Podstawy języka Python

Skrypt akademicki

Paweł Kłeczek

pkleczek@agh.edu.pl
home.agh.edu.pl/~pkleczek

v0.6.2
(2021-06-18)

Spis treści

Rozdział 1

Podstawy

Niniejszy rozdział obejmuje podstawowe zagadnienia niezbędne do rozpoczęcia przygody z Pythonem: ogólne założenia języka, podstawowe wbudowane typy danych, instrukcje sterujące oraz funkcje. Gdy opanujesz zawarty w nim materiał, będziesz w stanie stosować Pythona do rozwiązywania problemów inżynierskich i algorytmicznych.

Treść niniejszego skryptu opiera się w dużej mierze na materiałach z poniższych źródeł:

- The Python Tutorial (PyDocs)
- Python 3 Tutorial (Python Course)

1.1 Kodowanie pliku źródłowego

Kodowanie plików źródłowych różni się w zależności od użytego systemu operacyjnego i/lub edytora, przykładowo:

- wiele edytorów tekstu dla systemu Windows w polskiej wersji językowej korzysta domyślnie z kodowania *Windows-1250*,
- edytory tekstu dla systemu Windows w angielskiej wersji językowej korzystają domyślnie z kodowania *Windows-1252*, natomiast
- edytory tekstu dla systemów z rodziny UNIX często kodują pliki tekstowe z użyciem kodowania *Latin-1*.

Natomiast w celu zachowania maksymalnej przenośności programu warto skorzystać z kodowania Unicode (*UTF-8*), udostępniającego znaki większości alfabetów stosowanych obecnie na świecie (ma to znaczenie m.in. w przypadku wyświetlania takich znaków z użyciem funkcji `print()`, natomiast nie robi różnicy w przypadku komentarzy – komentarze są bowiem pomijane przez interpreter).

Aby interpreter języka Python wiedział, jakiego kodowania użyto w danym pliku tekstowym (i mógł m.in. ostrzec o wystąpieniu w kodzie programu znakach niepasujących do tego kodowania), warto umieścić na *samym* początku danego pliku (tj. jako zawartość wierszy 1–2) poniższe dwie linie specjalnego "magicznego"komentarza:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
```

Powyższy format informacji o kodowaniu jest również rozpoznawany przez wiele popularnych edytorów tekstu, co dodatkowo zminimalizuje błędy złego kodowania.

Dla zainteresowanych...

Dokładniejsze omówienie (i uzasadnienie) opisanej w tym rozdziale operacji znajdziesz w dokumencie PEP 263: Defining Python Source Code Encodings.

1.2 Typy danych

1.2.1 Liczby

Liczby całkowite są typu `int`, liczby rzeczywiste – typu `float`.

Oto przykłady operatorów arytmetycznych:

```
>>> 17 / 3  # "klasyczne" dzielenie zwraca typ float
5.666666666666667
>>>
>>> 17 // 3  # dzielenie bez reszty (odrzuca część ułamkową)
5
>>> 17 % 3  # operator % zwraca resztę z dzielenia
2
>>> 5 ** 3  # 5 do sześcianu
125
```

Należy pamiętać, że w przypadku operacji na operandach różnego typu liczbowego, zachodzi konwersja z `int` na `float`:

```
>>> 4 * 3.75 - 1
14.0
```

1.2.2 Łańcuchy znaków

Łańcuchy znaków – czyli obiekty typu `str` – definiuje się za pomocą apostrofów (`'...'`) lub cudzysłowów (`"..."`):

```
>>> 'spam eggs'    # apostrofy
'spam eggs'
>>> "spam eggs"    # cudzysłowy
"spam eggs"
```

Jeśli nie chcesz, aby znaki poprzedzone symbolem \ były traktowane jako znaki specjalne, użyj „surowych” łańcuchów – poprzedzając pierwszy apostrof/cudzysłów literą `r` (zob. przykł. 1.1).

Listing 1.1. „Surowe” łańcuchy znaków

```
>>> print('C:\some\name')    # Tu `\n` oznacza znak niedrukowany
                             #   -- znak nowego wiersza.
C:\some
ame
>>> print(r'C:\some\name')   # Tu `\n` oznacza dwa znaki drukowane
                             #   (bo zastosowano "surowy" łańcuch znaków).
C:\some\name
```

1.2.3 Kontrola typów

Język Python jest **typowany dynamicznie** (ang. dynamically typed) – oznacza to, że typ obiektu określany jest dopiero w trakcie działania programu[1]:

```
v = 3      # `v` to alias obiektu typu `int`...
v = 'abc'  # ... a teraz `v` to alias obiektu typu `str`
```

Po wykonaniu drugiej instrukcji utworzony wcześniej obiekt typu całkowitego wciąż znajduje się w pamięci, dopóki nie zostanie usunięty przez tzw. *odśmiecacz*[2], natomiast alias `v` zaczyna odnosić się do innego obiektu.

Język Python jest również **silnie typowany** (ang. strongly typed), a zatem nie można dokonywać operacji na danych o niezgodnych typach (w języku Python nie ma niejawnych konwersji – z wyjątkiem promocji `int` do `float`):

[1] W odróżnieniu od języków **typowanych statycznie** (ang. statically typed), w których typ zmiennej jest jawnie określany przez programistę w momencie jej definicji i nie może ulec zmianie. Przykładami języków typowanych statycznie są C i C++.

[2] zob. rozdz. 7.1 Garbage collector

```
'abc' + 3        # BŁĄD: nie można dodawać obiektów o różnych
                 #    typach (`str` i `int`)

