do odwoływania się do odpowiednich metod i atrybutów. W następnym przykładzie pokażę, jak są wywoływane metody obiektu i co się w tym czasie dzieje w tle programu.

Pamiętasz klasę `Car`? Wywołajmy ponownie jej metodę `exclaim()`:

```
>>> a_car = Car()
>>> a_car.exclaim()
Jestem klasą Car!
```

Oto co się dzieje w tle programu:

- Wyszukiwana jest klasa (`Car`), na podstawie której został utworzony obiekt `a_car`.
- Obiekt `a_car` jest umieszczany w argumencie `self` metody `exclaim()` zdefiniowanej w klasie `Car`.

Dla zabawy możesz wywołać metodę w poniższy sposób i uzyskać ten sam efekt co poprzednio (z użyciem zapisu `a_car.exclaim()`):

```
>>> Car.exclaim(a_car)
Jestem klasą Car!
```

W praktyce jednak nie ma powodu, aby stosować powyższą, bardziej złożoną składnię.

Dostęp do atrybutów

W Pythonie wszystkie atrybuty i metody są publiczne (widoczne poza obiektem) i oczekuje się od nich odpowiedzialnego zachowania (tzw. **zasada świadomych dorosłych**). Porównajmy bezpośrednie odwoływanie się do nich z kilkoma innymi technikami.

Dostęp bezpośredni

Jak już wiesz, do atrybutów można odwoływać się bezpośrednio:

```
>>> class Duck:
...         def __init__(self, input_name):
...             self.name = input_name
...
>>> fowl = Duck('Donald')
>>> fowl.name
'Donald'
```

Jeśli jednak atrybut jest widoczny i niechroniony, każdy może go zmienić:

```
>>> fowl.name = 'Dziwaczka'
>>> fowl.name
'Dziwaczka'
```

W następnych dwóch podrozdziałach pokażę, jak można zapewnić atrybutom prywatność i uchronić je przed niepożądanymi zmianami.

Gettery i settery

W niektórych obiektowych językach programowania istnieją atrybuty prywatne, do których nie można odwoływać się bezpośredniego z zewnątrz. Aby można było odczytywać i przypisywać im wartości, programista musi zdefiniować specjalne metody: **gettery** i **settery**.

W Pythonie nie ma atrybutów prywatnych, ale można definiować gettery i settery, które ukrywają rzeczywiste nazwy atrybutów i w jakimś stopniu zapewniają im prywatność. (Jeszcze lepszym rozwiązaniem jest stosowanie *właściwości*, opisanych w następnym podrozdziale).

W poniższym przykładzie zdefiniowana jest klasa `Duck`, zawierająca pojedynczy, statyczny atrybut `hidden_name`. Ponieważ nie chcemy, aby inni programiści odwoływali się do tego atrybutu bezpośrednio, zdefiniujemy dwie metody: gettera `get_name()` i settera `set_name()`. Obie będą wywoływane za pośrednictwem właściwości o nazwie `name`. Ponadto obie metody zawierają funkcję `print()`, informującą, że metoda została wywołana:

```
>>> class Duck():
...         def __init__(self, input_name):
...             self.hidden_name = input_name
...         def get_name(self):
...             print('Wywołany getter')
...             return self.hidden_name
...         def set_name(self, input_name):
...             print('Wywołany setter')
...             self.hidden_name = input_name
>>> don = Duck('Donald')
>>> don.get_name()
Wywołany getter
'Donald'
>>> don.set_name('Dziwaczka')
Wywołany setter
>>> don.get_name()
Wywołany getter
'Dziwaczka'
```

Dostęp do atrybutów za pośrednictwem właściwości

W Pythonie prywatność atrybutom zapewnia się za pomocą właściwości. W tym przykładzie można to zrobić na dwa sposoby. Pierwszy polega na umieszczeniu na końcu definicji klasy `Duck` wiersza `nazwa = property(get_name, set_name)`:

```
>>> class Duck():
>>>     def __init__(self, input_name):
>>>         self.hidden_name = input_name
>>>     def get_name(self):
>>>         print('Wywołany getter')
>>>         return self.hidden_name
>>>     def set_name(self, input_name):
>>>         print('Wywołany setter')
>>>         self.hidden_name = input_name
>>>     name = property(get_name, set_name)
```

Getter i setter można wywoływać w zwykły sposób:

```
>>> don = Duck('Donald')
>>> don.get_name()
Wywołany getter
'Donald'
>>> don.set_name('Dziwaczka')
Wywołany setter
>>> don.get_name()
Wywołany getter
'Dziwaczka'
```

Jednak teraz do ukrytego argumentu można się dodatkowo odwoływać za pomocą właściwości `name`:

```
>>> don = Duck('Donald')
>>> don.name
Wywołany getter
'Donald'
>>> don.name = 'Dziwaczka'
Wywołany setter
>>> don.name
Wywołany getter
'Dziwaczka'
```

Drugi sposób polega na zamianie nazw `get_name` i `set_name` na `name` i zastosowaniu następujących dekoratorów:

- `@property` przed nazwą *gettera*,
- `@name.setter` przed nazwą *settera*.

