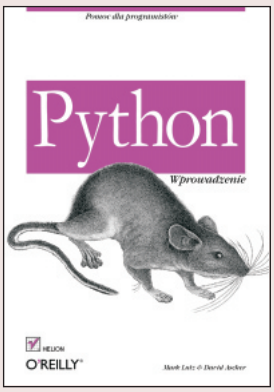
Python

Autorzy: Mark Lutz, David Ascher
Tłumaczenie: Zygmunt Wereszczyński
ISBN: 83-7197-596-1
Tytuł oryginału: Learning Python
Format: B5, stron: 365



Niniejsza książka stanowi wprowadzenie do języka Python. Jest to popularny język programowania obiektowego, używany zarówno w programach działających samodzielnie, jak i w skryptach obejmujących różne dziedziny zastosowań. Python jest bezpłatny, przenośny, bardzo wydajny i wyjątkowo łatwy w użyciu. Bez względu na to, czy ktoś jest nowicjuszem w programowaniu, czy też profesjonalistą, celem tej książki jest szybkie zapoznanie go z istotą języka Python.

Tekst niniejszej książki obejmuje podstawy języka Python; celowo zawęziliśmy jej zakres tematyczny, mając na uwadze łatwość korzystania z niej i jej objętość. Mówiąc inaczej: prezentacja materiału koncentruje się wokół zasadniczych koncepcji i czasem bywa odpowiednio uproszczona. Dlatego właśnie niniejsza książka stanowi znakomity opis języka Python, zarówno jeśli chodzi o wprowadzenie, jak i dalsze, bardziej zaawansowane etapy.

Pomimo tak ograniczonego celu książki (a może właśnie dlatego) sądzimy, że będzie ona dla czytelnika pierwszą wielką lekturą na temat programowania w języku Python. Można się z niej nauczyć wszystkiego, co jest potrzebne w początkowej fazie samodzielnego tworzenia przydatnych programów i skryptów w Pythonie. Po zakończeniu lektury Czytelnik nie tylko będzie znał sam język, ale także będzie wiedział, jak go zastosować w codziennych zadaniach. Książka daje także dobre przygotowanie do podjęcia bardziej zaawansowanych tematów, które programista spotka w swojej praktyce.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp *7*

Część I ***Rdzeń języka*** **15**

Rozdział 1. ***Pierwsze kroki*** ***17***

Dlaczego właśnie Python? 17
Jak należy uruchamiać programy w języku Python? 24
Pierwsze spojrzenie na pliki modułowe 30
Szczegóły związane z konfiguracją języka Python 32
Podsumowanie 36
Ćwiczenia 37

Rozdział 2. ***Typy i operatory*** ***39***

Struktura programu w języku Python 39
Przyczyny stosowania typów wbudowanych 39
Liczby 41
Łańcuchy 48
Listy 56
Słowniki 61
Krotki 65
Pliki 67
Ogólne właściwości obiektów 69
„Niespodzianki" typów wbudowanych 75
Podsumowanie 77
Ćwiczenia 78

Rozdział 3. ***Instrukcje podstawowe*** ***81***
Przypisanie 82
Wyrażenia 85
Instrukcja print 86
Testy if 87
Pętle while 94
Pętle for 97
Niespodzianki spotykane w kodzie 102
Podsumowanie 103
Ćwiczenia 104

Rozdział 4. ***Funkcje*** ***107***
Dlaczego należy stosować funkcje? 107
Podstawy funkcji 108
Zakresy działania w funkcjach 111
Przekazywanie argumentów 114
Informacje dodatkowe 120
Niespodzianki w funkcjach 126
Podsumowanie 131
Ćwiczenia 132

Rozdział 5. ***Moduły*** ***135***
Dlaczego używa się modułów? 135
Podstawowe informacje o modułach 136
Pliki modułowe są przestrzeniami nazw 138
Model importu 140
Ponowne ładowanie modułów 142
Informacje dodatkowe 144
Modułowe niespodzianki 151
Podsumowanie 156
Ćwiczenia 156

Rozdział 6. ***Klasy*** ***159***
Dlaczego należy używać klas? 159
Podstawy działania klas 160
Używanie instrukcji class 167
Zastosowania metod klasy 168

Dziedziczenie i przeszukiwanie drzew przestrzeni nazw ... 169
Przeciążanie operatorów w klasach ... 173
Komplet reguł dla przestrzeni nazw ... 176
Zastosowania klas w programach ... 178
Informacje dodatkowe ... 188
Niespodzianki w klasach ... 190
Podsumowanie ... 196
Ćwiczenia ... 196

Rozdział 7. *Wyjątki* ... *201*
Dlaczego należy używać wyjątków? ... 201
Podstawy działania wyjątków ... 202
Idiomatyczne określenia wyjątków ... 206
Tryby wychwytywania wyjątków ... 208
Informacje dodatkowe ... 211
Wyjątkowe niespodzianki ... 215
Podsumowanie ... 217
Ćwiczenia ... 218

Część II *Warstwy zewnętrzne* ... **221**

Rozdział 8. *Narzędzia wbudowane* ... *223*
Funkcje wbudowane ... 225
Moduły biblioteki ... 232
Ćwiczenia ... 249

Rozdział 9. *Typowe zadania w języku Python* ... *251*
Manipulacja strukturami danych ... 251
Manipulacja plikami ... 257
Manipulacja programami ... 268
Działanie w Internecie ... 270
Przykłady większych programów ... 273
Ćwiczenia ... 278

Rozdział 10. *Osnowy programowania i aplikacje* *281*

System automatycznego zgłaszania skarg 282
Tworzenie interfejsu za pomocą COM: tani rzecznik prasowy 287
Edytor elementów formularza danych 292
Rozważania o projektowaniu 297
JPython: szczęśliwy związek języków Python i Java 298
Inne osnowy i aplikacje 305
Ćwiczenia 307

Dodatki **309**

Dodatek A *Materiały źródłowe języka Python* *311*

Dodatek B *Zagadnienia związane z różnymi platformami* *323*

Dodatek C *Rozwiązania ćwiczeń* *329*

Skorowidz *357*

6

Klasy

W rozdziale tym przedstawimy klasę języka Python, czyli twór, który służy do wprowadzania nowych rodzajów obiektów w Pythonie. Klasy stanowią najważniejsze narzędzie programowania obiektowego, zatem w tym rozdziale omówimy także podstawy tego programowania. Klasy w języku Python tworzone są za pomocą nie omawianej dotychczas instrukcji `class`. Obiekty zdefiniowane za pomocą klas mogą wyglądać bardzo podobnie, jak wbudowane typy, które były już opisywane w tej książce.

Na jedną rzecz należy szczególnie zwrócić uwagę: obiektowość języka Python jest całkowicie opcjonalna i nie trzeba używać klas, by rozpocząć pracę w tym języku. Wiele zadań można w rzeczywistości wykonać za pomocą prostszych konstrukcji, takich jak np. funkcje. Klasy okazują się jednak bardzo przydatne w Pythonie i mamy nadzieję, że to udowodnimy. Mają one także swój udział w powszechnie stosowanych narzędziach języka Python, takich jak Tkinter, i większość programistów uważa, że przydaje się tu znajomość klas co najmniej w stopniu podstawowym.

Dlaczego należy używać klas?

Czytelnicy z pewnością pamiętają, że programy wykonują zadania za pomocą „nadziewania". Mówiąc prościej, klasy są po prostu sposobem definiowania nowych rodzajów nadzienia, które symbolizuje prawdziwe obiekty w dziedzinie tworzonych programów. Załóżmy na przykład, że zdecydowaliśmy się utworzyć hipotetycznego robota robiącego pizzę (wzmianka w rozdziale 4.). Jeśli do tego celu zastosujemy klasy, to możemy wymodelować taką strukturę robota, która będzie bliższa jego rzeczywistemu światu i stosunkom w nim panującym:

Dziedziczenie
: Roboty wytwarzające pizzę należą do rodzaju robotów i dlatego mają wszystkie zwyczajne właściwości robotów. Używając określeń z dziedziny programowania obiektowego, mówimy, że dziedziczą one właściwości ogólnej kategorii wszystkich robotów. Te wspólne właściwości wprowadzone są tylko raz dla przypadku ogólnego i będą używane przez wszystkie typy robotów budowanych w przyszłości.

Kompozycja
: Roboty wytwarzające pizzę są w rzeczywistości kolekcjami komponentów, które działają zespołowo. Aby nasz przykładowy robot był skuteczny, powinien zapewne posiadać ramiona

do zawijania ciasta, silniki do manewrowania przy piecu itd. W „mowie obiektowej" nasz robot jest przykładem kompozycji: zawiera on inne obiekty, które uaktywnia do wykonywania ich zadań. Każdy komponent może być zakodowany jako klasa, definiująca swoje własne zachowanie i zależności.

Oczywiście większość z nas nie zarabia na budowaniu robotów do wytwarzania pizzy, lecz ogólne pomysły obiektowego programowania, takie jak np. dziedziczenie i kompozycja, można zastosować w dowolnej aplikacji, która da się rozłożyć na zbiór obiektów. Na przykład typowe systemy interfejsu graficznego są tworzone w postaci zbiorów widżetów (przyciski, etykiety itd.), które są wszystkie rysowane, gdy rysowany jest zawierający je pojemnik (kompozycja). Oprócz tego można samodzielnie utworzyć własne widżety, które będą specjalizowanymi wersjami elementów interfejsu o bardziej ogólnych cechach (dziedziczenie).

Rozważając to wszystko z perspektywy konkretnego programowania, można stwierdzić, że klasy są jednostką programu języka Python, podobnie jak funkcje i moduły. Stanowią one inną przestrzeń do umieszczania logiki i danych programu. W rzeczywistości klasy definiują także nową przestrzeń nazw, bardzo podobnie do modułów. W porównaniu z innymi składnikami programu, które były do tej pory omawiane, klasy różnią się trzema cechami o podstawowym znaczeniu, dzięki którym można ich używać podczas budowy nowych obiektów:

Istnienie wielu egzemplarzy
: W przybliżeniu klasy można określić jako wzorce służące do generowania obiektów. Przy każdym wywołaniu klasy tworzony jest nowy obiekt, posiadający oddzielną przestrzeń nazw. Jak się przekonamy, każdy obiekt wygenerowany z klasy ma dostęp do atrybutów tej klasy i otrzymuje swoją własną przestrzeń nazw, w której będą przechowywane dane specyficzne dla tego obiektu.

Przystosowanie do specyficznych potrzeb za pomocą dziedziczenia
: Klasy obsługują także dziedziczenie w sensie obiektowym; można je rozszerzać za pomocą modyfikacji ich atrybutów poza samą klasą. Mówiąc bardziej ogólnie, klasy mogą tworzyć hierarchie przestrzeni nazw, w których definiują nazwy przeznaczone do użytku w obiektach utworzonych z klas w danej hierarchii.

Przeciążanie operatora
: Dzięki specjalnym metodom protokołowym klasy mogą definiować obiekty odpowiadające tym rodzajom działań, które były już pokazane podczas działania na typach wbudowanych. Na przykład obiekty utworzone za pomocą klas mogą być cięte, łączone, indeksowane itd. Jak się wkrótce przekonamy, w Pythonie możliwe są różne sztuczki, dzięki którym klasy mogą przejmować każdą operację na typach wbudowanych.

Podstawy działania klas

Czytelnikowi, nie mającemu wcześniej styczności z programowaniem obiektowym, klasy mogą się wydawać skomplikowane, szczególnie gdy wszystkie informacje na ich temat są przekazane w jednej dawce. Aby ułatwić zrozumienie tych zagadnień, rozpoczniemy więc od krótkiego pokazu klas w działaniu, podkreślając trzy wyróżniające je cechy, o których była mowa poprzednio. Klasy języka Python w swojej podstawowej postaci są łatwe do zrozumienia, zaś szczegółami zajmiemy się już za chwilę.

Klasy tworzą wiele egzemplarzy obiektów

Jak już wspomnieliśmy pod koniec rozdziału 5., poświęconego modułom, klasy są przeważnie przestrzeniami nazw, podobnie jak moduły. W odróżnieniu od modułów, w klasach występuje także tworzenie wielokrotnych kopii, dziedziczenie przestrzeni nazw i przeciążanie operatorów. Zajmiemy się teraz pierwszym z tych zagadnień.

Aby zrozumieć mechanizm tworzenia wielokrotnych kopii, należy sobie uświadomić, że istnieją dwa rodzaje obiektów w modelu programowania obiektowego języka Python: obiekty klasy i egzemplarze obiektów. Obiekty klasy zapewniają domyślne zachowanie i służą jako generatory dla egzemplarzy obiektów. Egzemplarze obiektów są rzeczywistymi obiektami, które są przetwarzane przez programy. Każdy z tych egzemplarzy jest przestrzenią nazw, rządzącą się własnymi prawami, lecz dziedziczy nazwy z klasy (czyli ma do nich dostęp), z której powstał. Obiekty klasy pochodzą z instrukcji, zaś egzemplarze obiektów pochodzą z wywołań. Przy każdym wywołaniu klasy uzyskuje się nowy egzemplarz. W tym momencie należy się skoncentrować, ponieważ przedstawimy najważniejsze zagadnienia, dotyczące programowania obiektowego w języku Python.

Obiekty klasy zapewniają domyślne zachowanie

Instrukcja class tworzy obiekt klasy i przypisuje mu nazwę
: Podobnie jak `def`, instrukcja `class` języka Python jest instrukcją wykonywalną. Gdy zostanie ona uruchomiona, wtedy utworzy nowy obiekt klasy i przypisze mu nazwę podaną w nagłówku `class`.

Przypisania wewnątrz instrukcji class tworzą atrybuty klasy
: Podobnie jak w modułach, przypisania w instrukcji `class` tworzą atrybuty w obiekcie klasy. Atrybuty klasy są dostępne za pomocą kwalifikacji nazw (`obiekt.nazwa`).

Atrybuty klasy eksportują stan i zachowanie obiektu
: Atrybuty obiektu klasy przechowują informację o jego stanie i zachowaniu, które mają być współdzielone przez wszystkie egzemplarze obiektów utworzonych w danej klasie. Instrukcje `def` wewnątrz instrukcji `class` generują *metody* przetwarzające egzemplarze.

Egzemplarze obiektów są generowane z klas

Wywołanie obiektu klasy (podobnie jak wywołuje się funkcję) tworzy nowy egzemplarz obiektu
: Przy każdym wywołaniu klasa generuje i zwraca nowy egzemplarz obiektu.

Każdy egzemplarz obiektu dziedziczy atrybuty klasy i otrzymuje swoją własną przestrzeń nazw
: Egzemplarze obiektów generowane z klas są nowymi przestrzeniami nazw. Początkowo są one puste, ale dziedziczą atrybuty obiektu klasy, z której zostały wygenerowane.

Przypisania do siebie w metodach tworzą atrybuty danego egzemplarza
: Pierwszy argument w funkcjach metody klasy (nazywany umownie `self`) odwołuje się do przetwarzanego egzemplarza obiektu. Przypisania do atrybutów `self` tworzą lub modyfikują dane w egzemplarzu, a nie w klasie.