'abc' + str(3)   # OK, gdyż `str()` reprezentację argumentu
                 #   jako łańcuch znaków (wynik: 'abc3')
```

1.2.4 Listy

Listy służą do przechowywania ciągu elementów. Elementy mogą być różnego typu, choć *zwykle* wszystkie elementy listy są tego samego typu.

Listę – obiekt typu `list` – definiuje się za pomocą nawiasów kwadratowych, wewnątrz których podaje się elementy rozdzielone przecinkami:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Dokładniejsze omówienie typu `list` znajdziesz w rozdziale 1.9.1 Listy.

1.2.5 Typy mutowalne i niemutowalne

Każda dana w Pythonie jest obiektem w ujęciu inżynierii oprogramowania – czyli jest instancją pewnej klasy. Od typu obiektu zależy, czy jest on mutowalny (tj. czy można zmienić zawartość obiektu bez zmieniania jego tożsamości), czy też nie:

- typy mutowalne: `list`, `dict`
- typy niemutowalne: `int`, `float`, `str`, `tuple`

Dostęp do obiektów niemutowalnych jest szybszy niż do obiektów mutowalnych, natomiast „zmiana” obiektu niemutowalnego jest bardziej kosztowna – wymaga bowiem wykonania kopii „zmienianego” obiektu i odpowiedniego przepięcia aliasu (tak, aby odnosił się do nowego obiektu), przykładowo:

```
v = 3  # `v` to alias obiektu typu `int` o wartości 3.
v = 4  # Teraz `v` to alias obiektu typu int` o wartości 4.
# Obiekt typu `int` o wartości 3 wciąż znajduje się w pamięci,
#   ale jest oznaczony przez interpreter jako "możliwy do usunięcia".
```

Próba zmiany obiektu niemutowalnego skutkuje błędem (zob. przykł. 1.2).

Listing 1.2. Próba zmiany obiektu niemutowalnego.

```
>>> word = 'abc'
>>> word[0] = 'X'
  ...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'str' object does not support item assignment
>>>
>>> 'X' + word[1:]   # rozwiązanie: stworzyć nowy łańcuch znaków
'Xbc'
```

W przypadku obiektów mutowalnych zmiana istniejącego obiektu jest możliwa (zob. przykł. 1.3).

Listing 1.3. Obiekty mutowalne można zmieniać.

```
>>> squares = [1, 4, 8]
>>> squares[2] = 9
>>> squares
[1, 4, 9]
```

1.2.6 Typy sekwencyjne

Zarówno łańcuchy znaków jak i listy należą do tzw. **typów sekwencyjnych** (ang. sequence types). Wszystkie typy sekwencyjne udostępniają pewne podstawowe operacje (zob. Tab. 1.1), które zostały omówione w poniższych podrozdziałach.

Tablica 1.1. Wybrane operacje właściwe dla typów sekwencyjnych

Operacja	Wynik
`len(s)`	długość `s` (tj. liczba elementów `s`)
`s[i]`	i-ty element `s`
`s[i:j]`	*slice* z `s` od i do j
`s + t`	konkatenacja `s` i `t`
`s * n , n * s`	konkatenacja n płytkich kopii `s`
`x in s`	`True` jeśli jeden z elementów `s` wynosi `x`, inaczej `False`
`x not in s`	`False` jeśli jeden z elementów `s` wynosi `x`, inaczej `True`
`min(s)`	najmniejszy element z `s`
`max(s)`	największy element z `s`

Długość ciągu

Wbudowana funkcja `len()` zwraca długość ciągu – liczbę jego elementów (wartość typu `int`):

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Indeksowanie

Indeksowanie (ang. indexing) umożliwia dostęp do pojedynczych elementów sekwencji (ciągu):

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0]  # dostęp do pierwszego elementu (tj. na pozycji 0)
1
>>> squares[1]  # dostęp do drugiego elementu (tj. na pozycji 1)
4
```

Indeks może być liczbą ujemną – wtedy liczenie pozycji zaczyna się od prawej strony ciągu – przy czym dla pewnego obiektu typu sekwencyjnego `seq` zapis `seq[-i]` (dla $i \in \mathbb{N}_+$) jest równoważny zapisowi `seq[len(seq) - i - 1]`:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[-1]  # ostatni element
25
>>> squares[-2]  # przedostatni element
16
```

Zwróć uwagę, że ponieważ $-0 = 0$, ujemne indeksy zaczynają się od `-1` (czyli indeks `-1` odnosi się do elementu *ostatniego*, a nie przedostatniego).

Próba odwołania się do elementu spoza zakresu spowoduje błąd:

```
>>> squares[7]  # ciąg `squares` ma tylko 5 elementów
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Slicing

Operacja **slicingu** służy do uzyskiwania podciągów (tzw. **slice'ów**):

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[0:2]  # elementy od pozycji 0 (włącznie) do 2 (bez tej pozycji)
[1, 4]
```

Zwróć uwagę, że `s[a:b]` zwraca zawsze elementy o indeksach z zakresu $\langle a, b)$, czyli element o indeksie b *nie wchodzi* w skład podciągu. W wyniku slicingu tworzona jest nowa (płytka) kopia ciągu[3].

Możesz pominąć indeksy (jeden z nich lub oba), gdyż domyślnie:

- pierwszy z nich ma wartość 0, a
- wartość drugiego odpowiada długości całego ciągu.

przy czym własności te sprawiają, że zawsze spełniona jest zależność `s[:i] + s[i:] == s`. Własności te ilustruje przykład 1.4.

Listing 1.4. Domyślne wartości indeksów przy *slice*'ingu

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[3:]  # elementy od elementu o indeksie 3 (włącznie) do końca ciągu
[16, 25]
>>> squares[:3]  # elementy od początku do elementu o indeksie 2 (włącznie)
[1, 4, 9]
>>> squares[:]   # wszystkie elementy, tworzona jest płytka kopia
[1, 4, 9, 16, 25]
>>> squares[-2:] # elementy od przedostatniego do ostatniego
[16, 25]
```

Aby zapamiętać jak działa slicing możesz wyobrazić sobie indeksy jako „wskaźniki" pomiędzy elementami pewnego ciągu, przy czym:

- „wskaźnik" na pozycję tuż przed pierwszym elementem ma numer 0, natomiast
- „wskaźnik" na pozycję tuż za ostatnim elementem w ciągu długości n ma numer n;

przykładowo:

```
  0   1   2   3   4   5    <- indeksy elementów
+---+---+---+---+---+---+
| P | y | t | h | o | n |  (ciąg o długości 6 elementów)
+---+---+---+---+---+---+
|   |   |   |   |   |   |
0   1   2   3   4   5   6  <- "wskaźniki"
```

Podciąg `[i:j]` zawiera wszystkie znaki pomiędzy krawędziami oznaczonymi odpowiednio i i j.