Nowy kod wygląda jak niżej:

```
>>> class Duck():
...         def __init__(self, input_name):
...             self.hidden_name = input_name
...         @property
...         def name(self):
...             print('Wywołany getter')
...             return self.hidden_name
...         @name.setter
...         def name(self, input_name):
...             print('Wywołany setter')
...             self.hidden_name = input_name
```

Do właściwości `name` można odwoływać się tak samo jak do atrybutu:

```
>>> fowl = Duck('Donald')
>>> fowl.name
Wywołany getter
'Donald'
>>> fowl.name = 'Dziwaczka'
Wywołany setter
>>> fowl.name
Wywołany getter
'Dziwaczka'
```

Jeżeli inny programista domyśli się, że w powyższym przykładzie użyte jest odwołanie do atrybutu `hidden_name`, będzie mógł odczytywać i zapisywać w nim wartości bezpośrednio, z użyciem składni `fowl.hidden_name`. W podrozdziale „Prywatność dzięki modyfikacji nazwy” poznasz specjalny sposób ukrywania nazw atrybutów w Pythonie.

Właściwości zwracające wyliczane wartości

W poprzednich przykładach odwoływaliśmy się za pomocą właściwości `name` do pojedynczego atrybutu (`hidden_name`) w obiekcie.

Jednak właściwość może również zwracać **wyliczaną wartość**. Zdefiniujmy klasę `Circle` z atrybutem `radius` i wyliczaną właściwością `diameter`:

```
>>> class Circle():
...         def __init__(self, radius):
...             self.radius = radius
...         @property
...         def diameter(self):
...             return 2 * self.radius
...
```

Na podstawie tej klasy utwórzmy obiekt z początkową wartością atrybutu `radius`:

```
>>> c = Circle(5)
>>> c.radius
5
```

Do właściwości `diameter` można odwoływać się tak samo jak do atrybutu `radius`:

```
>>> c.diameter
10
```

Teraz będzie najciekawsza część: w każdej chwili można zmienić wartość atrybutu `radius`, a właściwość `diameter` będzie wyliczana na podstawie aktualnej wartości tego atrybutu:

```
>>> c.radius = 7
>>> c.diameter
14
```

Jeżeli nie zostanie zdefiniowany setter dla atrybutu, nie będzie można spoza klasy przypisywać mu wartości. W ten sposób definiuje się atrybuty tylko do odczytu:

```
>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

Inna zaleta odwoływania się do właściwości zamiast bezpośrednio do atrybutów polega na tym, że w przypadku zmiany definicji atrybutu trzeba jedynie dostosować kod wewnątrz klasy, bez konieczności modyfikowania wszystkich odwołań w zewnętrznym kodzie.

Prywatność dzięki modyfikacji nazwy

W opisanym wcześniej przykładzie z klasą `Duck` odwoływaliśmy się do nie całkiem ukrytego atrybutu `hidden_name`. W Pythonie przyjęta została konwencja nazewnictwa atrybutów, które nie powinny być widoczne poza definicją klasy. Zgodnie z tą konwencją nazwa atrybutu powinna rozpoczynać się od podwójnego znaku podkreślenia (`__`).

Zmieńmy teraz nazwę atrybutu `hidden_name` na `__name`, jak w poniższym przykładzie:

```
>>> class Duck():
...         def __init__(self, input_name):
...             self.__name = input_name
...         @property
...         def name(self):
...             print('Wywołany getter')
```

```
...             return self.__name
...         @name.setter
...         def name(self, input_name):
...             print('Wywołany setter')
...             self.__name = input_name
...
```

Poświęćmy chwilę na sprawdzenie, czy wszystko działa poprawnie:

```
>>> fowl = Duck('Donald')
>>> fowl.name
Wywołany getter
'Donald'
>>> fowl.name = 'Dziwaczka'
Wywołany setter
>>> fowl.name
Wywołany getter
'Dziwaczka'
```

Jest dobrze. Tym bardziej że nie można odwoływać się do atrybutu `__name`:

```
>>> fowl.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

Opisana konwencja nazewnicza nie zapewnia atrybutom pełnej prywatności. Sprawia jedynie, że ich nazwy są nietypowe i jest raczej mało prawdopodobne, aby ujawniły się w zewnętrznym kodzie. Jeżeli chcesz się o tym przekonać, wpisz poniższy wiersz:

```
>>> fowl._Duck__name
'Donald'
```

Zwróć uwagę, że nie pojawił się napis `Wywołany getter`. Choć atrybuty nie są w ten sposób skutecznie chronione, ich nietypowe nazwy zmniejszają prawdopodobieństwo przypadkowego lub zamierzonego bezpośredniego odwołania się do nich.