Przykład

Pomijając kilka szczegółów, to już wszystko, co dotyczy obiektowości języka Python. Zajmiemy się teraz przykładem, ponieważ chcemy pokazać praktyczne zastosowanie tych idei. Najpierw zdefiniujemy klasę nazwaną `FirstClass`, używając instrukcji `class`:

```
>>> class FirstClass:                  # definiowanie obiektu klasy
...     def setdata(self, value):      # definiowanie metod klasy
...         self.data = value          # self jest egzemplarzem
...     def display(self):
...         print self.data            # self.data (dla danego egzemplarza)
```

Podobnie jak inne instrukcje złożone, instrukcja `class` rozpoczyna się od wiersza *nagłówkowego*, który zawiera nazwę klasy. Po wierszu nagłówkowym następuje *ciało*, składające się z zagnieżdżonych i wciętych instrukcji. W naszym przykładzie zagnieżdżone są instrukcje `def`, definiujące funkcje. Funkcje te określają zachowanie, które klasa ma eksportować. Jak już opisywaliśmy, instrukcja `def` jest przypisaniem, które w tym przypadku przypisuje wartości do nazw w zakresie instrukcji `class` i w ten sposób generuje atrybuty klasy. Funkcje wewnątrz klasy są zwykle nazywane *metodami*. Są to zwyczajne funkcje, określane za pomocą instrukcji `def`, chociaż przy wywołaniu ich pierwszy argument automatycznie otrzymuje egzemplarz obiektu dla którego metoda została wywołana. Oto para przykładowych egzemplarzy:

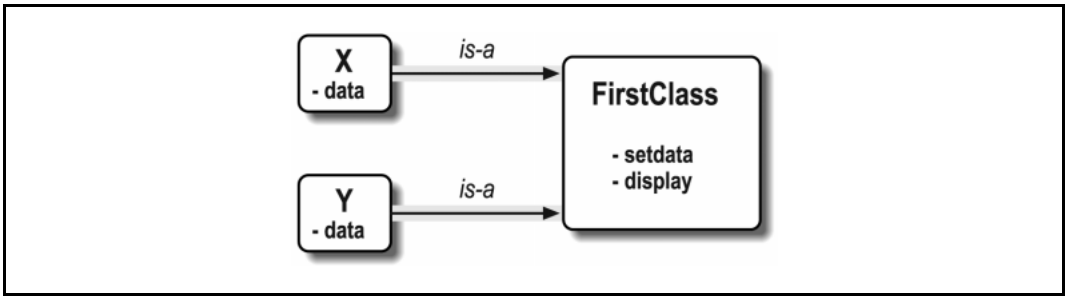
```
>>> x = FirstClass()                # tworzenie dwóch egzemplarzy
>>> y = FirstClass()                # każdy jest nową przestrzenią nazw
```

Za pomocą takiego wywołania klasy generujemy egzemplarze obiektów, które są po prostu przestrzeniami nazw, otrzymującymi od razu atrybuty klasy. Mówiąc poprawnie: mamy tu trzy obiekty — dwa egzemplarze i klasę — lecz w rzeczywistości mamy trzy powiązane *przestrzenie nazw* (patrz rysunek 6.1). Posługując się terminologią programowania obiektowego, mówimy, że `x` jest klasy `FirstClass`, podobnie jak `y`. Egzemplarze są początkowo puste, lecz mają powiązania z klasą; jeśli kwalifikujemy egzemplarz za pomocą nazwy atrybutu w obiekcie klasy, to Python pobiera nazwę z klasy (chyba że ta nazwa już znajduje się w egzemplarzu):

```
>>> x.setdata("King Arthur")  # wywołanie metody: self jest x lub y
>>> y.setdata(3.14159)        # uruchomienie: FirstClass.setdata(y, 3.14159)
```

Ani `x`, ani `y` nie mają własnej metody `setdata`; jeśli atrybut nie istnieje w egzemplarzu, Python śledzi powiązanie z egzemplarza do klasy. I to jest prawie wszystko, co chcieliśmy tu pokazać na temat *dziedziczenia* w języku Python: występuje ono w czasie kwalifikacji atrybutu i powoduje wyszukiwanie nazw w powiązanych obiektach (poprzez śledzenie powiązań `is-a`, jak to pokazano na rysunku 6.1).

Wartość przekazana w funkcji `setdata` klasy `FirstClass` jest przypisywana do `self.data`. W ramach danej metody `self` automatycznie odwołuje się do egzemplarza, który jest przetwarzany (`x lub y`), a więc przypisania zachowują wartości w przestrzeniach nazw egzemplarzy, a nie klasy (wyjaśnia to, w jaki sposób powstały nazwy `data` pokazane na rysunku 6.1). Klasy generują wiele egzemplarzy, zatem metody muszą korzystać z argumentu `self`, by uzyskać dostęp do egzemplarza, który ma być przetwarzany. Gdy wywołamy metodę `display` klasy, chcąc wydrukować `self.data`, to przekonamy się, że jest ona różna w każdym egzemplarzu. Z drugiej strony, `display` jest takie samo w `x` i `y`, ponieważ pochodzi (jest *dziedziczone*) z klasy:



Rysunek 6.1. Klasy i egzemplarze są powiązanymi obiektami przestrzeni nazw

```
>>> x.display()                  # self.data różni się w każdym egzemplarzu
King Arthur
>>> y.display()
3.14159
```

Zachowaliśmy tu obiekty różnego typu w członku `data` (łańcuch i liczbę zmiennoprzecinkową). Podobnie jak w innych sytuacjach, Python nie potrzebuje deklaracji typu dla atrybutów egzemplarza (zwanych czasem *członkami*). Atrybuty są powoływane do istnienia dopiero po pierwszym przypisaniu im wartości, podobnie jak w przypadku prostych zmiennych. W rzeczywistości atrybuty egzemplarza można zmieniać albo w samej klasie za pomocą przypisania do `self` w metodach, albo poza klasą za pomocą przypisania do istniejącego egzemplarza obiektu:

```
>>> x.data = "Nowe dane"          # można pobrać lub ustawić atrybuty
>>> x.display()                   # także na zewnątrz klasy
Nowe dane
```

Specjalizacja klas według dziedziczenia

W odróżnieniu od modułów, klasy pozwalają na modyfikację także za pomocą wprowadzenia nowych komponentów (*klas podrzędnych*), a nie tylko za pomocą modyfikacji istniejących komponentów na miejscu. Pokazaliśmy już, że egzemplarze obiektów wygenerowane z klasy dziedziczą jej atrybuty. Python pozwala również, by klasy dziedziczyły z innych klas. Toruje to drogę dla nowych struktur zwanych *osnowami*. Są to hierarchie klas, które specjalizują swoje zachowanie za pomocą zastępowania atrybutów w klasach leżących niżej w hierarchii. Główne idee całej tej maszynerii są następujące:

W nagłówku klasy podawane są w nawiasach klasy nadrzędne
: Aby dziedziczenie atrybutów z innej klasy było możliwe, należy po prostu wpisać ją w nawiasach w nagłówku instrukcji klasy. Klasa, która dziedziczy, jest nazywana *klasą podrzędną*, zaś klasa, z której pochodzi dziedziczenie, jest *klasą nadrzędną*.

Klasy dziedziczą atrybuty ze swoich klas nadrzędnych
: Podobnie jak w przypadku egzemplarza, klasa otrzymuje natychmiast wszystkie nazwy zdefiniowane w swojej klasie nadrzędnej. Nazwy te są znajdowane przez Python automatycznie podczas kwalifikacji, jeśli nie istnieją w klasie podrzędnej.

Egzemplarze dziedziczą atrybuty ze wszystkich dostępnych klas
Egzemplarze otrzymują nazwy z klasy, z której są generowane, oraz ze wszystkich klas nadrzędnych tej klasy. Podczas wyszukiwania nazwy Python sprawdza egzemplarz, następnie jego klasę, a później jej wszystkie klasy nadrzędne.

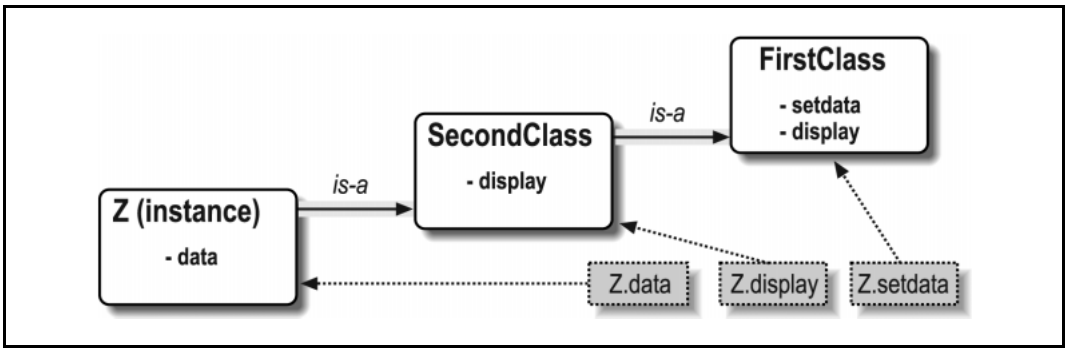
Zmiany działania są realizowane za pomocą wprowadzania klas podrzędnych, a nie przez modyfikacje klas nadrzędnych
Po ponownym zdefiniowaniu w klasach podrzędnych nazw występujących w klasach nadrzędnych, klasy podrzędne *zastępują* odziedziczone środowisko nowym.

Przykład

Nasz kolejny przykład korzysta z poprzedniego. Zdefiniujmy teraz nową klasę `SecondClass`, która dziedziczy wszystkie nazwy z `FirstClass` i dodaje swoją własną nazwę:

```
>>> class SecondClass(FirstClass):                # dziedziczy setdata
...     def display(self):                        # zmienia display
...         print 'Obecne dane = "%s"' % self.data
```

`SecondClass` definiuje ponownie metodę `display`, by używać innego formatu wyświetlania. Ponieważ `SecondClass` definiuje atrybut o tej samej nazwie, to zastępuje nim atrybut `display` w klasie `FirstClass`. Proces dziedziczenia rozpoczyna od przeszukiwania egzemplarzy, następnie obiektu klasy i klas nadrzędnych i zatrzymuje się po pierwszym wystąpieniu szukanej nazwy atrybutu. Ponieważ nazwa `display` zostanie znaleziona w `SecondClass` przed jej znalezieniem w `FirstClass`, mówimy, że `SecondClass` zastępuje `display` z `FirstClass` swoim własnym atrybutem. Innymi słowy, `SecondClass` dokonuje specjalizacji `FirstClass`, zmieniając działanie metody `display`. Z drugiej strony, `SecondClass` (i utworzone z niej egzemplarze) nadal dziedziczą dosłownie metodę `setdata` z `FirstClass`. Na rysunku 6.2 pokazano użyte tutaj przestrzenie nazw. Utwórzmy teraz przykładowy egzemplarz:



Rysunek 6.2. Specjalizacja wprowadzona za pomocą zastąpienia dziedziczonych nazw

```
>>> z = SecondClass()
>>> z.setdata(42)              # setdata znalezione w FirstClass
>>> z.display()                # znajduje zastąpioną metodę w SecondClass
Obecne dane = "42"
```

Podobnie jak poprzednio, tworzymy egzemplarz obiektu klasy `SecondClass`, wywołując ją. Wywołanie `setdata` nadal działa na wersji z `FirstClass`, lecz tym razem atrybut `display` pochodzi od `SecondClass` i wyświetla inny komunikat. Tutaj bardzo ważna informacja na temat programowania obiektowego: specjalizacja wprowadzona w `SecondClass` jest całkowicie zewnętrzna w stosunku do `FirstClass` i nie ma wpływu na istniejące lub przyszłe obiekty `FirstClass`, takie jak obiekt `x` z poprzedniego przykładu:

```
>>> x.display()     # x jest nadal egzemplarzem FirstClass (stary komunikat)
Nowe dane
```

Oczywiście jest to sztuczny przykład, lecz regułą jest, że często klasy obsługują rozszerzenia i ponowne użycie elementów lepiej, niż to czynią funkcje lub moduły. Wynika to właśnie stąd, że zmiany mogą być dokonywane na elementach zewnętrznych (w klasach podrzędnych).

Klasy mogą przesłaniać operatory języka Python

Na zakończenie tego wykładu spójrzmy na trzecią główną właściwość klas: przeciążanie operatorów. Mówiąc prosto, dzięki przeciążaniu operatorów obiekty wprowadzone za pomocą klas mogą być poddawane takim operacjom, którym podlegają wbudowane typy: dodawaniu, wycinaniu, drukowaniu, kwalifikacji itd. Można wprawdzie wprowadzić całe zachowanie obiektu, stosując funkcje metod, lecz przeciążanie operatorów pozwala na większą integrację tych obiektów z modelem obiektowym języka Python. I więcej: przeciążanie operatorów powoduje, że samodzielnie utworzone obiekty zachowują się tak, jak wbudowane, zatem sprzyja to rozwojowi interfejsów obiektowych, które są bardziej zwarte i łatwiejsze do opanowania. A oto główne wnioski wynikające z tych rozważań:

Metody o nazwach takich jak __X__ są specjalnymi punktami zaczepienia
: Przeciążanie operatorów w języku Python odbywa się za pomocą specjalnie nazwanych metod, które przesłaniają operacje.

Takie metody są wywoływane automatycznie, gdy język Python rozwija operatory
: Jeśli jakiś obiekt dziedziczy np. metodę `__add__`, to jest ona wywoływana wtedy, gdy ten obiekt pojawi się w wyrażeniu `+`.

Klasy mogą zastępować większość wbudowanych operacji na typach
: Istnieją dziesiątki nazw metod operatorów specjalnych, które mogą przechwytywać prawie wszystkie operacje dostępne dla typów wbudowanych.

Operatory pozwalają na integrację klas z modelem obiektowym języka Python
: Dzięki przeciążaniu operacji na typach, obiekty zdefiniowane przez użytkownika i wprowadzone za pomocą klas zachowują się tak, jak obiekty wbudowane.

Przykład

Przejdźmy teraz do innego przykładu. Definiujemy klasę podrzędną w stosunku do `SecondClass`, która wprowadza trzy atrybuty specjalne: `__init__`, wywoływane podczas tworzenia nowego egzemplarza obiektu (`self` jest nowym obiektem z `ThirdClass`) oraz `__add__` i `__mul__`, wywoływane wtedy, gdy egzemplarz z `ThirdClass` pojawia się odpowiednio w wyrażeniach `+` i `*`:

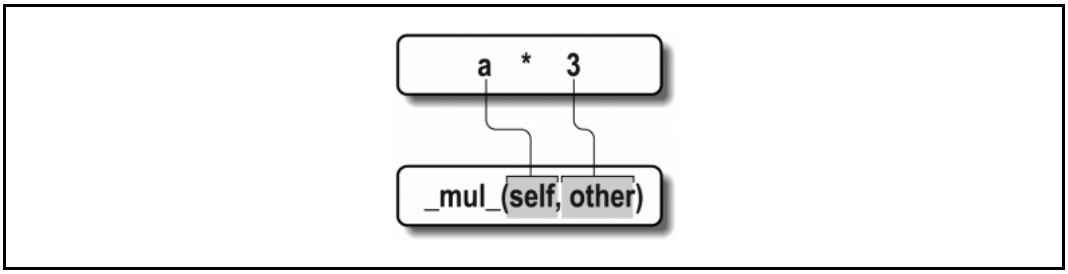
```
>>> class ThirdClass(SecondClass):          # is-a SecondClass
...     def __init__(self, value):          # działa na "ThirdClass(value)"
...         self.data = value
...     def __add__(self, other):           # działa na "self + other"
...         return ThirdClass(self.data + other)
...     def __mul__(self, other):
...         self.data = self.data * other   # działa na "self * other"

>>> a = ThirdClass("abc")  # nowe wywołanie __init__
>>> a.display()            # metoda odziedziczona
Obecne dane = "abc"

>>> b = a + 'xyz'          # nowe wywołanie __add__: tworzy nowy egzemplarz
>>> b.display()
Obecne dane = "abcxyz"

>>> a * 3                  # nowe wywołanie __mul__: zmienia egzemplarz
>>> a.display()
Obecne dane = "abcabcabc"
```

`Klasa ThirdClass` jest klasy `SecondClass`, zatem jej egzemplarze dziedziczą `display` z `SecondClass`. Wywołanie generujące egzemplarz klasy `ThirdClass` przekazuje następnie argument (`"abc"`). Jest on przekazywany do argumentu `value` w konstruktorze `__init__` i przypisywany tam do `self.data`. Potem obiekty `ThirdClass` mogą pokazywać się w wyrażeniach `+` i `*`. Python przekazuje egzemplarz obiektu z lewej strony do argumentu `self`, zaś wartość z prawej strony do `other` (patrz rysunek 6.3).



Rysunek 6.3. Odwzorowanie operatorów w metodach specjalnych

Metody specjalne, takie jak `__init__` i `__add__`, są dziedziczone przez klasy podrzędne i egzemplarze, podobnie jak inne nazwy przypisane w instrukcji klasy. Zauważmy, że metoda `__add__` tworzy *nowy* obiekt (wywołując `ThirdClass` z wynikową wartością), zaś `__mul__` modyfikuje aktualny egzemplarz obiektu w miejscu (ponownie przypisując atrybut `self`). Operator `*` tworzy nowy obiekt, gdy zastosuje się go do typów wbudowanych, takich jak liczby i listy, lecz w obiektach klasy można go interpretować dowolnie[1].