[3] Zagadnienie kopiowania zostało dokładniej omówione w rozdz. 1.9.7 Kopiowanie obiektów.

Analogicznie działa slicing z zastosowaniem indeksów ujemnych – trzeba tylko pamiętać, że wówczas liczymy „od końca”, czyli:

- „wskaźnik” na pozycję tuż przed pierwszym elementem w ciągu długości n ma numer $-n$,
- „wskaźnik” na pozycję tuż przed ostatnim elementem ma numer -1, natomiast
- „wskaźnik” o numerze 0 nie istnieje.

W poniższym przykładzie pierwszy wiersz z numerami „wskaźników” zawiera indeksy nieujemne $(0 \ldots 6)$, natomiast drugi wiersz zawiera odpowiadające im indeksy ujemne:

```
      | P | y | t | h | o | n |
      |   |   |   |   |   |   |
 I    0   1   2   3   4   5   6
II   -6  -5  -4  -3  -2  -1      <- W tym wierszu "wskaźnik" 0
                                      nie istnieje!
```

W przypadku indeksów ujemnych podciąg od i do j również zawiera wszystkie znaki pomiędzy krawędziami oznaczonymi odpowiednio numerami i i j.

W przeciwieństwie do indeksowania, w przypadków slicingu dopuszczalne jest użycie indeksu i (lub pary indeksów i oraz j) spoza zakresu, tj. o wartości mniejszej niż 0 lub o wartości większej niż indeks ostatniego elementu, gdyż każda z wartości indeksu użyta do uzyskania slice'a zostanie odpowiednio „przycięta” zgodnie z dopuszczalnym zakresem:

```
>>> squares = [1, 4, 9, 16, 25]  # len(squares) = 5
>>> squares[3:10]  # 10 > 5; równoważne: squares[3:len(squares)]
[16, 25]
>>> squares[-8:2]  # -8 < 0; równoważne: squares[0:2]
[1, 4]
```

Istnieje również możliwość dokonywania slicingu z zadanym krokiem k, czyli wyboru co k-tego elementu z zakresu $\langle a, b)$. Służy do tego składnia `s[a:b:k]`:

```
>>> squares = [1, 4, 9, 16, 25, 36, 49]
>>> squares[1::2]  # elementy o indeksach 1, 3, 5, ...
[4, 16, 36]
```

Konkatenacja i powtarzanie

Konkatenacja (ang. concatenation), in. łączenie, to operacja „sklejania” dwóch ciągów w jeden – służy do tego operator `+`:

```
>>> s1 = [1, 2]
>>> s2 = [3]
>>> s1 + s2
[1, 2, 3]
```

Ciągi można również **powtarzać** za pomocą operatora `*`:

```
>>> 3 * '123'
123123123
```

Operator `in`

Wbudowany operator `in` służy sprawdzeniu, czy dany element występuje w ciągu:

```
>>> s = [1, 2, 3]
>>> 1 in s
True
>>> 4 in s
False
```

W przypadku łańcucha znaków operator `in` sprawdza, czy ciąg zawiera zadany podciąg:

```
>>> s = 'abc'
>>> 'ab' in s
True
>>> 'bx' in s
False
```

Do sprawdzenia, czy dany element *nie występuje* w pewnym ciągu elementów (albo czy dany podciąg *nie występuje* w pewnym łańcuchu znaków) służy konstrukcja **`not in`**:

```
>>> s = [1, 2, 3]
>>> 1 not in s
False
>>> 4 not in s
True

>>> s = 'abc'
>>> 'ab' not in s
False
>>> 'bx' not in s
True
```

1.3 Wartość `None`

Specjalna wartość `None` służy do realizacji koncepcji „neutralnej wartości początkowej”, przykładowo:

- do zasygnalizowania, że obiekt nie został jeszcze zainicjalizowany albo
- do sygnalizowania brakującej wartości na liście elementów.

```
obj = None  # inicjalizacja wartością `None` to dobry wybór,
            #   gdy brak innej rozsądnej wartości początkowej
```

Rozważ następujący scenariusz:
Stacja pomiarowa ma działać w sposób ciągły i co godzinę dokonywać pomiaru temperatury (w stopniach Celsiusza), a wynik pomiaru przekazywać do dalszej analizy.

Pojawia się pytanie – w jaki sposób można sygnalizować błędny pomiar lub brak dokonania pomiaru (np. z powodów technicznych)? Nie możemy użyć żadnej spośród wartości rzeczywistych, gdyż (teoretycznie) mogą one oznaczać rzeczywistą temperaturę – niby dopuszczalny dolny zakres temperatury wynosi $-273.15\,^{\circ}\mathrm{C}$ i moglibyśmy użyć mniejszej wartości w celu sygnalizowania błędów, ale to byłoby rozwiązanie opierające się wyłącznie na pewnych konwencjach *w ramach danego programu*, a nie na rozwiązaniach „systemowych” tego problemu.

W języku Python obiekt `None` może posłużyć właśnie m.in. do wyrażania takiego specjalnego stanu:

```
measured_temperature = None  # `None` może wyrażać "brak" wartości
                             #  (np. brak wykonania pomiaru)
```

Dla zainteresowanych...

Innym zastosowaniem obiektu `None` jest nadanie zmiennej pewnej „neutralnej” wartości początkowej – w tym kontekście korzysta się z niego szczególnie często podczas definiowania funkcji, jako domyślnej wartości opcjonalnych parametrów[4].