Atrybuty klas i obiektów

Atrybut, któremu w definicji klasy zostanie przypisana wartość, jest dziedziczony przez obiekty utworzone na podstawie tej klasy:

```
>>> class Fruit:
...         color = 'czerwony'
...
>>> blueberry = Fruit()
>>> Fruit.color
'czerwony'
>>> blueberry.color
'czerwony'
```

Jednak zmiana wartości atrybutu w obiekcie nie jest odzwierciedlana w klasie:

```
>>> blueberry.color = 'niebieski'
>>> blueberry.color
'blue'
```

```
>>> Fruit.color
'czerwony'
```

Jeżeli wartość atrybutu w klasie zostanie zmieniona, nie wpłynie to na istniejące obiekty:

```
>>> Fruit.color = 'żółty'
>>> Fruit.color
'żółty'
>>> blueberry.color
'niebieski'
```

Za to wpłynie na nowe:

```
>>> new_fruit = Fruit()
>>> new_fruit.color
'żółty'
```

Rodzaje metod

Niektóre metody są częściami klasy, inne częściami obiektów utworzonych na podstawie danej klasy, a jeszcze inne nie należą do żadnej z tych dwóch kategorii. Poniżej przedstawione jest krótkie podsumowanie:

- Jeżeli metoda nie jest opatrzona żadnym dekoratorem, jest **metodą instancji**. Jej pierwszym argumentem jest `self`, w którym umieszczane jest odwołanie do obiektu.
- Jeżeli metoda jest opatrzona dekoratorem `@classmethod`, jest **metodą klasy**. Jej pierwszym argumentem jest `cls` (nazwa może być dowolna, byle tylko nie była zastrzeżona), w którym umieszczane jest odwołanie do klasy.
- Jeżeli metoda jest opatrzona dekoratorem `@staticmethod`, jest **metodą statyczną**. Jej pierwszym argumentem nie może być klasa ani obiekt.

W kolejnych podrozdziałach opisane są dokładniej poszczególne rodzaje metod.

Metody instancji

Jeżeli pierwszym argumentem metody zawartej w definicji klasy jest `self`, oznacza to, że jest to **metoda instancji**. W tym argumencie jest umieszczane odwołanie do obiektu, do którego ta metoda należy. Tego rodzaju metody tworzy się podczas definiowania klasy najczęściej. Zostały one również wykorzystane w opisanych wcześniej przykładach.

Metody klasy

Drugi rodzaj metod dotyczy klasy jako całości. Wszelkie zmiany wprowadzane w klasie są odzwierciedlane w tworzonych na jej podstawie obiektach. **Metodę klasy** definiuje się przez poprzedzenie jej nazwy dekoratorem `@classmethod`. Jej pierwszym argumentem jest klasa. Zgodnie z przyjętą konwencją argument ten ma nazwę `cls` (nie może to być słowo `class`, ponieważ jest ono zarezerwowane). Zdefiniujmy klasę `A` zawierającą metodę, która będzie zwracać liczbę obiektów utworzonych na podstawie tej klasy:

```
>>> class A():
...         count = 0
...         def __init__(self):
...             A.count += 1
...         def exclaim(self):
...             print("Jestem klasą A!")
...         @classmethod
...         def kids(cls):
...             print("Liczba obiektów:", cls.count)
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
Liczba obiektów: 3
```

Zwróć uwagę, że w metodzie `__init__()` zostało użyte odwołanie do atrybutu klasy `A.count`, a nie `self.count`, który jest atrybutem obiektu. W metodzie `kids()` jest natomiast użyty atrybut `cls.count`, ale równie dobrze może to być `A.count`.

Metody statyczne

Metody trzeciego typu zdefiniowane w klasie nie wpływają ani na nią samą, ani na jej obiekty. Definiuje się je dla wygody, aby nie plątały się w kodzie jako samodzielne funkcje. Nazwę **metody statycznej** poprzedza się dekoratorem `@staticmethod`. Jej pierwszym argumentem nie może być `self` ani `cls`. Poniżej przedstawiony jest przykład:

```
>>> class CoyoteWeapon():
...         @staticmethod
...         def commercial():
...             print('Broń CoyoteWeapon dostarcza firma Acme')
...
>>>
>>> CoyoteWeapon.commercial()
Broń CoyoteWeapon dostarcza firma Acme
```

Zwróć uwagę, że w celu odwołania się do metody nie został na podstawie klasy `CoyoteWeapon` utworzony obiekt.

Kacze typowanie

Implementacja *polimorfizmu* w Pythonie jest dość luźna. Polega na wywoływaniu metod o takich samych nazwach i argumentach niezależnie od tego, w jakich klasach zostały zdefiniowane.

Zdefiniujmy klasę `Quote` i dwie klasy pochodne. Niech każda z nich wykorzystuje tę samą metodę inicjującą `__init__()`. Dodatkowo zdefiniujmy dwie metody:

- `who()` — zwracającą wartość atrybutu `person`,
- `says()` — zwracającą wartość atrybutu `words` z dopisanym znakiem interpunkcyjnym.

Poniżej przedstawiony jest kod tych klas:

```
>>> class Quote():
...         def __init__(self, person, words):
...             self.person = person
...             self.words = words
...         def who(self):
...             return self.person
...         def says(self):
...             return self.words + '.'
...
>>> class QuestionQuote(Quote):
...          def says(self):
...              return self.words + '?'
...
>>> class ExclamationQuote(Quote):
...          def says(self):
...              return self.words + '!'
...
>>>
```

Ponieważ nie trzeba zmieniać sposobu inicjowania klas `QuestionQuote` i `ExclamationQuote`, nie została w nich nadpisana metoda `__init__()`. W efekcie obie klasy automatycznie wywołują metodę `__init__()` klasy rodzicielskiej `Quote`, przypisującą wartości atrybutom `person` i `words`. Dzięki temu można odwoływać się do atrybutu `self.words` w obiektach utworzonych na podstawie klas `QuestionQuote` i `ExclamationQuote`.