[1] Raczej nie należy tego robić (jeden z recenzentów doszedł aż do tego, że nazwał ten przykład „diabelstwo”!). Powszechna praktyka mówi, że operatory przeciążone powinny działać w taki sam sposób, jak działają operatory wbudowane. W naszym przypadku oznacza to, że metoda `__mul__` powinna zwrócić w wyniku raczej *nowy* obiekt, niż zmieniać w miejscu jego egzemplarz (`self`). Być może w lepszym stylu byłoby użycie metody `mul`, a nie przeciążanie `*` (np. `mul(3)` zamiast `* 3`). Z drugiej strony, praktyki stosowane przez jednych, nie muszą obowiązywać innych.

Używanie instrukcji class

Czy wszystko to, co napisaliśmy powyżej, ma sens? Jeśli nie, to nie należy się martwić; po tym krótkim wprowadzeniu zagłębimy się bardziej w szczegóły omawianych zagadnień. Pokazaliśmy już wprawdzie instrukcję `class` w początkowych przykładach, lecz teraz zrobimy to od strony bardziej formalnej. Podobnie jak w języku C++, instrukcja `class` jest także głównym narzędziem programowania obiektowego w języku Python. Ale w odróżnieniu od C++, `class` nie jest jednak deklaracją. Jest ona, tak samo jak `def` i `class`, budowniczym obiektów i ukrytym przypisaniem. Po uruchomieniu generuje ona obiekt klasy i zachowuje odwołanie do tego obiektu w nazwie użytej w nagłówku.

Postać ogólna

Pokazaliśmy już, że `class` jest instrukcją złożoną, której ciało tworzą wcięte instrukcje umieszczone pod wierszem nagłówkowym. Klasy nadrzędne w nagłówku są podawane w nawiasach, występują po nazwie definiowanej klasy i są oddzielone przecinkami. Wpisanie więcej niż jednej klasy nadrzędnej prowadzi do wielokrotnego dziedziczenia (wyjaśnimy to później w tym rozdziale):

```
class <nazwa>(superclass,...):          # przypisanie do nazwy
    data = value                        # współdzielone dane klasy
    def method(self,...):               # metody
        self.member = value             # dane egzemplarzy
```

W instrukcji klasy specjalnie nazwane metody przeciążają operatory; np. podczas tworzenia egzemplarza obiektu wywoływana jest funkcja o nazwie `__init__`, jeśli została zdefiniowana.

Przykład

Na początku tego rozdziału wspomnieliśmy, że klasy są przeważnie *przestrzeniami nazw* — narzędziem służącym do definiowania nazw (zwanych atrybutami), które eksportują dane i działania do klientów. Zatem w jaki sposób dostajemy się z instrukcji do przestrzeni nazw?

Podobnie jak w modułach, instrukcja zagnieżdżona w treści instrukcji `class` tworzy atrybuty. Gdy Python uruchamia instrukcję `class` (a nie wywołanie klasy), to uruchamia kolejno wszystkie jej instrukcje umieszczone w jej ciele, poczynając od góry. Przypisania, które następują w czasie tego procesu, tworzą nazwy w lokalnym zakresie klasy, zaś te nazwy stają się atrybutami w powiązanym obiekcie klasy. Klasy są zatem podobne zarówno do modułów, jak i do funkcji, ponieważ:

- Instrukcje `class` tworzą zakres lokalny, czyli miejsce, gdzie istnieją nazwy utworzone przez zagnieżdżone przypisania (podobnie jak w funkcjach).
- Nazwy przypisane w instrukcji `class` stają się atrybutami w obiekcie klasy (podobnie, jak w modułach).

Głównym wyróżnikiem klas jest fakt, że ich przestrzenie nazw są także podstawą dziedziczenia w języku Python; atrybuty są pobierane z innych klas tylko wtedy, gdy nie są znalezione w danej klasie lub w egzemplarzu obiektu. Ponieważ `class` jest instrukcją złożoną, to w jej ciele może

być zagnieżdżona instrukcja dowolnego rodzaju, np. `print`, `=`, `if` lub `def`. Pokazaliśmy już, że zagnieżdżona instrukcja `def` tworzy metodę klasy — inne przypisania także tworzą atrybuty. Załóżmy, że uruchamiamy następującą przykładową klasę:

```
class Subclass(aSuperclass):          # definicja klasy podrzędnej
    data = 'mielonka'                 # przypisanie atrybutu klasy
    def __init__(self, value):        # przypisanie atrybutu klasy
        self.data = value             # przypisanie atrybutu egzemplarza
    def display(self):
        print self.data, Subclass.data      # egzemplarz, klasa
```

Ta klasa zawiera dwie instrukcje `def`, które dowiązują atrybuty klasy do funkcji metod. Zawiera ona także instrukcję przypisania (=); ponieważ zaś nazwa `data` jest przypisana wewnątrz instrukcji `class`, to istnieje w lokalnym zakresie klasy i staje się atrybutem obiektu tej klasy. Podobnie jak wszystkie pozostałe atrybuty klasy, atrybut `data` jest dziedziczony i *współdzielony* przez wszystkie egzemplarze klasy[2]:

```
>>> x = Subclass(1)           # tworzy dwa egzemplarze obiektów
>>> y = Subclass(2)           # każdy ma swoje własne dane ("data")
>>> x.display(); y.display() # "self.data" inne, "Subclass.data" takie same
1 mielonka
2 mielonka
```

Gdy uruchamiamy ten kod, nazwa `data` istnieje w dwóch miejscach: w egzemplarzach obiektów (utworzonych za pomocą konstruktora `__init__`) i w klasie, z której są dziedziczone nazwy (utworzonej za pomocą przypisania =). Metoda klasy `display` wyświetla obydwie wersje, najpierw za pomocą kwalifikacji samego egzemplarza, a następnie za pomocą samej klasy. Ponieważ klasy są obiektami posiadającymi atrybuty, to ich nazwy można uzyskać za pomocą kwalifikacji nawet wtedy, gdy nie został wprowadzony egzemplarz.

Zastosowania metod klasy

Zapoznaliśmy już Czytelników z funkcjami, zatem możemy powiedzieć, że metody klasy również zostały opisane. Metody są po prostu obiektami funkcji tworzonymi przez instrukcję `def`, zagnieżdżoną w treści instrukcji `class`. Patrząc na to abstrakcyjnie, metody określają zachowania egzemplarzy obiektów, które mają być dziedziczone. Z punktu widzenia programisty należy rozumieć to tak, że metody działają dokładnie w taki sam sposób, jak proste funkcje, ale z jednym ważnym wyjątkiem: ich pierwszy argument zawsze otrzymuje egzemplarz obiektu, który jest wynikowym przedmiotem wywołania metody. Innymi słowy, język Python automatycznie odwzorowuje wywołania metod egzemplarza na funkcje metody danej klasy:

```
egzemplarz.metoda(args...) => staje się => klasa.metoda(egzemplarz, args...)
```

[2] Osoby znające język C++ mogą to potraktować jako coś zbliżonego do oznaczenia `static` dla danych klasy, czyli elementów (członków), które są przechowywane w klasie niezależnie od egzemplarzy. W języku Python nie jest to niczym specjalnym: wszystkie atrybuty klasy są po prostu nazwami przypisanymi w instrukcji `class` bez względu na to, czy odwołują się do funkcji (*metody* języka C++), czy do czegoś innego (*członkowie* języka C++).

Klasa jest tu określana za pomocą procedury wyszukiwania, użytej w procesie dziedziczenia języka Python. Pierwszy argument specjalny w metodzie klasy bywa zazwyczaj nazywany zgodnie z konwencją `self`. Jest on podobny do wskaźnika `this` języka C++, lecz metody języka Python zawsze muszą kwalifikować argument `self` jawnie, aby można było pobrać lub zmienić atrybuty egzemplarza, który jest przetwarzany przez bieżące wywołanie metody.

Przykład

Powrócimy teraz do przykładu i zdefiniujemy następującą klasę:

```
class NextClass:                    # definiowanie klasy
    def printer(self, text):        # definiowanie metody
        print text
```

Nazwa `printer` odnosi się do obiektu funkcji. Jest ona przypisana w zakresie instrukcji `class`, zatem staje się atrybutem klasy i jest dziedziczona przez każdy egzemplarz utworzony z tej klasy. Funkcja `printer` może być wywołana na dwa sposoby: przez egzemplarz lub przez samą klasę:

```
>>> x = NextClass()               # tworzenie egzemplarza
>>> x.printer('Hello world!')     # wywołanie jego metody
Hello world!
```

Po wywołaniu za pomocą kwalifikacji *egzemplarza* (jak powyżej), argument `self` metody `printer` ma automatycznie przypisany egzemplarz obiektu (`x`), zaś argument `text` otrzymuje łańcuch przekazany przy wywołaniu (`"Hello world!"`). Wewnątrz metody `printer` argument `self` może mieć dostęp lub ustawić dane egzemplarza, ponieważ odwołuje się on do egzemplarza aktualnie przetwarzanego. Można także wywołać `printer`, przechodząc przez klasę za pomocą jawnego przekazania egzemplarza do argumentu `self`:

```
>>> NextClass.printer(x, 'Hello world!')   # metoda klasy
Hello world!
```

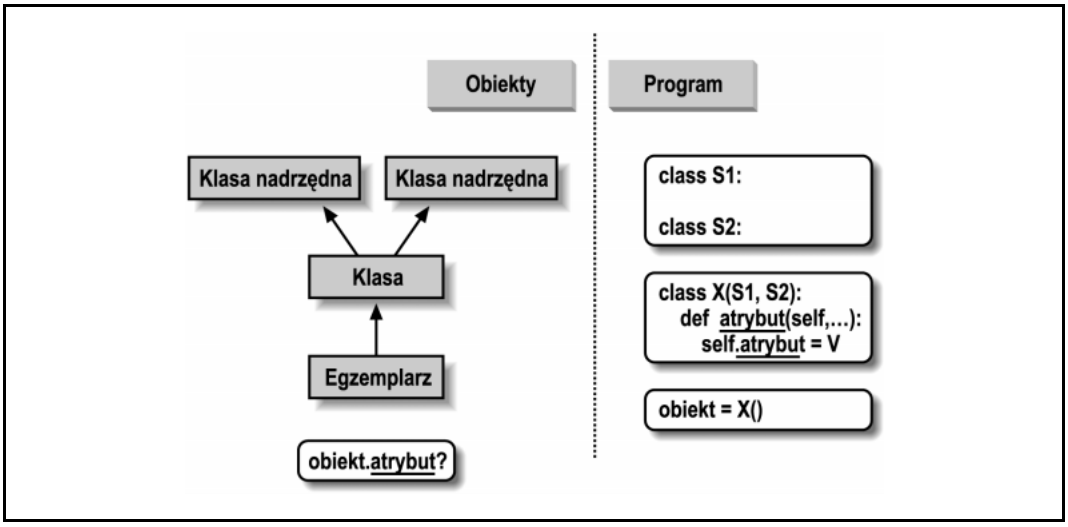
Wywołania skierowane przez egzemplarz i klasę mają dokładnie taki sam skutek, pod warunkiem, że przy przejściu poprzez klasę przekazywany jest ten sam egzemplarz obiektu. Za chwilę pokażemy, że wywołania przez klasę są podstawą rozszerzania (zamiast zastępowania) dziedziczonych zachowań.

Dziedziczenie i przeszukiwanie drzew przestrzeni nazw

Cały sens instrukcji `class`, działającej na przestrzeni nazw, polega na obsłudze dziedziczenia tych nazw. W języku Python dziedziczenie zachodzi wtedy, gdy obiekt jest kwalifikowany. Polega ono na przeszukiwaniu drzewa definicji atrybutu (jednej lub wielu przestrzeni nazw). Za każdym razem, gdy używa się wyrażenia o postaci `obiekt.atrybut`, w którym `obiekt` jest egzemplarzem lub obiektem klasy, Python przeszukuje drzewo przestrzeni nazw na poziomie, na którym znajduje się `obiekt`, i wyżej. Szukany jest `atrybut` (jego pierwsze wystąpienie). Ponieważ definicje znajdujące się na niższych poziomach w drzewie zastępują definicje z wyższych poziomów, dziedziczenie stwarza podstawy do specjalizacji.

Budowa drzewa atrybutów

Na rysunku 6.4 pokazano sposób, w jaki zbudowane są drzewa przestrzeni nazw. Ogólnie mówiąc:



Rysunek 6.4. Budowa drzewa przestrzeni nazw i dziedziczenie

- Atrybuty *egzemplarzy* są generowane za pomocą przypisań do atrybutów `self` w metodach.
- Atrybuty *klasy* są tworzone za pomocą instrukcji (przypisań) w instrukcjach `class`.
- Dowiązania *klas nadrzędnych* są tworzone za pomocą wpisania klas w nawiasach w nagłówku instrukcji `class`.

Ogólnym wynikiem tych działań jest drzewo przestrzeni nazw atrybutów, które u podstawy ma egzemplarz, potem przechodzi do klasy, z której został wygenerowany ten egzemplarz, a następnie rozrasta się na wszystkie klasy nadrzędne podane w nagłówku tej klasy. Python przeszukuje to drzewo w kierunku do góry, rozpoczynając od poziomu egzemplarza. Przeszukiwanie odbywa się zawsze, gdy do pobrania nazwy atrybutu z egzemplarza obiektu używana jest kwalifikacja[3].

Specjalizacja metod dziedziczonych

Opisany wyżej model dziedziczenia, którego podstawą jest przeszukiwanie drzewa, daje dobrą okazję do specjalizacji systemów. Nazwy w procesie dziedziczenia są szukane najpierw w klasach podrzędnych, a następnie w nadrzędnych, zatem klasy podrzędne mogą *zastępować* swoje domyślne zachowania, definiując ponownie atrybuty klas nadrzędnych. Można zatem budować całe systemy jako hierarchie klas, które są rozszerzane poprzez dodawania nowych zewnętrznych klas podrzędnych, a nie poprzez zmiany istniejących powiązań logicznych.

[3] Ten opis nie jest kompletny, ponieważ atrybuty egzemplarza i klasy mogą być tworzone także za pomocą przypisania do obiektów na zewnątrz instrukcji `class`. Jest to jednak rzadziej spotykane i czasem powoduje więcej błędów (zmiany nie są ograniczone do instrukcji `class`). W języku Python wszystkie atrybuty są zawsze dostępne domyślnie; o prywatności powiemy później w tym rozdziale.

Idea przeciążania nazw dziedziczonych prowadzi do różnorodnych technik specjalizacyjnych. Klasy podrzędne mogą np. całkowicie zastępować nazwy dziedziczone, pod warunkiem, że dostarczają nazw oczekiwanych przez klasy nadrzędne. Mogą one również rozszerzać metody klas nadrzędnych za pomocą ponownych wywołań tych klas z zastąpionej metody. Pokazaliśmy już zamianę w działaniu, oto zatem przykład pokazujący działanie rozszerzeń:

```
>>> class Super:
...     def method(self):
...         print 'in Super.method'
...
>>> class Sub(Super):
...     def method(self):                  # zastąpienie metody
...         print 'starting Sub.method'    # tu dodanie działań
...         Super.method(self)             # startuje działanie domyślne
...         print 'ending Sub.method'
...
```

Bezpośrednie wywołanie metody klasy nadrzędnej jest w tym miejscu sednem sprawy. Klasa `Sub` zamienia funkcję `method` klasy `Super` na swoją własną wyspecjalizowaną wersję. W ramach tej zamiany `Sub` wywołuje ponownie wersję wyeksportowaną przez `Super`, aby uzyskać zachowanie domyślne. Innymi słowy, `Sub.method` po prostu raczej rozszerza zachowanie `Super.method`, a nie całkowicie je zamienia:

```
>>> x = Super()          # tworzenie egzemplarza Super
>>> x.method()           # uruchomienie Super.method
in Super.method

>>> x = Sub()            # tworzenie egzemplarza Sub
>>> x.method()           # uruchamienie Sub.method, wywołującego Super.method
starting Sub.method
in Super.method
ending Sub.method
```

Rozszerzanie jest zwykle stosowane razem z konstruktorami. Metoda o specjalnej nazwie `__init__` jest nazwą dziedziczoną i jako *jedyna* jest znaleziona i uruchomiona, gdy tworzony jest egzemplarz. W celu uruchomienia konstruktorów klasy nadrzędnej, metoda `__init__` klasy podrzędnej powinna wywołać metodę `__init__` klasy nadrzędnej za pomocą kwalifikacji klas (np. `Class.__init__(self,...)`).