Ważne

Obiekt `None` jest unikalny – podczas całego działania programu w pamięci komputera istnieje tylko jedna jego instancja:

```
obj1 = None
obj2 = None  # `obj1` i `obj2` to aliasy tego samego obiektu `None`
```

[4] zob. rozdz. 1.8.1 Argumenty domyślne

1.4 Formatowanie wyjścia

We współczesnym Pythonie zalecanym sposobem formatowania wyjścia jest skorzystanie albo z metody `str.format()`, albo z tzw. *formatowanych literałów łańcuchowych*.

1.4.1 Metoda `str.format()`

Działanie metody `format()` dla pewnego obiektu `s` klasy `str` wygląda następująco:

- Łańcuch `s` może zawierać literały teksowe albo tzw. *pola zamiany* (ang. replacement fields), ograniczone nawiasami klamrowymi `{}`.
- Każde „pole zamiany" zawiera albo liczbowy indeks argumentu pozycyjnego, albo nazwę argumentu słownikowego[5].
- Metoda zwraca taką kopię łańcucha `s`, w której każde „pole zamiany" zostało podmienione na łańcuch znaków o wartości wyrażającej odpowiadający mu argument.

Przykładowo, poniższe wywołanie metody `format()` korzysta z argumentów pozycyjnych:

```
a = 1
b = 2
"The sum of {0} + {1} is {2}".format(a, b, a + b)   # 'The sum of 1 + 2 is 3'
```

i jest znacznie bardziej zwięzłe w porównaniu do konkatenacji łańcuchów:

```
"The sum of " + str(a) + " + " + str(b) + " is" + str(a + b)   # mało czytelne
```

W przypadku „pól zmiany" o szczególnym znaczeniu semantycznym przydatne jest korzystanie z argumentów słownikowych, przykładowo:

```
'P(x,y) = ({x}, {y})'.format(x=1, y=2)   # 'P(x,y) = (1, 2)'
```

Więcej przykładów użycia metody `str.format()` znajdziesz na stronie Format examples.

Dla zainteresowanych...

Szczegółowy opis składni metody `str.format()`, który da Ci pojęcie o większych możliwościach formatowania, znajdziesz na stronach:

- Format String Syntax
- Fancier Output Formatting (PyDocs)
- PEP 3101 – Advanced String Formatting

1.4.2 Formatowane literały łańcuchowe

Formatowane literały łańcuchowe (ang. formatted string literals, f-strings), wprowadzone w wersji Python 3.6, można traktować jako uproszczony, skrócony zapis formatowania, analogiczny w działaniu do metody `str.format()` – poprzedzając łańcuch znaków przedrostkiem „`f`"[6] możesz bezpośrednio w „polach zamiany" umieścić dowolne wyrażenia, których wartość zostanie obliczona na podstawie wartości obiektów widocznych w bieżącym kontekście i podstawiona w miejsce tych pól, przykładowo:

```
name = 'Bob'
age = 7
f'{name} is {age} years old.'   # 'Bob is 7 years old.'
f'In two years {name} will be {age + 2}.'   # 'In two years Bob will be 9.'
```

Jest to zapis bardziej zwięzły, niż użycie `str.format()`:

```
'{name} is {age} years old.'.format(name=name, age=age)   # rozwlekłe...
```

[5] Te typy argumentów zostaną dokładniej omówione w rozdz. 1.8.2 Argumenty pozycyjne i słownikowe.

[6] od ang. *formatted* – formatowany

1.5 Wyrażenia logiczne

Wyrażenie logiczne to takie wyrażenie, któremu można przyporządkować jedną z dwóch wartości – *prawda* albo *fałsz*. Wyrażenia logiczne w Pythonie działają na podobnej zasadzie, co w językach C i C++, z tym że operatory logiczne mają wyłącznie postać słowną:

- **not** – negacja
- **or** – alternatywa
- **and** – koniunkcja

natomiast operatory porównania są identyczne w każdym z tych trzech języków.
Oto przykład wyrażenia logicznego:

```
a > 3 or not b == 4   # przykłady operatorów porównania (>, ==)
                      #   i logicznych (or, not)
```

Python udostępnia dwa specjalne operatory – (**not**) **in** oraz **is** (**not**):

- **in** / **not in** – sprawdza, czy wartość występuje/nie występuje w ciągu
- **is** / **is not** – sprawdza, czy dwie zmienne odnoszą się do tego samego obiektu (to ma szczególne znaczenie w przypadku typów mutowalnych)

Ważne

Do sprawdzania czy zmienna jest *tożsama* z `None`, czyli „czy dana zmienna odnosi się do obiektu `None`", korzystaj zawsze z **is** (a nie z operatora porównania ==)!

Wartość wyrażenia logicznego można przypisać do zmiennej:

```
>>> age = 10
>>> has_age_discount = (age < 7) or (age > 65)
>>> has_age_discount
False
```

co poprawia czytelność kodu, gdy później z takiej wartości korzystamy w instrukcji warunkowej lub jako warunek pętli.

1.6 Instrukcje sterujące

Podstawowa funkcjonalność instrukcji sterujących w języku Python jest podobna do ich odpowiedników w językach C i C++, lecz język Python udostępnia też pewne dodatkowe możliwości.