Teraz utwórzmy obiekty:

```
>>> hunter = Quote('Elmer Fudd', 'poluję na króliki')
>>> print(hunter.who(), 'mówi:', hunter.says())
Elmer Fudd mówi: poluję na króliki.

>>> hunted1 = QuestionQuote('Królik Bugs', 'o co chodzi')
>>> print(hunted1.who(), 'mówi:', hunted1.says())
Królik Bugs mówi: o co chodzi?

>>> hunted2 = ExclamationQuote('Kaczor Donald', 'to jest sezon na króliki')
>>> print(hunted2.who(), 'mówi:', hunted2.says())
Kaczor Donald mówi: to jest sezon na króliki!
```

Każda z trzech wersji metody `says()` działa inaczej. Jest to polimorfizm, typowy dla języków obiektowych. Python w tej kwestii idzie dalej i umożliwia odwoływanie się do metod `who()` i `says()` we *wszystkich* obiektach, w których te metody są. Zdefiniujmy klasę `BabblingBrook`, która w żaden sposób nie będzie powiązana z klasami `QuestionQuote` i `ExclamationQuote`:

```
>>> class BabblingBrook():
...         def who(self):
...             return 'Brook'
...         def says(self):
...             return 'cześć!'
...
>>> brook = BabblingBrook()
```

Wywołajmy teraz metody `who()` i `says()` należące od różnych obiektów, z których jeden, `brook`, jest całkowicie niezależny od pozostałych:

```
>>> def who_says(obj):
...       print(obj.who(), 'mówi:', obj.says())
...
>>> who_says(hunter)
Elmer Fudd mówi: poluję na króliki.
>>> who_says(hunted1)
Królik Bugs mówi: o co chodzi?
>>> who_says(hunted2)
Kaczor Donald mówi: to jest sezon na króliki!
>>> who_says(brook)
Brook mówi: cześć!
```

Tego rodzaju działanie jest nazywane **kaczym typowaniem**, zgodnie z ludową mądrością:

> *Jeżeli coś wygląda jak kaczka, chodzi jak kaczka i kwacze jak kaczka, to znaczy, że jest to kaczka.*

Przecież ludowych mądrości się nie kwestionuje, prawda?

Magiczne metody

Potrafisz już tworzyć i stosować proste obiekty. To, czego nauczysz się w tym podrozdziale, może Cię mile zaskoczyć.

Wpisując na przykład instrukcję `a = 3 + 8`, możesz się zastanawiać, skąd obiekty zawierające liczby 3 i 8 „wiedzą”, jak interpretować znak dodawania. Z kolei skąd wiadomo, że w zapisie `name = 'Kaczor' + ' ' + 'Donald'` ten sam symbol oznacza łączenie ciągów znaków? I skąd zmienna `name` „wie”, w jaki sposób użyć znaku równości w celu uzyskania wyniku? Wszystko dzieje się za sprawą **specjalnych**, czy też, mówiąc bardziej tajemniczo, **magicznych metod**.

Nazwy tych metod rozpoczynają się od podwójnego znaku podkreślenia (__). Dlaczego stosowana jest taka konwencja? Ponieważ dzięki niej jest mało prawdopodobne, aby programista tak samo nazwał swoje metody. Jedną z takich metod już poznałeś: jest to `__init__()`, która inicjuje nowo utworzony obiekt z wykorzystaniem wartości podanych w argumentach. Wiesz również (patrz podrozdział „Prywatność dzięki modyfikacji nazwy”), dlaczego za pomocą znaków podkreślenia modyfikuje się nazwy atrybutów i metod.

Załóżmy, że mamy prostą klasę `Word` i potrzebna jest metoda `equals()` porównująca dwa słowa bez uwzględniania wielkości liter, tj. taka, która słowa `'cześć'` i `'CZEŚĆ'` traktuje tak samo.

Poniższy przykład przedstawia pierwszą wersję takiej klasy, zawierającą wywoływaną w zwykły sposób metodę `equals()`. Atrybut `self.text` zawiera ciąg znaków przypisany obiektowi utworzonemu na podstawie klasy `Word`, a metoda `equals()` porównuje ten ciąg z innym, zawartym w obiekcie utworzonym na podstawie tej samej klasy:

```
>>> class Word():
...       def __init__(self, text):
...           self.text = text
...
...       def equals(self, word2):
...           return self.text.lower() == word2.text.lower()
...
```

Utwórzmy teraz trzy obiekty zawierające trzy różne ciągi znaków:

```
>>> first = Word('cześć')
>>> second = Word('CZEŚĆ')
>>> third = Word('witaj')
```

Ciągi `'cześć'` i `'CZEŚĆ'` po zamianie znaków na małe litery są takie same:

```
>>> first.equals(second)
True
```

Natomiast ciąg `'witaj'` nie jest tym samym, co `'cześć'`:

```
>>> first.equals(third)
False
```

Metoda `equals()` zamienia wszystkie znaki obu ciągów na małe litery i porównuje je. Wygodniej byłoby jednak użyć operatora `==`, używanego do porównywania wbudowanych typów danych. Zróbmy więc coś. Nadajmy metodzie nową, specjalną nazwę `__eq__()` (za chwilę dowiesz się, dlaczego właśnie taką):

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

Sprawdźmy, czy to działa:

```
>>> first = Word('cześć')
>>> second = Word('CZEŚĆ')
>>> third = Word('witaj')
>>> first == second
True
>>> first == third
False
```

Magia! Wystarczyło użyć metody o specjalnej nazwie `__eq__()`. Tabele 11.1 i 11.2 zawierają nazwy najczęściej stosowanych magicznych metod.