Rozszerzanie jest jedynym sposobem tworzenia interfejsu do klasy nadrzędnej. Poniżej pokazano klasy podrzędne, które ilustrują ten podstawowy schemat:

- `Super` definiuje funkcje `method` i `delegate`, z których ostatnia oczekuje na działania (`action`) w klasie podrzędnej.
- `Inheritor` nie dostarcza żadnych nowych nazw, a więc pobiera wszystko to, co jest zdefiniowane w `Super`.
- `Replacer` zastępuje metodę (`method`) w `Super` swoją własną wersją metody.
- `Extender` przystosowuje metodę (`method`) klasy `Super`, zastępując ją i ponownie wywołując w celu uruchomienia domyślnego zachowania.
- `Provider` wprowadza metodę `action`, na którą oczekuje metoda `delegate` klasy `Super`.

```
class Super:
    def method(self):
        print 'in Super.method'      # domyślne
    def delegate(self):
        self.action()                # oczekiwane

class Inheritor(Super):
    pass

class Replacer(Super):
    def method(self):
        print 'in Replacer.method'

class Extender(Super):
    def method(self):
        print 'starting Extender.method'
        Super.method(self)
        print 'ending Extender.method'

class Provider(Super):
    def action(self):
        print 'in Provider.action'

if __name__ == '__main__':
    for klasa in (Inheritor, Replacer, Extender):
        print '\n' + klasa.__name__ + '...'
        klasa().method()
    print '\nProvider...'
    Provider().delegate()
```

W tym miejscu warto zwrócić uwagę na kilka rzeczy: kod samotestujący na końcu tego przykładu tworzy egzemplarze trzech różnych klas. Ponieważ klasy są obiektami, można je wstawiać do krotki i tworzyć egzemplarze w sposób podstawowy (ten temat omówimy później). Klasy mają także specjalny atrybut `__name__`, jak moduły — jest to po prostu wstępnie ustawiony łańcuch, zawierający nazwę podaną w nagłówku klasy. Gdy wywołujemy metodę `delegate` poprzez egzemplarz obiektu `Provider`, język Python znajduje metodę `action` w `Provider` za pomocą zwykłego przeszukania drzewa: wewnątrz metody `delegate w klasie Super`, `self` odwołuje się do egzemplarza `Provider`.

```
% python specialize.py

Inheritor...
in Super.method

Replacer...
in Replacer.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Przeciążanie operatorów w klasach

Przeciążanie operatorów opisaliśmy pobieżnie na początku tego rozdziału; tutaj uzupełnimy informacje i pokażemy powszechnie stosowane i przydatne metody przeciążania. Oto lista kluczowych wniosków, dotyczących przeciążania:

- Przeciążanie operatora pozwala klasom na przechwytywanie działania normalnych operacji języka Python.
- Klasy mogą przeciążać wszystkie operatory wyrażeń języka Python.
- Klasy mogą także przeciążać operacje na obiektach: drukowanie, wywołania, kwalifikację itd.
- Przeciążanie powoduje, że egzemplarze klas działają podobnie, jak wbudowane typy.
- Przeciążanie jest wprowadzane za pomocą dostarczenia specjalnie nazwanych metod klasy.

Oto prosty przykład działającego przeciążania. Jeśli w klasie zapewnimy specjalnie nazwane metody, to Python automatycznie wywoła je, gdy egzemplarze klasy pojawią się w odpowiednich operacjach. Na przykład pokazana niżej klasa `Number` dostarcza metodę przesłaniania konstruktora egzemplarza (`__init__`) oraz metodę przesłaniania operacji odejmowania (`__sub__`). Metody specjalne stanowią punkt zaczepienia, który pozwala utworzyć powiązanie z operacjami wbudowanymi:

```
class Number:
    def __init__(self, start):             # w Number(start)
        self.data = start
    def __sub__(self, other):              # w egzemplarz - other
        return Number(self.data - other)   # wynik w nowym egzemplarzu

>>> from number import Number              # pobiera klasę z modułu
>>> X = Number(5)                          # wywołuje Number.__init__(X, 5)
>>> Y = X - 2                              # wywołuje Number.__sub__(X, 2)
>>> Y.data
3
```

Podstawowe metody przeciążania operatorów

Prawie wszystko to, co można zrobić z obiektami wbudowanymi, takimi jak liczby całkowite i listy, ma swoje odpowiedniki w specjalnie nazwanych metodach, służących do przeciążania w klasach. W tabeli 6.1 podano najczęściej spotykane i użyteczne metody specjalne; jest ich znacznie więcej, ale w książce mamy na nie za mało miejsca. Pełnej listy dostępnych metod o specjalnych nazwach należy szukać w innych książkach na temat języka Python lub w podręczniku systemowym do biblioteki Pythona. Nazwy wszystkich metod przeciążających rozpoczynają się i kończą dwoma znakami podkreślenia (po to, aby odróżnić je od innych nazw definiowanych w klasach).

Przykłady

Pokażemy teraz kilka metod z tabeli 6.1 na przykładach.

Tabela 6.1. Wybór metod przeciążania operatora

Metoda	Przeciąża	Wywołanie
`__init__`	Konstruktor	Tworzenie obiektu: `Class()`
`__del__`	Destruktor	Zwolnienie obiektu
`__add__`	Operator '+'	`X + Y`
`__or__`	Operator '|' (bitowe `or`)	`X | Y`
`__repr__`	Drukowanie, przekształcenia	`print X`, `` `X` ``
`__call__`	Wywołania funkcji	`X()`
`__getattr__`	Kwalifikacja	`X.niezdefiniowany`
`__getitem__`	Indeksowanie	`X[klucz]`, w pętli `for`, w testach
`__setitem__`	Przypisanie indeksowane	`X[klucz] = wartość`
`__getslice__`	Wycinanie	`X[dolny:górny]`
`__len__`	Długość (rozmiar)	`len(X)`, testy prawdziwości
`__cmp__`	Porównanie	`X == Y`, `X < Y`
`__radd__`	Prawostronny operator '+'	Nieegzemplarz `+ X`

__getitem__ przesłania wszystkie odwołania indeksowe

Metoda `__getitem__` przesłania operacje indeksowania egzemplarza: gdy egzemplarz `X` pojawia się w wyrażeniu indeksowym, takim jak np. `X[i]`, Python wywołuje metodę `__getitem__` odziedziczoną przez egzemplarz (jeśli taka istnieje), przekazując `X` do pierwszego argumentu, zaś indeks w nawiasach kwadratowych do drugiego argumentu. Na przykład poniższa klasa zwraca kwadraty wartości indeksu:

```
>>> class indexer:
...     def __getitem__(self, index):
...         return index ** 2
...
>>> X = indexer()
>>> for i in range(5):
...     print X[i],                  # X[i] wywołuje __getitem__(X, i)
...
0 1 4 9 16
```

Pokażemy teraz specjalną sztuczkę, która nie zawsze jest rozumiana przez początkujących, chociaż bywa bardzo użyteczna: wprowadzając instrukcję `for` w rozdziale 3., wspomnieliśmy, że działa ona w sposób powtarzalny, indeksując sekwencję od zera do większych wartości indeksów aż do przekroczenia granic indeksu. Dlatego właśnie `__getitem__` można wykorzystać do przeciążania iteracji i testów zawierania. Jest to działanie w myśl hasła „kup jeden, drugi dostaniesz za darmo”: dowolny obiekt wbudowany lub obiekt zdefiniowany przez użytkownika, który reaguje na indeksowanie, reaguje także automatycznie na iterację i test zawierania:

```
>>> class stepper:
...     def __getitem__(self, i):
...         return self.data[i]
...
>>> X = stepper()                  # X jest obiektem kroczącym
```

```
>>> X.data = "Mielonka"
>>>
>>> for item in X:                # pętle for wywołują __getitem__
...     print item,               # dla pozycji indeksowych 0..N
...
M i e l o n k a
>>>
>>> 'l' in X                      # operator 'in' także wywołuje __getitem__
1
```

__getattr__ wychwytuje niezdefiniowane odwołania atrybutu

Metoda `__getattr__` przesłania kwalifikacje atrybutów. Mówiąc dokładniej, jest ona wywoływana z nazwą atrybutu, podaną jako łańcuch wszędzie tam, gdzie następuje próba kwalifikacji egzemplarza za pomocą niezdefiniowanej (nie istniejącej) nazwy atrybutu. Nie jest ona wywoływana, jeśli Python może znaleźć atrybut, używając swojej procedury przeszukiwania drzewa dziedziczenia. Z powodu takiego zachowania `__getattr__` jest przydatne jako punkt zaczepienia w ogólnych odpowiedziach na żądania atrybutów. Na przykład:

```
>>> class empty:
...     def __getattr__(self, attrname):
...         if attrname == "age":
...             return 36
...         else:
...             raise AttributeError, attrname
...
>>> X = empty()
>>> X.age
36
>>> X.name
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 6, in __getattr__
AttributeError: name
```

W powyższym przykładzie klasa `empty` i jej egzemplarz `X` nie mają swoich własnych prawdziwych atrybutów, zatem dostęp do `X.age` będzie kierowany do metody `__getattr__`. Do `self` jest przypisany egzemplarz (`X`), zaś do `attrname` — łańcuch będący nazwą niezdefiniowanego atrybutu (`"age"`). Taka definicja klasy powoduje, że `age` wygląda jak prawdziwy atrybut, bowiem zwracana jest wartość wyrażenia kwalifikacyjnego `X.age` (tutaj: `36`).

Sposób obsługi innych atrybutów nie jest znany, a więc wywoływana jest wbudowana obsługa wyjątku `AttributeError`. Dzięki temu Python jest poinformowany o tym, że dana nazwa rzeczywiście nie jest zdefiniowana i zapytanie o atrybut (`X.name`) generuje błąd. Metodę `__getattr__` będziemy jeszcze omawiać przy okazji opisywania delegacji, zaś o wyjątkach powiemy więcej w rozdziale 7.

__repr__ zwraca reprezentację łańcuchową

Oto przykład działania konstruktora `__init__` i metody `__add__` przeciążającej operator `+`, który przy okazji pokazuje metodę `__repr__`, zwracającą łańcuchową reprezentację egzemplarzy. Do przekształcania przetwarzanego obiektu `self.data` na łańcuch stosowane są odwrotne apostrofy. Jeśli obiekt jest zdefiniowany, to metoda `__repr__` jest wywoływana automatycznie podczas drukowania obiektów klasy lub ich przekształcania na łańcuchy.

```
>>> class adder:
...     def __init__(self, value=0):
...         self.data = value                    # inicjowanie danych
...     def __add__(self, other):
...         self.data = self.data + other        # dodawanie other w miejscu
...     def __repr__(self):
...         return `self.data`                   # przekształcanie na łańcuch
...
>>> X = adder(1)          # __init__
>>> X + 2; X + 2          # __add__
>>> X                     # __repr__
5
```

Podaliśmy tyle przykładów przeciążania, ile zmieściło się w książce. Pozostałe metody działają podobnie do tych, które zostały tu pokazane. Wszystkie z nich służą po prostu jako punkty zaczepienia przy przesłanianiu operacji na wbudowanych typach. Niektóre metody przeciążania nie mają jednoznacznie określonej listy argumentów lub zwracanej wartości. W dalszych częściach książki pokażemy jeszcze działanie kilku z nich, lecz pełnego przeglądu tego zagadnienia należy szukać w dokumentacji.

Komplet reguł dla przestrzeni nazw

Po opisie klas i egzemplarzy obiektów mamy już komplet informacji na temat przestrzeni nazw języka Python. Zamieszczamy zatem krótkie podsumowanie wszystkich reguł stosowanych przy rozpoznawaniu nazw. Przede wszystkim należy pamiętać o tym, że nazwy kwalifikowane i nazwy niekwalifikowane są traktowane odmiennie, zaś niektóre zakresy służą do inicjacji przestrzeni nazw obiektów:

- Nazwy niekwalifikowane (`X`) korzystają z zakresów.
- Nazwy kwalifikowane (`obiekt.X`) korzystają z przestrzeni nazw obiektów.
- Zakresy inicjują przestrzenie nazw obiektów (w modułach i w klasach).

Nazwy niekwalifikowane: są globalne do momentu przypisania

Nazwy niekwalifikowane są zgodne z regułami LGB omawianymi dla funkcji w rozdziale 4.

Przypisanie: `X = wartość`
: Powoduje, że nazwa staje się nazwą lokalną: tworzy lub zmienia nazwę `X` w bieżącym zakresie lokalnym, chyba że użyto deklaracji `global`.

Odwołanie: `X`
: Poszukuje nazwy `X` w bieżącym zakresie lokalnym, następnie w bieżącym zakresie globalnym, a potem w zakresie wbudowanym.

Nazwy kwalifikowane: przestrzenie nazw obiektów

Nazwy kwalifikowane odnoszą się do atrybutów specyficznych obiektów i podlegają regułom, które przedstawiliśmy podczas omawiania modułów. Dla egzemplarzy obiektów i obiektów klasy reguły odwołań są poszerzone w taki sposób, aby zawierały procedurę przeszukiwania dziedziczenia:

Przypisanie: `object.X = wartość`
Tworzy lub zmienia nazwę atrybutu `X` w przestrzeni nazw obiektu, który jest kwalifikowany.

Odwołanie: `object.X`
Poszukuje nazwy atrybutu `X` w obiekcie, a następnie we wszystkich dostępnych klasach powyżej tego obiektu (ale nie w modułach).

Słowniki przestrzeni nazw

W rozdziale 5. pokazaliśmy, że przestrzenie nazw modułu są w rzeczywistości słownikami prezentowanymi za pomocą wbudowanego atrybutu `__dict__`. To samo dotyczy także obiektów klasy i egzemplarzy obiektów: kwalifikacja jest w rzeczywistości słownikiem indeksowanym wewnętrznie, zaś dziedziczenie atrybutów oznacza po prostu przeszukiwanie powiązanych słowników.

Poniższy przykład pokazuje sposób rozrastania się słowników przestrzeni nazw po wprowadzeniu klas. Należy tu zwrócić uwagę na następujący fakt: jeżeli atrybut `self` zostanie przypisany w jednej z dwóch klas, to utworzy on (lub zmieni) atrybut w słowniku przestrzeni nazw egzemplarza, a nie klasy. Przestrzenie nazw egzemplarzy obiektów rejestrują dane specyficzne dla tych egzemplarzy; są one także powiązane z przestrzeniami nazw tych klas, które są przeszukiwane w wyniku dziedziczenia. Na przykład `X.hello` zostaje ostatecznie znalezione w słowniku przestrzeni nazw klasy `super`.