1.6.1 Instrukcje warunkowe

Instrukcja warunkowa w języku Python działa analogicznie do instrukcji warunkowej w językach C i C++ (zwróć uwagę na brak nawiasów okrągłych wokół wyrażenia warunkowego oraz występujący po nim dwukropek):

```
>>> x = 3
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

Klauzule **elif** oraz **else** są opcjonalne. W Pythonie nie ma odpowiednika znanej z języków C i C++ instrukcji `switch`.

Short-Circut Evaluation

Wartości wyrażeń logicznych zawierających operatory **and** i **or** są obliczane od lewej do prawej, a proces zostaje przerwany gdy tylko można już określić wynik całego wyrażenia. Przykładowo, jeśli `A` i `C` mają wartość *prawda* a `B` wartość *fałsz*, w wyrażeniu `A and B and C` wartość `C` nie zostanie obliczona (gdyż po obliczeniu `A and B` wiadomo już, że wartością całego wyrażenia będzie *fałsz*).

Wyrażenia warunkowe

Wyrażenie warunkowe (ang. conditional expression) jest odmianą instrukcji warunkowej `if-else` z tą różnicą, że wykonany blok kodu musi zwrócić jakąś wartość, która staje się jednocześnie wynikiem całego wyrażenia[7]. Oto składnia wyrażenia warunkowego:

```
x = true_value if condition else false_value
```

przykładowo

```
>>> x = 'even' if (5 % 2) == 0 else 'odd'
>>> x
odd
```

1.6.2 Pętle

Instrukcja `while`

Pętla **while** działa analogicznie do pętli w językach C i C++:

```
>>> t = 1
>>> while t < 10:
...     t = t + 1
>>> print(t)
10
```

Instrukcja `for`

W przypadku pętli **for** iteracja następuje po *elementach* danego ciągu (np. listy)[8]:

```
>>> # Measure some strings:
... words = ['cat', 'window']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
```

Funkcja `range`

Do iterowania po ciągu liczb przydaje się wbudowana funkcja `range()`, przyjmująca do trzech argumentów:

- `range(e)` – liczby od 0 do $(e-1)$
- `range(b, e)` – liczby od b do $(e-1)$
- `range(b, e, s)` – liczby od b do $(e-1)$, z krokiem s

Przykładowo:

```
>>> for i in range(3):
...     print(i)
...
```

[7] To odpowiednik operatora warunkowego `?:` w językach C i C++.

[8] Zatem pętla **for** w języku Python przypomina konstrukcję *range-based for loop* w standardzie C++11.

```
0
1
2
```

Oto dalsze przykłady ciągów generowanych za pomocą funkcji `range()` (zwróć uwagę na konieczność „zrzutowania” wyniku np. na listę, aby uzyskać bezpośredni dostęp do wszystkich wartości ciągu[9]):

```
>>> list(range(2, 5))
[2, 3, 4]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]
```

Instrukcje `break` i `continue`

Instrukcje **`break`** i **`continue`** zachowują się tak, jak w językach C i C++. Wykonanie poniższego programu:

```
for n in range(1, 10):
  if n % 2 == 0:
    print('even - skipped')
    continue
  if n == 5:
    break
  print(n)
```

da następujący rezultat:

```
1
even - skipped
3
even - skipped
```

1.7 Instrukcja `pass`

Instrukcja **`pass`** nie robi nic – może być zatem używa w sytuacjach, gdy ze względów *składniowych* wymagana jest instrukcja, lecz w praktyce program nie potrzebuje wykonywać żadnego działania. Instrukcja **`pass`** może Ci być zatem przydatna w prototypowaniu, gdy chcesz, aby np. dana pętla występowała już w programie (aby interpreter nie sygnalizował błędów składniowych), ale jeszcze nie chcesz podawać jej faktycznej implementacji (zob. przykł. 1.5).

Listing 1.5. Instrukcja **`pass`** nie robi nic...

```
for elem in (1, 2, 5):
  pass    # robocze ciało pętli
```

Dla zainteresowanych...

W sytuacji, gdy dana funkcjonalność nie została jeszcze w pełni zaimplementowana *i nie powinna być używana*, warto zaznaczyć ten fakt stosując zamiast instrukcji **`pass`** instrukcję **`raise`** `NotImplementedError()`, która rzuci stosowny wyjątek[10] – i tym samym przerwie dalsze wykonanie programu.

1.8 Definiowanie funkcji

Oto przykład definicji prostej funkcji sprawdzającej, czy dana liczba jest liczbą parzystą:

```
def is_even(n):
  return (n % 2 == 0)
```

[9] Wynika to stąd, że `range()` to tak naprawdę nie tyle zwykła funkcja, co generator.

[10] Mechanizm wyjątków został omówiony w rozdziale 5.2 Wyjątki.

Definicja funkcji rozpoczyna się słowem kluczowym **def**, po którym występuje nazwa funkcji i – w nawiasach okrągłych – lista parametrów, a na końcu znajduje się dwukropek. Ciało funkcji zaczyna się od kolejnej linijki i musi być wcięte. Do dobrych praktyk należy dokumentowanie działania funkcji za pomocą tzw. **docstringów** (ang. docstrings), czyli specjalnych komentarzy o ustalonej składni, które mogą być potem analizowane przez narzędzia wspomagające rozwój oprogramowania (np. przez IDE)[11].

Zmienne utworzone w ciele funkcji żyją tylko wewnątrz tej funkcji, natomiast wewnątrz funkcji widoczne są wszystkie zmienne widoczne w miejscu zdefiniowania funkcji (w szczególności – zmienne globalne).

W języku Python argumenty przekazywane są za pomocą mechanizmu **pass-by-assignment** – traktowane są jako referencje do momentu próby zmiany ich wartości, kiedy to zaczynają być traktowane jak przekazane przez wartość.

Do zilustrowania tego mechanizmu posłużymy się wbudowaną funkcją `id(obj)`, która zawraca unikalny identyfikator przypisany do obiektu `obj`, na który wskazuje dana referencja (ten nadany przez interpreter języka Pythona identyfikator nie ulega zmianie w trakcie cyklu życia obiektu).

Przyjmując poniższą definicję funkcji `ref_demo()`[12]:

```
def ref_demo(x):
    print("x={:2} id={}".format(x, id(x)))
    x = 42
    print("x={:2} id={}".format(x, id(x)))
```

otrzymamy następujący wynik skryptu:

```
>>> x = 9
>>> id(x)
41902552
>>> ref_demo(x)
x=  9  id= 41902552  # na początku `x` wskazuje na obiekt z miejsca wywołania
x= 42  id= 41903752  # potem `x` wskazuje na nowy obiekt (o wartości 42)
>>> id(x)
41902552  # po wyjściu z funkcji `x` znów wskazuje na obiekt o wartości 9
```

Funkcje w Pythonie *zawsze* zwracają wartość – domyślnie `None`. Wykonanie poniższego programu:

```
def no_return():
    pass  # W języku Python aby zdefiniować funkcję, która "nic nie robi"
          #   musimy użyć instrukcji `pass`.

print(no_return())
```

da następujący wynik

```
None
```

czyli mimo braku jawnej instrukcji „**return** `None`” interpreter i tak zwrócił `None`.

1.8.1 Argumenty domyślne

Poniższa funkcja ma dwa parametry, przy czym drugi posiada **argument domyślny** (ang. default argument):

```
def greet(n_loops, message=None):
    for i in range(n_loops):
        print(message if message is not None else '(-)')
```

W związku z tym funkcję `greet()` można wywołać na dwa sposoby:

- `greet(2)`
- `greet(2, 'Hi!')`

Wartość domyślna dla danego argumentu jest określana *tylko raz* – w momencie, gdy interpreter natrafia na definicję funkcji (a nie przy każdym jej wywołaniu), zatem poniższy program

[11] Zagadnienie docstringów zostało omówione dokładniej w rozdziale 6.4 Docstrings.

[12] Metoda `str.format()` została omówiona w rozdziale 1.4 Formatowanie wyjścia.

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

wypisze `5`.

Fakt, że wartość domyślna jest określana *tylko raz* (w momencie definicji funkcji) jest szczególnie ważny, gdy wartość domyślna jest typu mutowalnego! Poniższy program

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

wypisze

```
[1]
[1, 2]
[1, 2, 3]
```

gdyż obiekt będący domyślną wartością argumentu `L` – czyli lista – zostanie utworzony jeszcze przed pierwszym wywołaniem funkcji `f()`, a kolejne wywołania tej funkcji korzystają z tego pierwotnego obiektu (w tym przypadku modyfikując go poprzez dodanie elementu na koniec listy).

Jeśli nie chcesz takiego zachowania – czyli jeśli chcesz, aby przy każdym wywołaniu funkcji `f()` pominięcie argumentu odpowiadającego parametrowi `L` skutkowało utworzeniem *nowej* pustej listy – możesz zdefiniować funkcję `f()` w poniższy sposób:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print(f(1))
print(f(2))
```

Nadanie parametrowi `L` domyślnej wartości *niemutowalnej* sprawi, że późniejsze nadpisanie jej obiektem mutowalnym nie wpłynie na kolejne wywołania tej funkcji:

```
[1]
[2]
```

1.8.2 Argumenty pozycyjne i słownikowe

Funkcje mogą być wywoływane z **argumentami słownikowymi** (ang. keyword arguments) o postaci `kwarg=value`, przykładowo poniższa funkcja:

```
def get_travel_duration(distance, speed):
  return distance / speed
```

może być wywołana m.in. na poniższe dwa sposoby:

```
get_travel_duration(10.8, 30)                  # dwa argumenty pozycyjne
get_travel_duration(distance=10.8, speed=30)   # dwa argumenty słownikowe
```

Każdy argument nie będący argumentem słownikowym jest traktowany jako **argument pozycyjny** (ang. positional argument).

Pamiętaj, że argumenty słownikowe muszą być umieszczane po argumentach pozycyjnych:

```
get_travel_duration(10.8, speed=30)      # dopuszczalne
get_travel_duration(distance=10.8, 30)   # BŁĄD!
```

Natomiast kolejność samych argumentów słownikowych nie ma znaczenia, przykładowo:

```
get_travel_duration(distance=10.8, speed=30)
get_travel_duration(speed=30, distance=10.8)   # równoważne powyższemu
```

Argumenty słownikowe warto stosować wówczas, gdy poprawiają one czytelność kodu, czyli m.in. w poniższych sytuacjach:

- gdy ciężko określić „przeznaczenie” argumentu na podstawie nazwy wywoływanej funkcji, np. `text.splitlines(True)`
- gdy funkcja przyjmuje wiele argumentów, np.:

```
twitter_search('@obama', False, 20, True)   # mało czytelne
twitter_search('@obama', retweets=False,    # czytelne
    numtweets=20, popular=True)
```

albo kilka parametrów o wartościach domyślnych, np.:

```
def foo(n, p=None, q=None):
  ...

foo(1, q=2)    # odpowiada wywołaniu foo(1, None, 2)
```

Natomiast *nie należy* ich stosować m.in. w poniższych sytuacjach:

- gdy „przeznaczenie” argumentu jest oczywiste, np. `sin(2*pi)` albo `plot3d(x, y, z)`

W większości przypadków wymagane argumenty podaje się jako argumenty pozycyjne, a opcjonalne – jako słownikowe.

Dla zainteresowanych...

Materiały źródłowe:

- The Python Tutorial: Keyword Arguments (PyDocs)
- Any reason NOT to always use keyword arguments? (Stack Overflow)
- Clarify function calls with keyword arguments (by Jeff Paine)

1.8.3 Przeciążanie funkcji

W języku Python nie ma możliwości przeciążenia funkcji. Ponowne zdefiniowanie funkcji o tej samej nazwie (lecz tym razem z inną liczbą parametrów) spowoduje *nadpisanie* – czyli zastąpienie – pierwotnej definicji:

```
>>> def f(n):
...     return n + 42
...
>>> def f(n,m):
...     return n + m + 42
...
>>> f(3,4)
49
>>> f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes exactly 2 arguments (1 given)
>>>
```

Jeśli potrzebujemy zasymulować przeciążanie znane z języka C++, możemy uzyskać pożądany efekt za pomocą argumentów domyślnych (typ parametrów nie ma w języku Python znaczenia):

```
def f(n, m=None):
    if m:
        return n + m + 42
    else:
        return n + 42
```

1.9 Struktury danych

1.9.1 Listy

(Niniejszy rozdział rozszerza informacje z rozdziału 1.2.4 Listy.)