Tabela 11.1. Magiczne metody porównujące

Metoda	Opis
`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__gt__(self, other)`	`self > other`
`__le__(self, other)`	`self <= other`
`__ge__(self, other)`	`self >= other`

Tabela 11.2. Magiczne metody wykonujące operacje arytmetyczne

Metoda	Opis
`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self – other`
`__mul__(self, other)`	`self * other`
`__floordiv__(self, other)`	`self // other`
`__truediv__(self, other)`	`self / other`
`__mod__(self, other)`	`self % other`
`__pow__(self, other)`	`self ** other`

Operatory arytmetyczne, takie jak + (magiczna metoda `__add__()`) oraz – (metoda `__sub__()`), można stosować nie tylko z liczbami. Na przykład znak + służy do łączenia ciągów znaków, a * do ich powielania. Specjalnych metod jest znacznie więcej. Są one opisane na stronie *https://docs.python.org/pl/3.13/reference/datamodel.html#special-method-names*. Tabela 11.3 zawiera najczęściej stosowane.

Tabela 11.3. Kilka różnych magicznych metod

Metoda	Opis
`__str__(self)`	`str(self)`
`__repr__(self)`	`repr(self)`
`__len__(self)`	`len(self)`

Jedną z najczęściej implementowanych metod obok `__init__()` jest `__str__()`. Metoda ta jest wykorzystywana do wyświetlania informacji o obiekcie za pomocą funkcji `print()`, `str()` oraz funkcji formatujących, opisanych w rozdziale 5. Metodę `__repr__()` wykorzystuje zaś interaktywny interpreter do wyświetlania wartości. Jeżeli nie zdefiniuje się metody `__str__()` ani `__repr__()`, zostanie wyświetlony domyślny ciąg opisujący dany obiekt:

```
>>> first = Word('cześć')
>>> first
<__main__.Word object at 0x1006ba3d0>
>>> print(first)
<__main__.Word object at 0x1006ba3d0>
```

Zdefiniujmy więc metody `__str__()` i `__repr__()`, aby wyświetlać bardziej przydatne informacje o klasie `Word`:

```
>>> class Word():
...         def __init__(self, text):
...             self.text = text
...         def __eq__(self, word2):
...             return self.text.lower() == word2.text.lower()
...         def __str__(self):
...             return self.text
...         def __repr__(self):
...             return 'Word("'  + self.text  + '")'
...
```

```
>>> first = Word('cześć')
>>> first          # Wywoływana metoda __repr__()
Word("cześć")
>>> print(first)    # Wywoływana metoda __str__()
cześć
```

Gdy będziesz tworzyć własne klasy, możliwe, że będziesz chciał pisać takie metody. W podrozdziale „Klasy danych" dowiesz się, czy trzeba je pisać samodzielnie. Więcej informacji na temat specjalnych metod znajdziesz w oficjalnej dokumentacji języka Python na stronie *https://docs.python.org/pl/3.13/reference/datamodel.html#special-method-names*.

Agregowanie i komponowanie klas

Dziedziczenie klas jest przydatną techniką, gdy klasa pochodna musi w większości sytuacji działać tak jak jej klasa rodzicielska. Zazwyczaj w takich przypadkach tworzy się rozbudowane struktury dziedziczenia, ale przez to klasy pochodne są ściśle powiązane ze szczegółami klasy rodzicielskiej.

Czasami bardziej uzasadnione jest **agregowanie** i **komponowanie** klas. Na czym polega różnica pomiędzy obiema technikami? Komponowanie opiera się na założeniu, że dana rzecz jest częścią innej rzeczy. Kaczka jest ptakiem (dziedziczenie) i ma ogon (komponowanie). Ogon nie jest kaczką, tylko jej częścią. W poniższym przykładzie tworzone są obiekty `a_bill` i `a_tail`, przekazywane następnie nowemu obiektowi `duck`:

```
>>> class Bill():
...         def __init__(self, description):
...             self.description = description
...
>>> class Tail():
...         def __init__(self, length):
...             self.length = length
...
>>> class Duck():
...         def __init__(self, bill, tail):
...             self.bill = bill
...             self.tail = tail
...         def about(self):
...             print('Kaczka ma', self.bill.description,
...                 'dziób i', self.tail.length, 'ogon')
...
>>> a_tail = Tail('długi')
>>> a_bill = Bill('żółty')
>>> duck = Duck(a_bill, a_tail)
>>> duck.about()
Kaczka ma żółty dziób i długi ogon
```

Agregacja klas wyraża relacje między nimi, ale jest dość luźna. Jedna klasa wykorzystuje drugą, ale obie istnieją niezależnie od siebie.

Kiedy stosować obiekty, a kiedy inne struktury?