```
>>> class super:
...     def hello(self):
...         self.data1 = "mielonka"
...
>>> class sub(super):
...     def howdy(self):
...         self.data2 = "jajka"
...
>>> X = sub()                    # tworzy nową przestrzeń nazw (słownik)
>>> X.__dict__
{}
>>> X.hello()                    # zmienia przestrzeń nazw egzemplarza
>>> X.__dict__
{'data1': 'mielonka'}

>>> X.howdy()                    # zmienia przestrzeń nazw egzemplarza
>>> X.__dict__
{'data2': 'jajka', 'data1': 'mielonka'}

>>> super.__dict__
{'hello': <function hello at 88d9b0>, '__doc__': None}

>>> sub.__dict__
{'__doc__': None, 'howdy': <function howdy at 88ea20>}

>>> X.data3 = "tost"
>>> X.__dict__
{'data3': 'tost', 'data2': 'jajka', 'data1': 'mielonka'}
```

Funkcja `dir`, którą omawialiśmy w rozdziałach 1. i 2., działa również na klasach i egzemplarzach obiektów. W rzeczywistości działa ona na wszystko to, co ma atrybuty. Wywołanie `dir(obiekt)` zwraca zatem taką samą listę, co wywołanie `obiekt.__dict__.keys()`.

Zastosowania klas w programach

Dotychczas koncentrowaliśmy się na klasie, czyli na narzędziu programowania obiektowego w języku Python. Programowanie obiektowe dotyczy również zagadnień związanych z projektowaniem, czyli tego, w jaki sposób należy używać klas w modelowaniu użytecznych obiektów. W tym podrozdziale zajmiemy się nowymi ideami z zakresu programowania obiektowego i podamy kilka przykładów bardziej zbliżonych do rzeczywistości niż te, które były prezentowane wcześniej. Większość pojęć z zakresu programowania, które będą tu omawiane, wymaga objaśnienia znacznie szerszego, niż tu możemy zapewnić. Jeśli więc ten podrozdział rozbudzi ciekawość Czytelników, sugerujemy następny etap: zapoznanie się z tekstami na temat projektowania systemów obiektowych lub ze wzorcami takich projektów.

Python i programowanie obiektowe

Programowanie obiektowe w języku Python można scharakteryzować za pomocą trzech pojęć:

Dziedziczenie
: Polega na wyszukiwaniu atrybutów w języku Python (w wyrażeniach `X.name`).

Polimorfizm
: W wyrażeniu `X.method` znaczenie `method` zależy od typu (klasy) `X`.

Kapsułkowanie
: Metody i operatory określają zachowanie; ukrywanie danych jest konwencją domyślną.

Na tym etapie Czytelnik powinien już dobrze rozumieć, czym jest dziedziczenie w języku Python. Smaczek polimorfizmu języka Python wypływa z jego braku deklaracji typów. Ponieważ atrybuty są zawsze rozpoznawane w fazie działania programu, to obiekty, które mają takie same interfejsy, są zamienne. Programy-klienci nie muszą wiedzieć, jaki rodzaj obiektu wprowadza wywoływaną metodę[4]. Kapsułkowanie oznacza w języku Python pakowanie, a nie prywatność. Prywatność jest tutaj opcją, jak to pokażemy w dalszej części tego rozdziału.

Programowanie obiektowe i dziedziczenie „jest”

Mechanizm dziedziczenia omówiliśmy już dość szczegółowo, ale chcemy pokazać przykład jego zastosowania w modelowaniu sytuacji.

Z punktu widzenia programisty dziedziczenie rozpoczyna się od kwalifikacji atrybutu i wyszukiwania nazwy w egzemplarzu, w jego klasie, a następnie w klasach nadrzędnych. Z punktu widzenia projektanta dziedziczenie jest sposobem na określenie członkostwa. Klasa definiuje zestaw właściwości, które mogą być dziedziczone przez zestawy bardziej specyficzne (np. przez klasy podrzędne).

[4] Niektóre języki programowania obiektowego definiują polimorfizm w taki sposób, że oznacza on przeciążanie funkcji na podstawie sygnatur typu ich argumentów. Ponieważ w języku Python nie ma deklaracji typów, to taki pomysł nie ma tu zastosowania. Wybór na podstawie typu można jednak zawsze zakodować za pomocą testów `if` i wbudowanej funkcji `type(X)` (na przykład `if type(X) is type(0): rozpatrzTypInteger()`).

Aby to zobrazować, przywołajmy robota wytwarzającego pizzę, o którym była mowa na początku tego rozdziału. Załóżmy, że zdecydowaliśmy się sprawdzić alternatywną ścieżkę kariery zawodowej i otwieramy restaurację serwującą pizzę. Jedną z pierwszych spraw, którą musimy załatwić, jest zatrudnienie pracowników do obsługi klientów, robienia pizzy itd. Będąc w głębi serca inżynierami, postanowiliśmy także zbudować robota, który będzie wytwarzał pizzę. Chcąc także uwzględnić poprawność polityczną i cybernetyczną, zdecydowaliśmy, że nasz robot będzie pełnokrwistym pracownikiem — z wynagrodzeniem.

Personel naszej pizzerii może być zdefiniowany w przykładowym pliku *pracownicy.py* za pomocą podanych niżej klas. Mamy tu cztery klasy oraz pewien kod samotestujący. Klasa najbardziej ogólna o nazwie `Pracownik` obejmuje zwyczajne zachowania, takie jak podwyżka wynagrodzenia (`dajPodwyzke`) czy drukowanie (`__repr__`). Istnieją dwa rodzaje pracowników i dlatego mamy dwie klasy podrzędne względem klasy `Pracownik` — `Szef` oraz `Obsluga`. Obydwie klasy zastępują odziedziczoną metodę `work` w celu drukowania bardziej specyficznych komunikatów. Oprócz tego nasz robot robiący pizzę jest modelowany za pomocą bardziej specjalizowanej klasy: `PizzaRobot` *jest* rodzaju `Szef`, który *jest* rodzaju `Pracownik`. W terminologii programowania obiektowego nazywamy te relacje dowiązaniami „*jest*” (*is-a*): robot *jest* szefem, który *jest* pracownikiem.

```
class Pracownik:
    def __init__(self, nazwa, placa=0):
        self.nazwa   = nazwa
        self.placa = placa
    def dajPodwyzke(self, procent):
        self.placa = self.placa + (self.placa * procent)
    def praca(self):
        print self.nazwa, "robi nadzienie"
    def __repr__(self):
        return "<Pracownik: nazwa=%s, placa=%s>" % (self.nazwa, self.placa)

class Szef(Pracownik):
    def __init__(self, nazwa):
        Pracownik.__init__(self, nazwa, 50000)
    def praca(self):
        print self.nazwa, "robi jedzonko"

class Obsluga(Pracownik):
    def __init__ (self, nazwa):
        Pracownik.__init__(self, nazwa, 40000)
    def praca(self):
        print self.nazwa, "obsluguje klientow"

class PizzaRobot(Szef):
    def __init__(self, nazwa):
        Szef.__init__(self, nazwa)
    def praca(self):
        print self.nazwa, "robi pizze"

if __name__ == "__main__":
    bob = PizzaRobot('bob')      # utworzenie robota o nazwie bob
    print bob                    # uruchomienie odziedziczonej __repr__
    bob.dajPodwyzke(0.20)        # podwyższenie pensji boba o 20%
    print bob; print

for klasa in Pracownik, Szef, Obsluga, PizzaRobot:
    obj = klasa(klasa.__name__)
    obj.praca()
```

Po uruchomieniu kodu samotestującego tego modułu zostanie utworzony robot o nazwie `bob` robiący pizzę, który dziedziczy nazwy z trzech klas: `PizzaRobot`, `Szef` i `Pracownik`. Na przykład drukowanie `bob` uruchamia metodę `Pracownik.__repr__`, a danie mu podwyżki wywołuje `Pracownik.dajPodwyzke`, ponieważ właśnie w tym miejscu dziedziczenie go znajduje.

```
C:\python\examples> python pracownicy.py
<Pracownik: nazwa=bob, placa=50000>
<Pracownik: nazwa=bob, placa=60000.0>

Pracownik robi nadzienie
Szef robi jedzonko
Obsluga obsluguje klientow
PizzaRobot robi pizze
```

W hierarchii klas, takiej jak w tym przykładzie, można zazwyczaj tworzyć egzemplarze z dowolnej klasy, a nie tylko z tej jednej na dole. Na przykład pętla `for` w kodzie samotestującym tego modułu tworzy egzemplarze obiektów ze wszystkich czterech klas. Każda klasa odpowiada inaczej, ponieważ metoda `praca` jest inna w każdej z nich. Klasy te po prostu symulują obiekty z prawdziwego świata: metoda `praca` drukuje komunikat o stanie w teraźniejszości, ale można ją rozszerzyć tak, aby komunikat dotyczył także przyszłości.

Programowanie obiektowe i kompozycja „ma”

O znaczeniu kompozycji wspomnieliśmy już na początku tego rozdziału. Z punktu widzenia programisty, kompozycja wprowadza osadzanie innych obiektów w obiekcie-pojemniku i ich uaktywnianie w celu uzyskania metod pojemnika. Dla projektanta kompozycja stanowi inny sposób przedstawiania relacji w danym problemie. Kompozycja, zamiast ustanawiać członkostwo, posługuje się komponentami, czyli częściami całości. Kompozycja odzwierciedla także relacje między częściami; osoby zajmujące się programowaniem obiektowym zazwyczaj nazywają to relacją „*ma*” (*has-a*).

Po wprowadzeniu naszych pracowników wrzucimy ich do pizzerii i damy im jakieś zajęcie. Pizzeria jest obiektem złożonym: *ma* piec i *ma* pracowników (obsługa i szefowie). Gdy klient składa zamówienie, komponenty pizzerii przystępują do działania: obsługa przyjmuje zamówienie, szef robi pizzę itd. Poniższy przykład symuluje wszystkie obiekty i ich relacje występujące w tym scenariuszu:

```
from pracownicy import PizzaRobot, Obsluga

class Klient:
    def __init__(self, nazwa):
        self.nazwa = nazwa
    def zamowienie(self, obsluga):
        print self.nazwa, "zamawia u", obsluga
    def placic(self, obsluga):
        print self.nazwa, "placi za pozycje do", obsluga

class Piec:
    def piecze(self):
        print "piec piecze"
class PizzaSklep:
    def __init__(self):
        self.obsluga = Obsluga('Pat')       # osadza inne obiekty
        self.szef    = PizzaRobot ('Bob')   # robot o imieniu bob
        self.piec    = Piec()

    def zamowienie(self, nazwa):
        klient = Klient(nazwa)              # uaktywnia inne obiekty
```

```
        klient.zamowienie(self.obsluga)     # zamówienia klientów u obsługi
        self.szef.praca()
        self.piec.piecze()
        klient.placic(self.obsluga)

if __name__ == "__main__":
    scena = PizzaSklep()                    # tworzenie kompozycji
    scena.zamowienie('Homer')               # symulacja zamówienia Homera
    print '...'
    scena.zamowienie('Shaggy')              # symulacja zamówienia Shaggy
```

Klasa `PizzaSklep` jest pojemnikiem i kontrolerem; jej konstruktor tworzy i osadza egzemplarze z klas pracowników, o których była mowa w ostatnim podrozdziale oraz ze zdefiniowanej wyżej klasy `Piec`. Gdy kod samotestujący tego modułu wywołuje metodę `PizzaSklep zamowienie`, obiekty osadzone wykonują kolejno swoje działania. Zauważmy, że dla każdego zamówienia tworzony jest nowy obiekt `Klient`, a do metod `Klient` przekazywany jest osadzony obiekt `Obsluga`. Klienci wchodzą i wychodzą, lecz obsługa stanowi część kompozycji pizerii. Widać także, że pracownicy są nadal zaangażowani w relacje dziedziczenia. Kompozycja i dziedziczność uzupełniają się nawzajem:

```
C:\python\examples> python pizzasklep.py
Homer zamawia u <Pracownik: nazwa=Pat, pensja=40000>
Bob robi pizze
piec piecze
Homer placi za pozycje do <Pracownik: nazwa=Pat, placa=40000>
...
Shaggy zamawia u <pracownik: nazwa=Pat, placa=40000>
Bob robi pizze
piec piecze
Shaggy placi za pozycje do <Pracownik: nazwa=Pat, placa=40000>
```

Po uruchomieniu tego modułu nasza pizzeria rozpoczyna obsługę dwóch zamówień: jednego od Homera i następnego od Shaggy. Podkreślamy, że jest to tylko zabawa z symulacją; prawdziwa pizzeria miałaby więcej części, zaś tutaj nie ma prawdziwej pizzy. Obiekty i ich stosunki reprezentują działającą kompozycję. Należy zapamiętać, że klasy mogą reprezentować prawie dowolne obiekty i relacje, które dadzą się wyrazić w mowie. Potem wystarczy po prostu zastąpić rzeczowniki klasami, a czasowniki metodami i już uzyskuje się pierwszy zarys projektu.

Programowanie obiektowe i delegacja

Programiści posługujący się językami obiektowymi mówią często o tzw. delegacji. Zazwyczaj polega to na wprowadzeniu obiektów kontrolera, które osadzają inne obiekty, do których przekazują żądania wykonania operacji. Kontrolery mogą dbać o działalność administracyjną, taką jak np. utrzymywanie ścieżki dostępu itd. W języku Python delegacja jest często wprowadzana jako punkt zaczepienia metody `__getattr__`. Ponieważ metoda ta przesłania dostęp do nieistniejących atrybutów, to klasa opakowująca może używać `__getattr__` do skierowania dowolnego dostępu do opakowanego obiektu. Oto przykład:

```
class wrapper:
    def __init__(self, object):
        self.wrapped = object                       # zapisuje obiekt
    def __getattr__(self, attrname):
        print 'Trace:', attrname                    # śledzi pobranie
        return getattr(self.wrapped, attrname)     # pobranie delegacji
```

Dlaczego należy zwracać uwagę na klasy i trwałość

Oprócz tego, że klasy użyte w przykładzie pizzerii pozwalają na symulację zachowań rzeczywistego świata, można ich także użyć jako podstawy działania trwałej bazy danych restauracji. W rozdziale 10. pokażemy, że egzemplarze klas mogą być zachowywane na dysku za pomocą modułów `pickle` lub `shelve`. Interfejs „marynujący" obiekty jest niezwykle prosty w obsłudze:

```
import pickle
obiekt = jakasKlasa()
plik   = open(nazwa_pliku, 'w')      # tworzenie pliku zewnętrznego
pickle.dump(obiekt, plik)            # zapis obiektu w pliku

plik   = open(nazwa_pliku, 'r')
obiekt = pickle.load(plik)           # odtworzenie obiektu z pliku
```

Półka (`shelve`) działa w podobny sposób, lecz automatycznie „marynuje" obiekty w bazie danych obsługiwanej za pomocą klucza:

```
import shelve
obiekt = jakasKlasa()
baza   = shelve.open('nazwa_pliku')
baza['klucz'] = obiekt          # zapis z kluczem
obiekt = baza['klucz']          # odtworzenie
```

„Marynowanie" przekształca obiekty na uszeregowane strumienie bajtów, które mogą być zapisywane w plikach, wysyłane w sieci itd. W naszym przykładzie użycie klas do modelowania pracowników oznacza, że możemy uzyskać prostą bazę pracowników i restauracji za darmo: „marynowanie" takich egzemplarzy obiektów do pliku utrwala je podczas działania programu w języku Python. Szczegóły dotyczące „marynowania" podano w rozdziale 10.

Można używać klasy `wrapper` z tego modułu w celu kontrolowania dowolnego obiektu z atrybutami: listami, słownikami, a nawet z klasami i egzemplarzami. W podanym niżej przykładzie klasa po prostu drukuje komunikat, podający wynik śledzenia każdego dostępu do atrybutu:

```
>>> from trace import wrapper
>>> x = wrapper([1,2,3])              # opakowanie listy
>>> x.append(4)                       # delegacja do metody listy
Trace: append
>>> x.wrapped                         # wydruk członka
[1, 2, 3, 4]

>>> x = wrapper({"a": 1, "b": 2})     # opakowanie słownika
>>> x.keys()                          # delegacja do metody słownika
Trace: keys
['a', 'b']
```

Rozszerzanie wbudowanych typów obiektów

Klasy są także powszechnie stosowane do rozszerzania możliwości wbudowanych typów języka Python, by dało się obsłużyć bardziej egzotyczne struktury danych. Aby np. dodać do list metody wstawiania i usuwania z kolejki, można utworzyć takie klasy, które *opakowują* (osadzają) obiekt listy i wyeksportować metody wstawiania i usuwania stosowane do przetwarzania listy.

Czy Czytelnicy pamiętają zestaw funkcji, o których pisaliśmy w rozdziale 4.? Oto jak one wyglądają po ich zamianie na klasy języka Python. W podanym niżej przykładzie wprowadziliśmy nowy typ *obiektu* zbioru, przenosząc niektóre funkcje ze wspomnianego zbioru do metod i dodając podstawowe przeciążanie operatora. Przeważnie klasa ta po prostu opakowuje listę języka Python w dodatkowy zestaw operacji, ale ponieważ jest klasą, to umożliwia także tworzenie wielu egzemplarzy i przystosowywanie do własnych potrzeb poprzez dziedziczenie w klasach podrzędnych.