Aby dołączyć (pojedynczy) element do końca listy, skorzystaj z metody `list.append(x)`:

```
>>> cubes = [1, 8, 27]
>>> cubes.append(64)
>>> cubes
[1, 8, 27, 64]
```

Z kolei metoda `list.extend(iterable)` dołącza na koniec listy wszystkie elementy z kolekcji `iterable`:

```
>>> lst1 = [1, 2]
>>> lst2 = [3, 4]
>>> lst1.extend(lst2)
>>> lst1
[1, 2, 3, 4]
```

Ponieważ w języku Python dany obiekt kontenera może jednocześnie przechowywać obiekty różnych typów, wynik dołączania listy do listy jest następujący:

```
>>> lst1 = [1, 2]
>>> lst2 = [3]
>>> lst1.append(lst2)
>>> lst1
[1, 2, [3]]  # element #2 to LISTA
```

Możesz przypisać nowe wartości do całego *slice*'a, przy czym taka operacja może zmienić rozmiar listy (w szczególności – może spowodować usunięcie wszystkich elementów):

```
>>> letters = ['a', 'b', 'c', 'd']
>>> letters
['a', 'b', 'c', 'd']

>>> letters[1:3] = ['B', 'C']  # zmień kilka wartości
>>> letters
['a', 'B', 'C', 'd']

>>> letters[1:3] = []  # usuń kilka wartości
>>> letters
['a', 'd']

>>> letters[:] = []  # usuń wszystkie elementy - zastępując je pustą listą
>>> letters
[]
```

Usuwać elementy z listy można jeszcze na inne sposoby:

- Aby usunąć z listy pierwszy element o zadanej wartości `x`, skorzystaj z metody `list.remove(x)` [13].
- Aby usunąć element o zadanym indeksie (lub elementy z zadanego zakresu), użyj operatora **`del`** [14].

[13] Uwaga! Metoda `list.remove(x)` rzuca wyjątek, gdy lista nie zawiera ani jednego elementu o wartości `x`. Więcej o wyjątkach przeczytasz w rozdziale 5 Błędy i wyjątki.

[14] zob. rozdz. 1.10 Operator `del`

- Aby usunąć element o zadanym indeksie *ze zwracaniem* (czyli z możliwością zachowania elementu usuwanego z listy do dalszego wykorzystania w programie), użyj metody `pop([i])` (metoda przyjmuje indeks usuwanego elementu, domyślnie usuwa element z końca listy).
- Metoda `clear()` usuwa wszystkie elementy.

Dla zainteresowanych...

Pełną listę operacji udostępnianych przez typ `list` znajdziesz na stronie The Python Tutorial: 5. Data Structures.

1.9.2 Krotki

Krotka (ang. tuple) to niemutowalny typ sekwencyjny służący do przechowywania ciągu elementów różnego typu – ma zatem funkcjonalność zbliżoną do listy, z tym że bez możliwości zmieniania wartości elementów krotki po jej utworzeniu. W przeciwieństwie do listy, krotkę zwykle stosuje się do przechowywania elementów *różnego* typu. Krotkę definiuje się rozdzielając wartości przecinkami:

```
>>> t = 1, 5, 'abc'
>>> t[0]
1
>>> t
(1, 5, 'abc')
>>> # same krotki są niemutowalne:
... t[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # ale mogą zawierać obiekty mutowalne:
... v = ([1, 2], [3, 2])
>>> v
([1, 2], [3, 2])
```

Pustą krotkę definiuje się za pomocą `()`, natomiast zdefiniowanie krotki zawierającej pojedynczy element wymaga specjalnego zabiegu – dodania przecinka po elemencie (zob. przykł. 1.6). Brak przecinka spowoduje potraktowanie wyrażenia jako zwykłej wartości w nawiasach, a nie jako jednoelementowej krotki.

Listing 1.6. Tworzenie pustej krotki i krotki jednoelementowej

```
>>> empty = ()    # pusta krotka
>>> len(empty)
0
>>> singleton = ('hello',)     # <-- UWAGA: przecinek na końcu!
>>> len(singleton)
1
>>> singleton
('hello',)
>>> type(singleton)
<class 'tuple'>
>>>
>>> # Anty-przykład tworzenia jednoelementowej krotki.
>>> singleton_attempt = ('hello')  # <-- UWAGA: BRAK przecinka na końcu!
>>> type(singleton_attempt)
<class 'str'>
```

Rozpakowanie krotki (tuple unpacking)

Operacja **tuple unpacking** polega na „rozdzieleniu" krotki na poszczególne elementy (zob. przykł. 1.7), przy czym liczba zmiennych po lewej stronie wyrażenia musi być równa liczbie elementów w krotce[15].

[15] To odpowiednik funkcji `std::tie()` z języka C++.

Listing 1.7. Tuple unpacking.

```
>>> t = 1, 5, 'abc'
>>> x, y, z = t    # tuple unpacking
>>> print(x, y, z)
1 5 abc
```

Symbol podkreślenia (_)

Symbol podkreślenia (_) posiada w języku Python kilka różnych znaczeń[16], a jedno z przydatniejszych polega na ignorowaniu wartości – czyli stwierdzeniu „dana zwrócona wartość nas nie interesuje" – co jest szczególnie przydatne podczas rozpakowywania krotek.

Przykładowo, jeśli piszesz funkcję, która otrzymuje 3-elementową krotkę wyrażającą wartości współrzędnych punktu w przestrzeni trójwymiarowej, ale do dalszych obliczeń potrzebujesz wyłącznie wartości współrzędnych x i z, możesz to zapisać w poniższy sposób:

```
def foo(coord_3d):  # coord_3d: krotka wyrażająca współrzędne (x, y, z)
    x, _, z = coord_3d  # "zignoruj" drugi element krotki
```

1.9.3 Zbiory

Język Python udostępnia typ `set` realizujący funkcjonalność matematycznego zbioru (w zbiorze elementy nie powtarzają się), w szczególności umożliwia wykonywanie operacji związanych z algebrą zbiorów. Zbiór definiuje się za pomocą nawiasów klamrowych, rozdzielając elementy zbioru przecinkami, przy czym zbiór pusty definiuje się za pomocą funkcji `set()`[17] (zob. przykł. 1.8).