Poniżej znajduje się kilka wskazówek ułatwiających podejmowanie decyzji, czy kod i dane należy umieścić w klasie, module (opisanym w rozdziale 12.) czy w innej strukturze:

- Obiekty są najbardziej przydatne wtedy, gdy potrzebnych jest kilka podobnie działających instancji (odpowiedzialne są za to metody), ale różniących się wewnętrznymi stanami (wartościami atrybutów).
- Klasy, w odróżnieniu od modułów, podlegają dziedziczeniu.
- Jeżeli potrzebny jest tylko jeden egzemplarz czegoś, najlepiej użyć modułu. Niezależnie od tego, ile razy w kodzie pojawia się odwołanie do modułu, ładowany jest on tylko raz (do programistów Java i C++: moduł w Pythonie jest odpowiednikiem *singletonu*).
- Innym, opartym na klasie, sposobem na uzyskanie jednego, spójnego działania jest **wzorzec Borg** (jego nazwa wzięła się od obcych z serialu *Star Trek: Następne pokolenie*).
- Jeżeli w argumentach różnych funkcji trzeba umieszczać kilka wartości, warto je zebrać w klasę. Na przykład do opisania kolorowego obrazu można użyć słownika z kluczami `size` i `color`. Dla poszczególnych obrazów można tworzyć osobne słowniki i umieszczać je w argumentach funkcji `scale()` oraz `transform()`. Jednak takie podejście sprawi, że w kodzie w miarę tworzenia kolejnych kluczy i funkcji pojawi się bałagan. Lepszym sposobem jest zdefiniowanie klasy `Image` zawierającej atrybuty `size` i `color` oraz metody `scale()` i `transform()`. Dzięki temu wszystkie dane i metody związane z kolorowym obrazem będą się znajdowały w jednym miejscu.
- W celu rozwiązania problemu należy stosować najprostsze struktury. Słownik, lista czy krotka są prostsze, mniejsze i szybsze niż moduły, które z kolei są prostsze niż klasy.

  Rada twórcy Pythona:

  > *Unikaj tworzenia nadmiernie zaawansowanych struktur. Krotki (również nazwane) są lepsze niż obiekty. Stosuj proste pola zamiast getterów i setterów. Twoimi sprzymierzeńcami są wbudowane typy danych. Jak najczęściej używaj liczb, ciągów znaków, krotek, list, zbiorów i słowników. Korzystaj z bibliotek kolekcji, szczególnie* `deque`.
  >
  > — Guido van Rossum

- Stosuj nową, alternatywną strukturę zwaną **klasą danych**, opisaną w podrozdziale „Klasy danych”.

Nazwane krotki

W tym miejscu warto przedstawić nazwaną krotkę, o której wspomina Guido. Jest to klasa pochodna od krotki. Do umieszczonych w niej wartości można odwoływać się zarówno za pomocą nazw (z użyciem sufiksu `.nazwa`), jak i pozycji (z użyciem zapisu `[indeks]`).

Przekształćmy klasę `Duck` z poprzedniego przykładu w nazwaną krotkę, w której `bill` i `tail` będą zwykłymi ciągami znaków. Użyjemy do tego celu funkcji `namedtuple()`, która ma dwa argumenty:

- nazwę,
- ciąg znaków zawierający nazwy pól oddzielone spacjami.

Nazwane krotki nie są domyślnie dostępne. Aby móc ich używać, należy załadować odpowiedni moduł, jak w pierwszym wierszu poniższego przykładu:

```
>>> from collections import namedtuple
>>> Duck = namedtuple('Duck', 'bill tail')
>>> duck = Duck('żółty', 'długi')
>>> duck
Duck(bill='żółty', tail='długi')
>>> duck.bill
'żółty'
>>> duck.tail
'długi'
```

Nazwaną krotkę można również utworzyć za pomocą słownika:

```
>>> parts = {'bill': 'żółty', 'tail': 'długi'}
>>> duck2 = Duck(**parts)
>>> duck2
Duck(bill='żółty', tail='długi')
```

Zwróć uwagę na użyty w powyższym kodzie argument `**pars`. Jest to argument nazwany. Klucze i wartości są wyodrębniane ze słownika `pars` i umieszczane w argumentach klasy `Duck`. Uzyskany wynik jest taki sam jak w przypadku następującego kodu:

```
>>> duck2 = Duck(bill = 'żółty', tail = 'długi')
```

Nazwana krotka jest niemutowalna, ale można zamieniać jej pola i tworzyć w ten sposób inne nazwane krotki:

```
>>> duck3 = duck2._replace(tail='imponujący', bill='szeroki')
>>> duck3
Duck(bill='szeroki', tail='imponujący')
```

Można również utworzyć zmienną `duck` zawierającą słownik:

```
>>> duck_dict = {'bill': 'żółty', 'tail': 'długi'}
>>> duck_dict
{'tail': 'długi', 'bill': 'żółty'}
```

Do słownika można dodawać pola:

```
>>> duck_dict['color'] = 'zielony'
>>> duck_dict
{'color': 'zielony', 'tail': 'długi', 'bill': 'żółty'}
```

Nie można jednak tak robić w przypadku krotki:

```
>>> duck.color = 'zielony'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute 'color'
```

Podsumowując, nazwana krotka ma następujące zalety i wady:

- Wyglądem i działaniem przypomina niemutowalny obiekt.
- Zajmuje mniej miejsca w pamięci niż obiekt i jest od niego szybsza.
- Do jej atrybutów można się odwoływać z użyciem zapisu z kropką lub nawiasów klamrowych, jak w słowniku.
- Można ją stosować jako klucz w słowniku.