```
class Set:
    def __init__(self, value = []):     # konstruktor
        self.data = []                   # zarządza listą
        self.concat(value)

    def intersect(self, other):          # other jest dowolną sekwencją
        res = []                         # self jest przedmiotem
        for x in self.data:
            if x in other:               # wskazuje wspólne pozycje
                res.append(x)
            return Set(res)              # zwraca nowy Set

    def union(self, other):              # other jest dowolną sekwencją
        res = self.data[:]               # kopia danej listy
        for x in other:                  # dodawanie pozycji do other
            if not x in res:
               res.append(x)
        return Set(res)

    def concat(self, value):             # value: list, Set...
        for x in value:                  # usuwa duplikaty
           if not x in self.data:
                self.data.append(x)
    def __len__(self):          return len(self.data)        # len(self)
    def __getitem__(self, key): return self.data[key]        # self[i]
    def __and__(self, other):   return self.intersect(other) # self & other
    def __or__(self, other):    return self.union(other)     # self | other
    def __repr__(self):         return 'Set:' + `self.data`  # print
```

Za pomocą przeciążonego indeksowania nasza klasa `Set` może często działać jako zamaskowana prawdziwa lista. Ponieważ zamierzamy poprosić Czytelników o wykonanie działań i rozszerzenie tej klasy w ćwiczeniu podanym na końcu tego rozdziału, to o tym kodzie nie będziemy więcej pisać (aż do dodatku C, w którym są podane rozwiązania ćwiczeń).

Dziedziczenie wielokrotne

Omawiając szczegóły związane z instrukcją `class`, wspomnieliśmy, że w wierszu nagłówkowym można wpisać w nawiasach więcej niż jedną klasę nadrzędną. Gdy tak zrobimy, to używamy czegoś, co jest nazwane *wielokrotnym dziedziczeniem*. Klasa i jej egzemplarze dziedziczą nazwy ze wszystkich wpisanych klas nadrzędnych. Podczas wyszukiwania atrybutu Python szuka klas nadrzędnych w wierszu nagłówkowym danej klasy, począwszy od lewej strony. Wyszukiwanie trwa aż do momentu dopasowania nazw. Wyszukiwanie odbywa się najpierw w głąb, a następnie przenosi się od lewej do prawej — wynika to z faktu, że każda klasa nadrzędna może mieć swoje własne klasy nadrzędne.

Teoretycznie wielokrotne dziedziczenie jest dobre do modelowania obiektów, które należą do więcej niż jednego zbioru. Na przykład dana osoba może być inżynierem, pisarzem, muzykiem itp. i może dziedziczyć właściwości wszystkich tych zbiorów. Jednak w praktyce wielokrotne dziedziczenie jest

bardzo zaawansowanym narzędziem i może stać się zbyt skomplikowane, jeśli się go nadużywa. Pokażemy to ponownie jako niespodziankę pod koniec tego rozdziału. Podobnie jak wszystko inne w programowaniu: jest to narzędzie przydatne, gdy się z niego właściwie korzysta.

Jednym z najczęściej spotykanych zastosowań dziedziczenia wielokrotnego jest „domieszka" metod ogólnego przeznaczenia, pochodzących z klas nadrzędnych. Takie klasy nadrzędne są zwykle zwane klasami *domieszkowymi* (*mixin*). Dostarczają one metod, które można dodawać do klas aplikacji za pomocą dziedziczenia. Na przykład domyślny sposób języka Python na drukowanie egzemplarza obiektu klasy jest niezbyt użyteczny:

```
>>> class Mielonka:
...     def __init__(self):              # brak __repr__
...         self.data1 = "jedzenie"
...
>>> X = Mielonka()
>>> print X                              # format domyślny: klasa, adres
<Mielonka instance at 87f1b0>
```

Jak już pokazywaliśmy w poprzednim podrozdziale, dotyczącym przeciążania operatorów, można zastosować metodę `__repr__`, wprowadzając dzięki niej przystosowaną do własnych potrzeb reprezentację łańcucha. Zamiast jednak kodować `__repr__` w każdej klasie, którą chcemy wydrukować, dlaczego nie zakodować jej tylko raz w klasie narzędzi ogólnego przeznaczenia, by potem móc ją dziedziczyć we wszystkich klasach?

Do tego właśnie służą klasy domieszkowe. Poniższy kod definiuje klasę domieszkową o nazwie `Lister`, która przeciąża metodę `__repr__` w każdej klasie podanej w jej wierszu nagłówkowym. Skanuje ona po prostu słownik atrybutów egzemplarza (który, jak wiadomo, jest eksportowany w `__dict__`) w celu zbudowania łańcucha składającego się z nazw i wartości wszystkich atrybutów egzemplarza. Ponieważ klasy są obiektami, to mechanizm formatujący `Lister` może być używany w odniesieniu do egzemplarzy dowolnej klasy podrzędnej. Jest to zatem narzędzie podstawowe.

`Lister` korzysta z dwóch specjalnych sztuczek do zdobycia nazwy klasy i adresu egzemplarza. Egzemplarze mają wbudowany atrybut `__class__`, odwołujący się do klasy, z której był utworzony egzemplarz, zaś klasy mają atrybut `__name__`, który jest nazwą podaną w nagłówku. Zatem wyrażenie `self.__class__.__name__` pobiera nazwę klasy egzemplarza. Adres komórki pamięci, pod którym rezyduje egzemplarz, uzyskujemy za pomocą wywołania wbudowanej funkcji `id`, zwracającej adres dowolnego obiektu:

```
# Klasa Lister może być wmieszana do dowolnej klasy
# w zakresie zapewnienia formatowanego wydruku egzemplarzy
# za pomocą dziedziczenia zakodowanej tu metody __repr__;
# self jest egzemplarzem najniższej klasy;

class Lister:
   def __repr__(self):
       return ("<Instance of %s, address %s:\n%s>" %
                        (self.__class__.__name__,     # nazwa mojej klasy
                         id(self),                    # mój adres
                         self.attrnames()) )          # lista nazwa=wartość
   def attrnames(self):
       result = ''
       for attr in self.__dict__.keys():   # skanuje słownik egzemplarza
           if attr[:2] == '__':
               result = result + "\tname %s=<built-in>\n" % attr
```

```
            else:
                result = result + "\tname %s=%s\n" % (attr,
                ↳self.__dict__[attr])
        return result
```

Teraz klasa `Lister` jest przydatna dla każdej tworzonej klasy — nawet dla tych klas, które już mają klasę nadrzędną. Oto właśnie przykład przydatności wielokrotnego dziedziczenia: dodając `Lister` do listy klas nadrzędnych w nagłówku danej klasy otrzymujemy bez przeszkód odpowiednią metodę `__repr__`, podczas gdy nadal dziedziczymy z istniejącej klasy nadrzędnej:

```
from mytools import Lister       # otrzymanie klasy narzędzi

class Super:
    def __init__(self):          # superklasa __init__
       self.data1 = "mielonka"

class Sub(Super, Lister):          # wmieszanie __repr__
    def __init__(self):            # Lister ma dostęp do self
        Super.__init__(self)
        self.data2 = "jajka"       # więcej attr przykładu
        self.data3 = 42

if __name__ == "__main__":
    X = Sub()
    print X                        # wmieszane repr
```

Tutaj klasa `Sub` dziedziczy nazwy zarówno z klasy `Super`, jak i z `Lister`; jest więc mieszanką swoich własnych nazw i nazw jej obydwu klas nadrzędnych. Gdy tworzymy egzemplarz klasy `Sub` i drukujemy go, to uzyskujemy reprezentację pochodzącą z klasy `Lister`:

```
C:\python\examples> python testmixin.py
<Instance of Sub, address 135423340:
         name data2=jajka
         name data3=42
         name data1=mielonka
>
```

Klasa `Lister` działa w dowolnej klasie, w którą zostanie wmieszana, ponieważ `self` odnosi się do egzemplarza klasy podrzędnej, która „wciąga" klasę `Lister`, bez względu na to, co nią może być. Jeśli później zdecydujemy się na rozszerzenie `__repr__` w klasie `Lister` tak, aby drukować także atrybuty klasy, które dziedziczy egzemplarz, to jesteśmy bezpieczni. Ponieważ jest to metoda odziedziczona, to modyfikacja `__repr__` w klasie `Lister` aktualizuje zarazem każdą klasę podrzędną, w którą jest ona wmieszana[5]. Klasy domieszkowe są w pewnym sensie klasowym ekwiwalentem modułów. Oto klasa `Lister`, działająca w trybie pojedynczego dziedziczenia w egzemplarzach różnych klas. Widać tu wyraźnie, że w programowaniu obiektowym można powtórnie używać kodu:

[5] Dla ciekawego Czytelnika mamy informację, że klasy mają także wbudowany atrybut zwany `__bases__`, który jest krotką obiektów klas nadrzędnych danej klasy. Wyświetlarka lub przeglądarka hierarchii klas ogólnego przeznaczenia może rekurencyjnie przechodzić od `__class__` egzemplarza do jego klasy, a następnie od `__bases__` klasy do wszystkich klas nadrzędnych. Ten pomysł pokażemy jeszcze w ćwiczeniu, ale oprócz tego warto zapoznać się z innymi książkami na ten temat lub podręcznikiem systemowym języka Python. Można tam znaleźć więcej szczegółów na temat specjalnych atrybutów obiektu.

```
>>> from mytools import Lister
>>> class x(Lister):
...     pass
...
>>> t = x()
>>> t.a = 1; t.b = 2; t.c = 3
>>> t
<Instance of x, address 7797696:
         name b=2
         name a=1
         name c=3
>
```

Klasy są obiektami: fabryki obiektów

Klasy są obiektami, zatem można łatwo je przekazywać poza program, przechowywać w strukturach danych itp. Można także przekazywać klasy do funkcji, które generują dowolne rodzaje obiektów. W kręgach projektantów zajmujących się programowaniem obiektowym takie funkcje są czasem nazywane *fabrykami*. Wymagają one wiele wysiłku w językach ze ścisłą kontrolą typów (np. w C++), lecz w języku Python są wręcz trywialne: funkcja `apply`, którą opisywaliśmy w rozdziale 4., może w pojedynczym kroku wywołać dowolną klasę z argumentem, by wygenerować dowolny rodzaj egzemplarza[6]:

```
def factory(aClass, *args):                 # krotka varargs
    return apply(aClass, args)              # wywołanie aClass

class Mielonka:
    def doit(self, message):
        print message

class Osoba:
    def __init__(self, nazwa, praca):
         self.nazwa = nazwa
         self.praca = praca

object1 = factory(Mielonka)                 # utworzenie Mielonka
object2 = factory(Osoba, "Guido", "guru")   # utworzenie Osoba
```

W powyższym przykładzie zdefiniowano funkcję generatora obiektów, zwaną `factory`. Oczekuje ona na przekazanie obiektu klasy (dowolnej klasy) łącznie z jakimiś argumentami dla konstruktora klasy. Funkcja korzysta z metody `apply` do wywołania funkcji i zwrotu jej egzemplarza. Pozostała część przykładu to po prostu definicje dwóch klas i generacja ich egzemplarzy za pomocą przekazania ich do funkcji `factory`. Jest to jedyna funkcja `factory`, którą, w razie potrzeby, Czytelnik będzie musiał napisać w języku Python. Działa ona dla każdej klasy i dla wszelkich argumentów konstruktora. Możliwe jest jeszcze tylko jedno ulepszenie: aby obsługiwać argumenty w postaci słów kluczowych w wywołaniach konstruktora, funkcja `factory` może pobierać je za pomocą argumentu `**args` i następnie przekazywać do `apply` jako trzeci argument:

```
def factory(aClass, *args, **kwargs):       # słownik kwargs
    return apply(aClass, args, kwargs)      # wywołanie aClass
```

[6] Rzeczywiście, `apply` może wywołać *każdy* obiekt wywoływalny; dotyczy to funkcji, klasy i metody. Funkcja `factory` może tutaj uruchamiać dowolny wywoływalny element, a nie tylko klasę (niezależnie od nazwy argumentu).

Czytelnicy powinni już wiedzieć, że w języku Python wszystko jest „obiektem", nawet klasy. W językach takich jak C++ stanowią one zaledwie dane wejściowe dla kompilatora. W języku Python tylko obiekty wywodzące się z klas są obiektami w sensie programowania obiektowego, zatem nie jest możliwe dziedziczenie w obiektach nie wywodzących się z klas (czyli takich jak np. listy i liczby), chyba że zostaną one opakowane w klasy.

Metody są obiektami: związane lub niezwiązane

Mówiąc o obiektach, zdajemy sobie sprawę, że metody są także rodzajem obiektu, bardzo podobnie jak funkcje. Ponieważ metody klasy mogą być dostępne albo z egzemplarza, albo z klasy, to w rzeczywistości prowadzi to do dwóch subtelnych smaczków języka Python:

Niezwiązane metody klasy: brak self
: Dostęp do atrybutu funkcji klasy za pomocą kwalifikowania klasy zwraca *niezwiązany obiekt metody*. Aby go wywołać, trzeba w sposób jawny podać egzemplarz obiektu dokładnie taki, jaki jest jego pierwszy argument.

Związane metody egzemplarza: self + pary funkcji
: Dostęp do atrybutu funkcji klasy za pomocą kwalifikacji egzemplarza zwraca *związany obiekt metody*. Język Python automatycznie opakowuje egzemplarz w funkcję w związanej metodzie obiektu, zatem nie trzeba przekazywać egzemplarza, by wywołać metodę.

Obydwa rodzaje metod są obiektami pełnej krwi: mogą one być przekazywane gdziekolwiek, zachowywane w listach itd. Obydwa wymagają również przekazania egzemplarza w swoim pierwszym argumencie, gdy są uruchamiane (np. wartość dla `self`), ale dostarczany jest on automatycznie przez język Python, gdy wywołujemy związaną metodę z egzemplarza. Zdefiniujmy np. następującą klasę:

```
class Mielonka:
    def doit(self, komunikat):
        print komunikat
```

Możemy teraz utworzyć egzemplarz i pobrać metodę związaną bez jej faktycznego wywoływania. Kwalifikacja `object.name` jest wyrażeniem obiektowym; tutaj zwraca obiekt związanej metody, który opakowuje egzemplarz (`object1`) w funkcję metody (`Mielonka.doit`). Można przypisać metodę związaną do innej nazwy i wywołać ją tak, jakby była prostą funkcją:

```
object1 = Mielonka()
x = object1.doit               # obiekt metody związanej
x('hello world')               # wprowadzono egzemplarz
```

Z drugiej strony, jeśli kwalifikujemy klasę, aby dojść do `doit`, to ponownie otrzymujemy obiekt metody niezwiązanej, który po prostu jest odwołaniem do obiektu funkcji. Aby wywołać taki rodzaj metody, należy przekazać egzemplarz jako skrajny lewy argument:

```
t = Mielonka.doit              # obiekt metody niezwiązanej
t(object1, 'howdy')            # przekazanie egzemplarza
```

W większości przypadków wywołujemy metody natychmiast po pobraniu ich za pomocą kwalifikacji (tj. `self.attr(args)`), zatem nie zawsze dostrzegamy po drodze obiekt metody. Jeśli jednak rozpoczniemy pisanie kodu, który wywołuje obiekty w sposób podstawowy, to musimy uważać, aby traktować w sposób specjalny metody niezwiązane: wymagają one jawnego przekazania obiektu.

Informacje dodatkowe

Atrybuty prywatne (nowość w wersji 1.5)

W ostatnim rozdziale wspomnieliśmy, że każda nazwa przypisana na górnym poziomie pliku jest eksportowana przez moduł. Domyślnie to samo odnosi się do klas: ukrywanie danych jest konwencją i programy-klienci mogą pobierać lub zmieniać dowolną klasę lub atrybut egzemplarza. W rzeczywistości wszystkie atrybuty są publiczne (`public`) i wirtualne (`virtual`), posługując się terminologią języka C++. Wszystkie one są dostępne zewsząd i wszystkie są wyszukiwane dynamicznie w fazie działania programu.

Tak było przynajmniej do wersji 1.5 języka Python. W wersji 1.5 Guido wprowadził pojęcie *przekręcania nazw* (*name mangling*) w celu uczynienia lokalnymi niektórych nazw w klasach. Prywatne nazwy są cechą zaawansowaną, w pełni opcjonalną i prawdopodobnie nie będą zbyt użyteczne, aż do momentu tworzenia dużych hierarchii klas. Poniżej podajemy jednak przegląd informacji dla ciekawych.

W języku Python w wersji 1.5 nazwy wewnątrz instrukcji `class`, które rozpoczynają się od dwóch podkreśleń (i nie kończą się dwoma podkreśleniami), są automatycznie zmieniane tak, aby zawierały nazwę otaczającej ją klasy. Na przykład nazwa taka jak `__X` w klasie `Class` jest automatycznie zmieniana na `_Class__X`. Jest to czymś niezwykłym, ponieważ zmodyfikowana nazwa zawiera nazwę otaczającej ją klasy. Dzięki temu nie będzie dochodzić do konfliktu z podobnymi nazwami w innych klasach w danej hierarchii.

Język Python przekręca nazwy w klasie wszędzie tam, gdzie one występują. Na przykład atrybut egzemplarza o nazwie `self.__X` jest przekształcany na `self._Class__X`, zatem przekręcanie nazw dotyczy również atrybutów *egzemplarzy* obiektów. Ponieważ więcej niż jedna klasa może dodawać atrybuty do egzemplarza, to przekręcanie nazw automatycznie pomaga uniknąć sprzeczności.

Przekręcanie nazw odbywa się tylko w instrukcjach `class` i dotyczy tylko tych nazw, w których występują dwa początkowe znaki podkreślenia. Z tego powodu kod może stać się niezbyt czytelny. Nie oznacza to jednak tego samego, co deklaracje `private` w języku C++ (jeśli Czytelnik zna nazwę dołączonej klasy, to nadal może pobrać przekręcone atrybuty!). Pozwala to jednak uniknąć przypadkowych sprzeczności nazw, gdy nazwa atrybutu jest używana przez więcej niż jedną klasę w hierarchii.

Łańcuchy dokumentacyjne

Po przedstawieniu klas możemy już powiedzieć, do czego służą atrybuty `__doc__`, o których wcześniej wspominaliśmy. W celu opisania tworzonego kodu używaliśmy dotychczas komentarzy, które rozpoczynają się od znaku `#`. Komentarze są przydatne dla osób czytających programy, lecz nie będą dostępne podczas pracy programu. Na szczęście Python umożliwia powiązanie *łańcuchów* dokumentacyjnych z obiektami jednostek programowych, wykorzystując do tego celu specjalną składnię. Jeśli plik modułowy, instrukcja `def` lub instrukcja `class` rozpoczynają się od stałej łańcuchowej, a nie od instrukcji, to Python wstawia łańcuch do atrybutu `__doc__` generowanego obiektu. Pokazany niżej program definiuje np. łańcuchy dokumentacyjne dla wielu obiektów:

```
"Ja jestem: docstr.__doc__"

class mielonka:
    "Ja jestem: mielonka.__doc__ albo docstr.mielonka.__doc__"

    def method(self, arg):
        "Ja jestem: mielonka.method.__doc__ albo self.method.__doc__"
        pass

def func(args):
    "Ja jestem: docstr.func.__doc__"
    pass
```

Główną zaletą łańcuchów dokumentacyjnych jest to, że są one dołączane w fazie działania programu. Po zakodowaniu tekstu jako łańcucha dokumentacyjnego można kwalifikować obiekt, aby pobrać jego dokumentację.