Klasy danych

Często programiści tworzą obiekty tylko w celu przechowywania danych w atrybutach, a nie przetwarzania ich za pomocą metod. Wiesz już, że nazwane krotki można stosować do gromadzenia danych. W wersji Python 3.7 pojawiły się dodatkowo klasy danych.

W poniższym przykładzie tworzony jest zwykły obiekt, zawierający jedynie atrybut `name`:

```
>> class TeenyClass():
...      def __init__(self, name):
...          self.name = name
...
>>> teeny = TeenyClass('imię')
>>> teeny.name
'imię'
```

Ten sam efekt można osiągnąć w nieco inny sposób za pomocą klasy danych:

```
>>> from dataclasses import dataclass
>>> @dataclass
...    class TeenyDataClass:
...        name: str
...
>>> teeny = TeenyDataClass('imię')
>>> teeny.name
'imię'
```

Jak widać, trzeba było nie tylko użyć dekoratora `@dataclass`, ale również zdefiniować atrybuty za pomocą adnotacji (*https://oreil.ly/NyGfE*). Adnotacja ma postać *typ* lub *`nazwa: typ = wartość`*, na przykład `color: str` lub `color: str = "czerwony"` (w rozdziale 14. znajdziesz informacje na temat adnotacji typów). Można stosować nie tylko wbudowane typy, takie jak `str` czy `int`, ale dowolne, w tym własne klasy.

Aby utworzyć klasę danych, należy podać argumenty w kolejności określonej w definicji klasy. Natomiast nazwane argumenty można stosować w dowolnej kolejności:

```
>>> from dataclasses import dataclass
>>> @dataclass
...     class AnimalClass:
...         name: str
...         habitat: str
...         teeth: int = 0
...
>>> snowman = AnimalClass('yeti', 'Himalaje', 46)
```

```
>>> duck = AnimalClass(habitat='staw', name='kaczka')
>>> snowman
AnimalClass(name='yeti', habitat='Himalaje', teeth=46)
>>> duck
AnimalClass(name='kaczka', habitat='staw', teeth=0)
```

W klasie `AnimalClass` jest zdefiniowana domyślna wartość atrybutu `teeth`, więc nie trzeba go określać w celu utworzenia obiektu reprezentującego kaczkę.

Do atrybutów obiektu można odwoływać się w taki sam sposób jak do atrybutów każdego innego obiektu:

```
>>> duck.habitat
'staw'
>>> snowman.teeth
46
```

Muszę wspomnieć, że klasy danych definiują wiele cech automatycznie, więc nie musisz ich pisać samodzielnie, dzięki czemu zaoszczędzisz wiele czasu. O klasach danych można powiedzieć znacznie więcej. Zajrzyj na stronę podręcznika (*https://realpython.com/python-data-classes/*) lub do oficjalnej (bardzo obszernej) dokumentacji (*https://oreil.ly/J19Yl*).

Pakiet attrs

Dowiedziałeś się, jak się tworzy klasy i atrybuty. Wiesz, że wymaga to wpisywania mnóstwa kodu, m.in. definiowania metody `__init__()`, przypisywania argumentom wartości za pomocą atrybutu `self` i definiowania różnych metod z podwójnymi znakami podkreślenia w nazwach, na przykład `__str__()`. Nazwane krotki i klasy danych dostępne w standardowej bibliotece są alternatywnymi, prostszymi strukturami, które można wykorzystywać do tworzenia kolekcji danych.

Artykuł *One Python Library Everyone Needs* („Pewna biblioteka, której każdy potrzebuje”, *https://oreil.ly/QbbI1*) zawiera porównanie zwykłych klas, nazwanych krotek i klas danych. Jego autor zaleca, aby korzystać z zewnętrznego pakietu `attrs` (*https://oreil.ly/Rdwlx*), dzięki któremu można wpisywać mniej kodu, weryfikować poprawność danych i wykonywać wiele innych operacji. Rzuć okiem na ten moduł i oceń, czy bardziej Ci się przyda niż standardowo dostępne techniki.

Powtórka/zapowiedź

To był dość długi rozdział o obiektach i wiele wiedzy do przyswojenia. Możliwe, że nigdy nie będziesz musiał pisać własnej klasy! Może zabrzmię jak heretyk[3], ale możesz programować w Pythonie, używając innych wbudowanych danych i struktur kodu.

W praktyce część zastosowań klas można równie dobrze obsłużyć przy użyciu modułów i pakietów, którymi zajmiemy się w następnym rozdziale.

[3] Wróć do słów Guido, które zacytowałem wcześniej.

Do zrobienia

11.1. Utwórz pustą klasę `Thing` i wyświetl ją. Następnie utwórz na jej podstawie obiekt `example` i też go wyświetl. Czy uzyskane wyniki są takie same, czy różne?

11.2. Utwórz nową klasę `Thing2` i przypisz jej atrybutowi `letters` ciąg `'abc'`. Wyświetl ten atrybut.

11.3. Utwórz jeszcze jedną klasę, `Thing3`. Tym razem przypisz atrybutowi `letters` instancji (obiektu) ciąg `'xyz'`. Wyświetl ten atrybut. Czy w tym celu trzeba utworzyć obiekt na podstawie klasy?