```
>>> import docstr
>>> docstr.__doc__
'Ja jestem: docstr.__doc__'
>>> docstr.mielonka.__doc__
'Ja jestem: mielonka.__doc__ albo docstr.mielonka.__doc__'
>>> docstr.mielonka.method.__doc__
'Ja jestem: mielonka.method.__doc__ albo self.method.__doc__'
>>> docstr.func.__doc__
'Ja jestem: docstr.func.__doc__'
```

Takie działanie bywa przydatne, zwłaszcza w fazie tworzenia programu. Można dzięki temu odszukać dokumentację komponentów z poziomu interakcyjnego wiersza poleceń (jak to pokazano wyżej) bez konieczności otwierania pliku źródłowego i czytania komentarzy opatrzonych znakiem #. W podobny sposób może z tego skorzystać przeglądarka obiektów języka Python, wyświetlając opisy łącznie z obiektami.

Niestety, łańcuchy dokumentacyjne nie są powszechnie używane przez programistów języka Python. Aby odnieść jak największe korzyści, programiści muszą tworzyć dokumentację zgodnie z pewną konwencją. Nasze doświadczenie mówi, że w praktyce konwencje tego rodzaju są rzadko stosowane. Oprócz tego łańcuchy dokumentacyjne są wprawdzie dostępne w fazie działania programu, lecz także mniej elastyczne niż komentarze ze znakiem # (które mogą pojawiać się w dowolnych miejscach programu). Obydwie postacie są przydatne i *każda* dokumentacja programu jest dobra... pod warunkiem, że jest dobrze napisana.

Porównanie klas i modułów

Na zakończenie rozważań ogólnych cofniemy się nieco i porównamy tematy ostatnich dwóch rozdziałów — moduły i klasy. Obydwa twory są przestrzeniami nazw, zatem ich rozróżnienie może być czasem kłopotliwe. W skrócie:

Moduły

- Są pakietami danych i działań.
- Są tworzone w postaci plików języka Python lub rozszerzeń języka C.
- Są używane poprzez importowanie.

Klasy

- Wprowadzają nowe obiekty.
- Są tworzone za pomocą instrukcji klas.
- Są używane poprzez wywoływania.
- Zawsze istnieją w module.

Klasy mają także dodatkowe właściwości, których nie mają moduły, np. przeciążanie operatorów, tworzenie wielu egzemplarzy oraz dziedziczenie. Pomijając fakt, że obydwa te składniki języka są przestrzeniami nazw, Czytelnik chyba może już teraz stwierdzić, iż są one różnymi tworami.

Niespodzianki w klasach

Większość problemów z klasami sprowadza się zwykle do problemów z przestrzeniami nazw (ma to sens, bowiem klasy są po prostu przestrzeniami nazw, które kryją w rękawach kilka dodatkowych sztuczek).

Zmiana atrybutów klasy miewa skutki uboczne

Według teorii wszystkie klasy (i egzemplarze klasy) są obiektami zmiennymi. Podobnie jak wbudowane listy i słowniki, mogą one być modyfikowane na miejscu — za pomocą przypisania do ich atrybutów. Podobnie jak w przypadku list i słowników, oznacza to także, że modyfikacja klasy lub egzemplarza obiektu może wpłynąć na wiele odwołań do nich.

W zasadzie tego właśnie chcemy (ogólnie mówiąc, w taki właśnie sposób obiekty zmieniają swój stan), lecz znajomość tego zagadnienia staje się sprawą szczególnie ważną podczas modyfikacji atrybutów klasy. Ponieważ wszystkie egzemplarze obiektów generowane z klasy współdzielą przestrzeń nazw tej klasy, to każda modyfikacja na poziomie klasy wpływa na wszystkie egzemplarze, jeśli nie mają one swoich własnych wersji zmienianych atrybutów klasy.

Ponieważ wszystkie klasy, moduły i egzemplarze są po prostu obiektami posiadającymi przestrzenie nazw atrybutów, to w fazie działania programu można normalnie zmieniać ich atrybuty za pomocą przypisań. Rozważmy teraz zdefiniowaną niżej klasę. Przypisanie do nazwy wewnątrz ciała tej klasy generuje atrybut `X.a`, który istnieje w obiekcie klasy w fazie działania programu i będzie dziedziczony przez wszystkie egzemplarze `X`:

```
>>> class X:
...     a = 1          # atrybut klasy
...
>>> I = X()
>>> I.a               # odziedziczony przez egzemplarz
1
>>> X.a
1
```

Wszystko do tego momentu idzie całkiem nieźle. Zwróćmy jednak uwagę na to, co się dzieje, gdy zmieniamy atrybut klasy w sposób dynamiczny: zmienia się on także w każdym obiekcie, który dziedziczy z klasy. Oprócz tego nowe egzemplarze utworzone z klasy uzyskują wartość ustawianą dynamicznie, bez względu na to, co nakazuje kod źródłowy klasy:

```
>>> X.a = 2          # może zmienić coś więcej niż X
>>> I.a              # I zmienia się również
2
>>> J = X()          # J dziedziczy z wartości X w fazie działania programu
>>> J.a              # (ale przypisanie do J.a zmienia a w J, nie w X lub w I)
2
```

Rozwiązanie

Użyteczna cecha czy niebezpieczna pułapka? Trzeba to rozsądzić samemu, chociaż całe zadanie można wykonać za pomocą zmiany atrybutów klasy bez tworzenia pojedynczego egzemplarza. W rzeczywistości ten mechanizm może symulować „rekordy" lub „struktury" używane w innych językach. Rozpatrzmy jako przykład niezwykły lecz poprawny program w języku Python:

```
class X: pass                # tworzenie kilku przestrzeni nazw atrybutu
class Y: pass

X.a = 1                      # użycie atrybutów klasy jako zmiennych
X.b = 2                      # nie można nigdzie znaleźć egzemplarzy
X.c = 3
Y.a = X.a + X.b + X.c

for X.i in range(Y.a): print X.i     # wydruk 0..5
```

Klasy `X` i `Y` działają tutaj tak, jak moduły bezplikowe — przestrzenie nazw do przechowywania zmiennych, których nie chcemy utracić. Jest to sztuczka całkowicie dopuszczalna we własnych programach w języku Python, lecz wydaje się mniej odpowiednia w zastosowaniu do klas utworzonych przez innych programistów. Nie zawsze bowiem można mieć pewność, że atrybuty klasy, które sami modyfikujemy, nie mają specjalnego znaczenia dla wewnętrznego zachowania się klasy. Jeśli ktoś zamierza zasymulować strukturę języka C (czyli *struct*), to zamiast zmieniać klasy, powinien modyfikować egzemplarze, ponieważ wtedy dotyczy to tylko jednego obiektu:

```
>>> class Record: pass
...
>>> X = Record()
>>> X.name = 'bob'
>>> X.job = 'Robiacy pizze'
```

Dziedziczenie wielokrotne: kolejność ma znaczenie

Może się to wydawać oczywiste, lecz jest godne podkreślenia: jeżeli używamy dziedziczenia wielokrotnego, to kolejność wpisywania klas nadrzędnych w nagłówku instrukcji `class` może mieć szczególne znaczenie. W podanym wcześniej przykładzie załóżmy, że klasa `Super` wprowadziła także metodę `__repr__`; czy chcielibyśmy wówczas dziedziczyć `Lister` lub `Super`? Uzyskamy je od jakiejkolwiek klasy, która występuje jako pierwsza w nagłówku klasy `Sub`, ponieważ wyszukiwanie podczas dziedziczenia następuje od lewej do prawej. Załóżmy następnie, że

`Super` i `Lister` mają także swoje własne wersje o innych nazwach; jeśli chcemy mieć jedną nazwę z `Super` i jedną z `Lister`, to musimy zastąpić dziedziczenie ręcznym przypisaniem do nazwy atrybutu w klasie `Sub`:

```
class Lister:
    def __repr__(self): ...
    def other(self): ...

class Super:
    def __repr__(self): ...
    def other(self): ...

class Sub(Super, Lister):        # podnosi __repr__ z Super wpisawszy ją jako
pierwszą
    other = Lister.other         # ale wyraźnie podnosi wersję other z Lister
    def __init__(self):
         ...
```

Rozwiązanie

Dziedziczenie wielokrotne to zaawansowane narzędzie i nawet jeśli ktoś zrozumiał ostatni podrozdział, to powinien używać tego narzędzia okazjonalnie i z dużą rozwagą. W przeciwnym wypadku znaczenie nazwy może zależeć od kolejności ustawienia klas w dowolnie odległej, lecz usuniętej klasie podrzędnej.

Atrybuty funkcji klasy są specjalne

Jest to proste, jeśli rozumie się podstawy modelu obiektowego języka Python. Obserwuje się jednak, że nowi użytkownicy wpadają w kłopoty, zwłaszcza jeśli swoje podstawy programowania obiektowego zdobyli w innych językach (a zwłaszcza w języku Smalltalk). W języku Python funkcje metody klasy nigdy nie mogą być wywołane bez egzemplarza. Wcześniej w tym rozdziale mówiliśmy o metodach niezwiązanych: jeżeli pobieramy funkcję metody za pomocą kwalifikacji klasy (zamiast egzemplarza), to otrzymujemy metodę niezwiązaną. Nawet wówczas, gdy obiekty metody niezwiązanej są definiowane za pomocą instrukcji `def`, nie są one prostymi funkcjami i nie można ich wywołać bez przekazania egzemplarza.

Załóżmy teraz, że chcemy użyć atrybutów klasy do sprawdzenia liczby egzemplarzy wygenerowanych z tej klasy. Pamiętając o tym, że atrybuty klasy są współdzielone przez wszystkie egzemplarze, możemy przechowywać licznik w samym obiekcie klasy:

```
class Mielonka:
    numInstances = 0
    def __init__(self):
        Mielonka.numInstances = Mielonka.numInstances + 1
    def printNumInstances():
        print "Liczba utworzonych egzemplarzy: ", Mielonka.numInstances
```

Coś takiego nie będzie działać: metoda `printNumInstances` nadal oczekuje, że w wywołaniu będzie przekazany egzemplarz, bowiem funkcja jest związana z klasą (pomimo braku argumentów w nagłówku `def`):

```
>>> from mielonka import *
>>> a = Mielonka()
>>> b = Mielonka()
>>> c = Mielonka()
>>> Mielonka.printNumInstances()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unbound method must be called with class instance 1st argument
```

Rozwiązanie

Nie należy oczekiwać rzeczy niemożliwych: metody niezwiązane nigdy nie będą dokładnie takie same, jak proste funkcje. Rozwiązanie problemów tego rodzaju jest w rzeczywistości sprawą wiedzy, ale jeśli ktoś koniecznie chce wywoływać funkcje, które mają dostęp do członków klasy bez przekazywania egzemplarza, to powinien po prostu uczynić je prostymi funkcjami, a nie metodami klasy. Dzięki temu wywołanie nie będzie oczekiwało egzemplarza:

```
def printNumInstances():
    print "Liczba utworzonych egzemplarzy: ", Mielonka.numInstances

class Mielonka
    numInstances = 0
    def __init__(self):
         Mielonka.numInstances = Mielonka.numInstances + 1

>>> import mielonka
>>> a = mielonka.Mielonka()
>>> b = mielonka.Mielonka()
>>> c = mielonka.Mielonka()
>>> mielonka.printNumInstances()
Liczba utworzonych egzemplarzy:  3
```

Całą tę pracę można również wykonać za pomocą wywołania poprzez egzemplarz, tak jak zwykle:

```
class Mielonka:
    numInstances = 0
    def __init__(self):
        Mielonka.numInstances = Mielonka.numInstances + 1
    def printNumInstances(self):
        print "Liczba utworzonych egzemplarzy: ", Mielonka.numInstances

>>> from mielonka import Mielonka
>>> a, b, c = Mielonka(), Mielonka(), Mielonka()
>>> a.printNumInstances()
Liczba utworzonych egzemplarzy: 3
>>> b.printNumInstances()
Liczba utworzonych egzemplarzy: 3
>>> Mielonka().printNumInstances()
Liczba utworzonych egzemplarzy: 4
```

Niektórzy teoretycy języka narzekają, że coś takiego oznacza, iż Python nie ma metod klasy, a tylko metody egzemplarza. Podejrzewamy, że w rzeczywistości mają na myśli to, iż klasy w języku Python działają w taki sam sposób, jak w innych językach. Python ma rzeczywiście obiekty metody związanej i niezwiązanej, łącznie z dobrze zdefiniowaną semantyką. Kwalifikowanie klas daje nam metodę niezwiązaną, która jest specjalnym rodzajem funkcji. Python rzeczywiście ma atrybuty klasy, chociaż funkcje w klasach oczekują egzemplarza jako argumentu.

Python także udostępnia *moduły* jako narzędzie do podziału przestrzeni nazw, zatem w zasadzie nie ma potrzeby opakowywania funkcji w klasy, chyba że wprowadzają one zachowanie obiektowe. Proste funkcje w modułach zwykle wykonują większość tego, co mogą wykonać metody klasy bez egzemplarza. W pierwszym przykładzie podanym w tym podrozdziale `printNumInstances` jest od razu związana z klasą, ponieważ istnieje w tym samym module.

Metody, klasy i zagnieżdżone zakresy

Klasy wprowadzają lokalny zakres tak, jak to czynią funkcje. Należy więc przygotować się na to, że takie same niespodzianki związane z zakresem mogą wystąpić w ciele instrukcji `class`. Oprócz tego metody są głębiej zagnieżdżonymi funkcjami, zatem pojawiają się w nich takie same problemy, jak w funkcjach. Zaskoczenie staje się powszechne, gdy zagnieżdżane są klasy. W podanym niżej przykładzie funkcja `generate` powinna zwracać egzemplarz zagnieżdżonej klasy `Mielonka`. W ramach jej kodu nazwa klasy `Mielonka` jest przypisana w lokalnym zakresie funkcji `generate`. W ramach funkcji `method` w klasie nazwa klasy `Mielonka` nie jest jednak widoczna; `method` ma dostęp tylko do swojego własnego zakresu lokalnego, do zakresu otaczającego funkcję `generate` i do nazw wbudowanych:

```
def generate():
    class Mielonka:
        count = 1
        def method(self):          # nazwa Mielonka niewidoczna:
            print Mielonka.count   # nie lokalna (def), globalna (moduł),
                                     ↳wbudowana
    return Mielonka()

generate().method()

C:\python\examples> python nester.py
Traceback (innermost last):
  File "nester.py", line 8, in ?
    generate().method()
  File "nester.py", line 5, in method
    print Mielonka.count      # nie lokalna (def), globalna (moduł), wbudowana
NameError: Mielonka
```

Rozwiązanie

Najważniejszą informacją, którą możemy tu przekazać, jest to, aby pamiętać o regule LGB; działa ona w klasach i funkcjach metod tak samo, jak w prostych funkcjach. Na przykład wewnątrz funkcji metody kod ma niekwalifikowany dostęp tylko do lokalnych nazw (w metodzie `def`), nazw globalnych (w otaczającym module) i nazw wbudowanych. Szczególnie brakuje tutaj otaczającej instrukcji `class`; aby dostać się do atrybutów klasy, metody muszą kwalifikować `self`, czyli egzemplarz. Aby wywołać jedną metodę z innej, wywołujący musi skierować wywołanie przez `self` (np. `self.method()`).

Istnieje wiele sposobów, aby podany wyżej przykład zaczął działać. Jednym z najprostszych jest przesunięcie nazwy `Mielonka` poza zakres otaczającego ją modułu za pomocą deklaracji globalnych. Metoda `method` rozpoznaje nazwy w otaczającym module zgodnie z regułą LGB, zatem odwołania do `Mielonka` działają wówczas poprawnie:

```
def generate():
    global Mielonka                 # wymusza zakres modułu dla Mielonka
    class Mielonka:
        count = 1
        def method(self):
            print Mielonka.count    # działa: w globalnej (dołączony moduł)
    return Mielonka()

generate().method()                 # drukuje 1
```

Być może lepiej byłoby przebudować przykład w taki sposób, aby klasa `Mielonka` była zdefiniowana na najwyższym poziomie modułu za pomocą właściwości zagnieżdżania jej poziomu, a nie za pomocą deklaracji `global`. Zarówno zagnieżdżona funkcja `method`, jak i `generate` z najwyższego poziomu znajdują nazwę `Mielonka` w swoich globalnych zakresach:

```
def generate():
    return Mielonka()

class Mielonka:                 # definicja na najwyższym poziomie modułu
    count = 1
    def method(self):
        print Mielonka.count    # działa: w globalnej (dołączony moduł)

generate().method()
```

Możemy również pozbyć się odwołania do `Mielonka` w `method`, używając specjalnego atrybutu `__class__`, który, jak pokazaliśmy, zwraca obiekt klasy egzemplarza:

```
def generate():
    class Mielonka:
        count = 1
        def method(self):
            print self.__class__.count # działa: kwalifikuje dla uzyskania
klasy
    return Mielonka()
generate().method()
```

Aby wykonać to zadanie, można także użyć sztuczki ze *zmiennym domyślnym* argumentem, pokazanej w rozdziale 4. Jest to jednak tak skomplikowane, że prawie nam wstyd pokazywać to Czytelnikom. Podane wyżej rozwiązania mają zazwyczaj większy sens:

```
def generate():
    class Mielonka:
        count = 1
        fillin = [None]
        def method(self, klass=fillin):  # zachowanie z otaczającego zakresu
            print klass[0].count         # działa: wartość domyślna włączona
    Mielonka.fillin[0] = Mielonka
    return Mielonka()

generate().method()
```

Zwracamy uwagę Czytelników na to, że nie możemy napisać `klass=Mielonka` w nagłówku `def` metody `method`, ponieważ nazwa `Mielonka` nie jest także widoczna w ciele `Mielonka`; nie jest to nazwa lokalna (w ciele klasy), ani globalna (w otaczającym module), ani wbudowana. Nazwa `Mielonka` istnieje tylko w lokalnym zakresie funkcji `generate`, której nie może widzieć ani zagnieżdżona klasa, ani jej metoda. Reguła LGB działa w ten sam sposób dla nich obydwu.

Podsumowanie

Ten rozdział był poświęcony dwóm specjalnym obiektom w języku Python: klasom i egzemplarzom, a także narzędziom języka, które je tworzą i przetwarzają. Obiekty klasy są tworzone za pomocą instrukcji `class`, określają domyślne zachowanie i służą jako generatory wielu egzemplarzy obiektów. Łącznie te dwa twory obsługują środowisko programowania obiektowego i umożliwiają ponowne użycie kodu. Mówiąc w skrócie: klasy pozwalają na wprowadzanie nowych obiektów, które eksportują zarówno dane, jak i zachowania.

Mówiąc o głównych różnicach tych obiektów, należy podkreślić, że klasy obsługują tworzenie wielu kopii obiektów, mogą być specjalizowane na podstawie dziedziczenia oraz mogą przeciążać operatory. Każda z tych właściwości została omówiona w tym rozdziale. Ponieważ wszystkie klasy są przestrzeniami nazw, pokazano również sposoby, dzięki którym klasy rozszerzają znaczenie modułu oraz przestrzeni nazw funkcji. Na zakończenie przedstawiono kilka zagadnień z dziedziny projektowania obiektowego, m.in. kompozycję i delegację, podając sposób ich wprowadzania w języku Python.

Następny rozdział kończy opis rdzenia języka krótkim przeglądem obsługi wyjątków — prostego narzędzia używanego raczej przy przetwarzaniu zdarzeń, niż przy tworzeniu składników programu. Podsumowując to, co zostało pokazane w tym rozdziale, podajemy krótki słowniczek terminów dotyczących klas i używanych w języku Python:

Klasa
: Obiekt (i instrukcja), która definiuje dziedziczone atrybuty.

Egzemplarz
: Obiekty utworzone z klasy, które dziedziczą jej atrybuty i posiadają swoją własną przestrzeń nazw.

Metoda
: Atrybut obiektu klasy, który jest związany z obiektem funkcji.

self
: Zgodnie z konwencją, nazwa nadawana wynikowemu egzemplarzowi obiektu w metodach.

Dziedziczenie
: Odbywa się wówczas, gdy egzemplarz lub klasa uzyskuje dostęp do atrybutów klasy za pomocą kwalifikacji.

Klasa nadrzędna
: Klasa, od której inna klasa dziedziczy atrybuty.

Klasa podrzędna
: Klasa, która dziedziczy nazwy atrybutów z innej klasy.

Ćwiczenia

Chcemy, by Czytelnicy w tej sesji ćwiczeń utworzyli kilka klas i poeksperymentowali z pewnym gotowym kodem. Oczywiście problem z gotowym kodem jest taki, że musi on istnieć. Aby pracować z klasą opisaną w ćwiczeniu 5., należy albo pobrać jej kod źródłowy z Internetu (patrz

Wstęp), albo wpisać go ręcznie (jest on całkiem krótki). Programy stają się coraz bardziej zaawansowane, zatem należy sprawdzać rozwiązania podane na końcu książki. Osobom bardzo zajętym polecamy szczególnie ostatnie ćwiczenie na temat kompozycji — dostarczy im ono nieco rozrywki (oczywiście Autorzy znają już odpowiedzi).

1. *Podstawy*. Utworzyć klasę o nazwie `Adder`, która eksportuje metodę `add(self, x, y)`, drukującą komunikat „Nie zaimplementowano". Następnie zdefiniować dwie klasy podrzędne w stosunku do `Adder`, które wprowadzają metodę `add`:

   - `ListAdder` z metodą `add`, która zwraca połączenie swoich dwóch argumentów będących listami,
   - `DictAdder` z metodą `add`, która zwraca nowy słownik z pozycjami istniejącymi w swoich dwóch argumentach słownikowych (można użyć dowolnej definicji dodawania).

   Wykonać kilka doświadczeń, tworząc w trybie interakcyjnym egzemplarze wszystkich trzech klas i wywołując ich metody `add`. Na zakończenie rozszerzyć swoje klasy tak, aby zapisać obiekt w konstruktorze (lista lub słownik) i przeciążyć operator + w celu zastąpienia metody `add`. Gdzie (tzn. w których klasach) znajduje się najlepsze miejsce na wstawianie konstruktorów i metod przeciążania operatora? Jakie rodzaje obiektów można dodać do egzemplarzy klas?

2. *Przeciążanie operatora*. Utworzyć klasę o nazwie `Mylist`, która „opakowuje" listę języka Python: powinna ona przeciążać większość operatorów listy i operacji: +, indeksowanie, iterację i wycinanie, oraz metody listy, takie jak `append` i `sort`. Spis wszystkich możliwych metod, które dopuszczają przeciążanie, znajduje się w podręczniku systemowym języka Python. Zapewnić, by konstruktor tworzonej klasy pobierał istniejącą listę (lub egzemplarz `Mylist`) i kopiował jej składniki do członka egzemplarza. Poeksperymentować z utworzoną klasą w trybie interakcyjnym, próbując znaleźć odpowiedzi na następujące pytania:

   - Dlaczego kopiowanie wartości początkowej jest tutaj tak istotne?
   - Czy można tu użyć pustego wycinka (np. `start[:]`) do skopiowania wartości początkowej, jeśli jest to egzemplarz `Mylist`?
   - Czy istnieje jakiś ogólny sposób na skierowanie wywołań metody listy do opakowanej listy?
   - Czy można dodawać `Mylist` i zwykłą listę? Co można powiedzieć o dodawaniu egzemplarza listy i `Mylist`?
   - Jakie typy obiektu powinny zwracać operatory, takie jak + i wycinanie? Co można w tym kontekście powiedzieć o indeksowaniu?

3. *Klasy podrzędne*. Utworzyć klasę podrzędną klasy `Mylist` z ćwiczenia 2. i nazwać ją `MylistSub`. Powinna ona rozszerzać `Mylist` w taki sposób, aby można było przesyłać komunikat do `stdout` przed wywołaniem każdej przeciążonej operacji. Oprócz tego podklasa powinna zliczać liczbę wywołań. `MylistSub` powinna dziedziczyć zachowania podstawowej metody z `Mylist`, czyli np. dodawanie sekwencji do `MylistSub` powinno powodować wydruk komunikatu, zwiększenie zawartości licznika dla wywołań operacji + i uruchomienie metody klasy nadrzędnej. Dodatkowo należy także wprowadzić nową metodę, która wyświetla liczniki operacji na `stdout`. Wykonać w trybie interakcyjnym kilka doświadczeń z utworzoną klasą. Czy liczniki zliczają wywołania dla danego egzemplarza, czy dla klasy (tzn. wszystkich egzemplarzy klasy)? Jak należy zaprogramować oba warianty?

Wskazówka: zależy to od tego, do którego obiektu są przypisani członkowie licznika, bowiem członkowie klasy są współdzieleni przez egzemplarze, członkowie `self` są danymi egzemplarza.

4. *Metody metaklasy*. Utworzyć klasę o nazwie `Meta` z metodami, które przechwytują każdą kwalifikację atrybutu (zarówno przy pobieraniu, jak i przypisaniu). Klasa ma wpisywać do `stdout` komunikat zawierający argumenty tych metod. Utworzyć egzemplarz `Meta` i wykonać interakcyjnie kilka doświadczeń z kwalifikowaniem. Co się dzieje przy próbie użycia egzemplarza w wyrażeniach? Wypróbować dodawanie, indeksowanie i wycinanie egzemplarza utworzonej klasy.
5. *Zbiory obiektów*. Wykonać doświadczenie z klasą zbioru opisaną w tym rozdziale (z podrozdziału „Rozszerzanie wbudowanych typów obiektów"). Uruchomić polecenia wykonujące podane niżej operacje:
   a. Utworzyć dwa zbiory liczb całkowitych i obliczyć ich przecięcie i złączenie za pomocą wyrażeń operatorowych `&` i `|`.
   b. Utworzyć zbiór z łańcucha i wykonać eksperyment z indeksowaniem tego zbioru. Jakie metody klasy są wywołane?
   c. Spróbować wykonać iterację pozycji w zbiorze łańcuchowym, używając pętli `for`. Jakie metody działają tym razem?
   d. Należy obliczyć przecięcie i złączenie zbioru łańcuchowego i prostego łańcucha. Czy to działa?
   e. Rozszerzyć zbiór za pomocą klas podrzędnych, by obsługiwał dowolną liczbę operandów, posługując się argumentem `*args`. Wskazówka: patrz wersje funkcji dla tych algorytmów podane w rozdziale 4. Obliczyć przecięcia i złączenia wielu operandów za pomocą klasy podrzędnej zbioru. W jaki sposób można obliczyć przecięcie trzech lub więcej zbiorów, wiedząc, że operator `&` działa tylko dwustronnie?
   f. W jaki sposób należy emulować inne operacje na listach w klasie zbioru? Wskazówka: `__add__` może wyłapywać łączenie, a `__getattr__` może przekazywać większość wywołań metody listy poza opakowaną listę.
6. *Powiązania drzewa klas*. W przypisie w podrozdziale na temat dziedziczenia wielokrotnego wspomniano, że klasy mają atrybut `__bases__`, który zwraca krotkę obiektów klas nadrzędnych klasy (tych w nawiasach w nagłówku klasy). Użyć `__bases__` do rozszerzenia klasy domieszkowej `Lister` w taki sposób, by drukowała ona także nazwy klas bezpośrednio nadrzędnych dla klasy egzemplarza. Po wykonaniu tego zadania pierwszy wiersz reprezentacji łańcuchowej powinien wyglądać następująco:

```
<Instance of Sub(Super, Lister), address 7841200:
```

   Co można więcej powiedzieć o wypisywaniu atrybutów klasy?
7. *Kompozycja*. Zasymulować scenariusz zamawiania posiłku w barze, definiując cztery klasy:
   - `Lunch`: pojemnik i klasa kontrolera,
   - `Klient`: aktor, który kupuje jedzenie,
   - `Pracownik`: aktor, u którego klient zamawia,
   - `Jedzenie`: to, co zamawia klient.

Jako początkowa wskazówka niechaj posłużą podane niżej klasy i metody, które Czytelnik będzie definiował:

```
class Lunch:
    def __init__(self)   # utworzenie/osadzenie Klienta i Pracownika
    def zamowienie(self, jedzenieNazwa)  # start zamawiania przez Klienta
    def wynik(self)      # zapytanie Klienta, jaki rodzaj Jedzenia chce jeść

class Klient:
    def __init__(self)        # inicjacja: moje jedzenie = None
    def skladaZamowienie(self, jedzenieNazwa, pracownik)  # zamówienie
    def printJedzenie(self)   # drukowanie nazwy mojego jedzenia

class Pracownik:
    def bracZamowienie(self, jedzenieNazwa) # podaje żądane Jedzenie

class Jedzenie:
    def __init__(self, nazwa)      # zachowuje nazwę jedzenia
```

Symulacja zamówienia ma działać następująco:

- Konstruktor klasy `Lunch` powinien tworzyć i osadzać egzemplarze z klas `Klient` i `Pracownik` oraz eksportować metodę `zamowienie`. Wywołana metoda `zamowienie` powinna poprosić, aby `Klient` złożył zamówienie, korzystając z metody `skladaZamowienie`. Metoda `skladaZamowienie` z klasy `Klient` powinna w odpowiedzi zapytać obiekt `Pracownik` o nowy obiekt `Jedzenie`, wywołując metodę `bracZamowienie` z klasy `Pracownik`.
- Obiekty `Jedzenie` powinny przechowywać łańcuch, będący nazwą posiłku (np. `"burritos"`), przekazywany kolejno z `Lunch.zamowienie` do `Klient.skladaZamowienie`, do `Pracownik.bracZamowienie` i na koniec do konstruktora `Jedzenie`. Klasa najwyższego poziomu `Lunch` powinna także eksportować metodę zwaną `wynik`, która prosi klienta o wydrukowanie nazwy posiłku, który otrzymał od klasy `Pracownik` (może być to użyte do testowania symulacji).
- Zauważmy, że klasa `Lunch` musi albo przekazać klasę `Pracownik` do klasy `Klient`, albo przekazać samą siebie do klasy `Klient`, by umożliwić klasie `Klient` wywoływanie metod klasy `Pracownik`.

8. Poeksperymentować z klasami w trybie interakcyjnym, importując klasę `Lunch`. Wywołać jej metodę `zamowienie`, by uruchomić całe działanie, a następnie wywołać metodę `wynik` w celu sprawdzenia, czy `Klient` otrzymał to, co zamówił. W tej symulacji `Klient` jest aktywnym agentem; w jaki sposób zmienią się klasy, gdyby `Pracownik` był obiektem, który inicjuje relację klient-pracownik?