11.4. Utwórz klasę `Element` zawierającą atrybuty instancji `name`, `symbol` i `number`. Na jej podstawie utwórz obiekt i przypisz atrybutom wartości, odpowiednio, `'wodór'`, `'H'` oraz `1`.

11.5. Zdefiniuj słownik z następującymi kluczami i wartościami: `{'name': 'wodór', 'symbol': 'H', 'number': 1}`. Następnie utwórz obiekt `hydrogen` na podstawie klasy `Element` z wykorzystaniem tego słownika.

11.6. W klasie `Element` zdefiniuj metodę `dump()` wyświetlającą wartości atrybutów obiektu (`name`, `symbol` i `number`). Utwórz na podstawie nowej klasy obiekt `hydrogen` i wyświetl jego atrybuty za pomocą metody `dump()`.

11.7. Wywołaj funkcję `print(hydrogen)`. W klasie `Element` zmień nazwę `dump` na `__str__`, utwórz nowy obiekt `hydrogen` i ponownie wywołaj funkcję `print(hydrogen)`.

11.8. Zmień definicję klasy `Element` tak, aby atrybuty `name`, `symbol` i `number` były prywatne. Dla każdego z nich utwórz getter zwracający jego wartość.

11.9. Zdefiniuj klasy `Bear`, `Rabbit` i `Duck`. W każdej z nich zdefiniuj tylko jedną metodę, `eats()`, która niech zwraca w klasie `Bear` ciąg `'jagody'`, w klasie `Rabbit` ciąg `'sałata'`, a w klasie `Duck` ciąg `'zielenina'`. Utwórz obiekty na podstawie każdej z klas i wyświetl, co każdy reprezentowany przez nie zwierzak je.

11.10. Zdefiniuj klasy `Laser`, `Claw` i `SmartPhone`. Niech każda z nich zawiera tylko jedną metodę, `does()`, która niech zwraca w klasie `Laser` ciąg `'niszczy'`, w klasie `Claw` ciąg `'chwyta'`, a w klasie `SmartPhone` ciąg `'dzwoni'`. Następnie zdefiniuj klasę `Robot`, której atrybutami niech będą obiekty każdej z powyższych klas. W klasie `Robot` zdefiniuj metodę `does()` wyświetlającą informację o tym, co robi każdy komponent.

Opanuj Pythona od podstaw po AI i twórz nowoczesny kod bez tajemnic!

Python to jeden z najpopularniejszych języków programowania na świecie. Jego wszechstronność, czytelność i zgromadzona wokół niego ogromna społeczność sprawiają, że napędza on rewolucję technologiczną — od tworzenia nowoczesnych aplikacji webowych, przez analizę danych, aż po rozwój sztucznej inteligencji i uczenia maszynowego. Ta pozycja to idealny przewodnik dla każdego, kto chce wejść do tego fascynującego świata lub uporządkować swoją wiedzę. W dobie błyskawicznie zmieniających się trendów IT solidne opanowanie Pythona to inwestycja, która gwarantuje przewagę na rynku pracy i otwiera drzwi do kariery w najbardziej innowacyjnych projektach.

Książka stanowi kompleksowe, a zarazem przystępne wprowadzenie do programowania w Pythonie. Autor prowadzi czytelnika od absolutnych podstaw — takich jak zmienne, typy danych, pętle i funkcje — aż po zaawansowane zagadnienia niezbędne w codziennej pracy profesjonalnego programisty. W trzecim wydaniu zaktualizowano wszystkie informacje i dodano zupełnie nowe rozdziały poświęcone sztucznej inteligencji, data science i optymalizacji wydajności kodu. Czytelnik dowie się, jak zarządzać środowiskiem programistycznym, korzystać z relacyjnych i nierelacyjnych baz danych, tworzyć aplikacje sieciowe, a także wdrażać współbieżność. To praktyczny podręcznik, który uczy nie tylko samej składni, ale przede wszystkim „pythonicznego" sposobu rozwiązywania problemów.

Ta książka pokazuje, jak Python krzyżuje się z innymi ważnymi dziedzinami (sztuczną inteligencją, bazami danych, serwerami sieciowymi itd.) w dzisiejszym świecie, który tak szybko się zmienia.

Patrick Viafore, autor książki *Robust Python*

Najważniejsze zagadnienia:

- Podstawy języka: struktury danych, pętle, funkcje i obiekty
- Zarządzanie środowiskiem, testowanie i debugowanie kodu
- Praca z bazami danych (SQL, NoSQL) i plikami
- Tworzenie aplikacji sieciowych i API (między innymi FastAPI, Flask)
- Nowoczesne trendy: sztuczna inteligencja i data science
- Optymalizacja wydajności i współbieżność (asyncio, Mojo)

Bill Lubanovic to programista z ponad 40-letnim doświadczeniem. Pracował między innymi z Uniksem, sieciami i Pythonem dla takich firm jak Internet Archive, CrowdStrike czy Northwest Airlines. Jest autorem bestsellerowych książek, w tym *Szybki jak FastAPI*, i współautorem publikacji o administrowaniu Linuksem. Mieszka w górach w Minnesocie, gdzie nieustannie zgłębia nowe technologie.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-4091-8
9 788328 940918
Cena: 139,00 zł