Listing 1.8. Tworzenie zbiorów i operacje na zbiorach.

```
>>> s1 = {1, 2}
>>> s2 = {2, 3}
>>> s1 | s2   # suma zbiorów
{1, 2, 3}
>>> s1 ^ s2   # różnica symetryczna zbiorów
{1, 3}
>>> set()     # pusty zbiór
{}
```

1.9.4 Słowniki

Język Python udostępnia wbudowany typ `dict` realizujący funkcjonalność **słownika**[18] (ang. dictionary), czyli skojarzonych ze sobą par „klucz i odpowiadająca mu wartość". Kluczem może być dowolny typ niemutowalny (zwykle: `str`, `int`). Klucze w ramach słownika są unikalne – jeśli w słowniku znajduje się już dany klucz, próba dodania pary zawierającej ten sam klucz spowoduje nadpisanie starej wartości skojarzonej z tym kluczem. Słownik definiuje się podając w nawiasach klamrowych rozdzieloną przecinkami listę par „klucz–wartość" (zob. przykł. 1.9)

Ponieważ słownik składa się z par *klucz–wartość*, iterować po słowniku można na trzy sposoby:

- Po wszystkich parach *klucz–wartość* (tj. elementach typu `tuple`):

  ```
  for key_value_tuple in d.items():
      # ...
  ```

  przy czym zwykle zależy nam na uzyskaniu osobno klucza, a osobno wartości skojarzonej z kluczem – wówczas można zastosować *tuple unpacking*:

[16] zob. Role of Underscore(_) in Python

[17] Zapis `{}` oznacza pusty słownik – typ danych realizujący kontener asocjacyjny (zob. rozdz. rozdz. 1.9.4 Słowniki).

[18] in. tablicy asocjacyjnej, mapy

Listing 1.9. Tworzenie słowników i operacje na słownikach.

```
>>> tel = {'jack': 4098, 'sape': 4139}  # utwórz słownik z dwoma wpisami
>>> tel['jack']          # odczytaj wartość skojarzoną z kluczem
4098
>>> tel['guido'] = 1000  # wstaw wartość (skojarzoną z kluczem)
>>> tel
{'sape': 4139, 'guido': 1000, 'jack': 4098}
>>> tel['guido'] = 4127  # zmień wartość skojarzoną z kluczem
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> list(tel.keys())     # zwróć listę kluczy występujących w słowniku
                         #  (w dowolnej kolejności)
['sape', 'guido', 'jack']
>>> 'guido' in tel       # sprawdź, czy klucz występuje w słowniku
True
```

```
for key, value in d.items():
    # ...
```

- Po wszystkich kluczach:

```
for key in d.keys():
    # ...
```

co jest równoważne poniższej (skróconej) formie:

```
for key in d:
    # ...
```

- Po wszystkich wartościach:

```
for key in d.values():
    # ...
```

1.9.5 Struktury zagnieżdżone

Listy, słowniki i krotki można zagnieżdżać. W przypadku struktur danych **zagnieżdżanie** (ang. nesting) oznacza definiowanie obiektu będącego strukturą danych, który zawiera inne obiekty-struktury (zob. przykł. 1.10).

Listing 1.10. Zagnieżdżanie struktur danych.

```
>>> nl = [['a', 'b'], [1, 2]]     # zagnieżdżone listy
>>> nl[0]
['a', 'b']
>>> nl[0][1]
'b'
>>>
>>> nt = ((1, 'a'), 5.7)          # zagnieżdżone krotki
>>> nd = {1:'x', 2:{'abc':10.8}}  # zagnieżdżony słownik
```

1.9.6 Modyfikacja struktur podczas iteracji

Jeśli wewnątrz pętli potrzebujesz dokonać modyfikacji ciągu po którym iterujesz (np. chcesz dodać lub usunąć pewne elementy), warto wcześniej utworzyć kopię takiego ciągu (zob. przykł. 1.11).
W przeciwnym razie w pętli z przykładu 1.11 wpadniemy w pętlę nieskończoną, która będzie dodawać wciąż nowe wyrazy `'windows'` do listy, po której następuje iteracja.

Listing 1.11. Chcąc modyfikować ciąg w czasie iteracji, iteruj po jego kopii.

```
>>> words = ['cat', 'window']
>>> for w in words[:]:   # iteruj po kopii całej listy
...     if len(w) > 4:
...         words.insert(0, w)
...
>>> words
['window', 'cat', 'window']
```

1.9.7 Kopiowanie obiektów

W języku Python zmienne domyślnie zachowują się tak, jak referencje w języku C++; mówimy o tzw. **aliasingu** (ang. aliasing), czyli sytuacji, gdy kilka zmiennych odnosi się do tego samego obiektu. Obiektowi wskazywanemu przez zmienną przydzielany jest osobny obszar pamięci dopiero wówczas, gdy przypiszemy do niej nową wartość (zob. przykł. 1.12).

Listing 1.12. Aliasing zachodzi, gdy kilka nazw odnosi się do tego samego obiektu.

```
>>> x = 3
>>> y = x   # aliasing
>>> print(id(x), id(y))   # id() zwraca unikalny identyfikator przypisany do
                          #   obiektu, na który wskazuje dana referencja
                          #   (nadany przez interpreter Pythona, nie ulega
                          #   zmianie w trakcie cyklu życia obiektu)
9251744 9251744
>>> y = 4   # zmiana obiektu - `y` odnosi się do nowego obszaru pamięci
>>> print(id(x), id(y))
9251744 9251776
```

Taki model zarządzania obiektami (tworzenie faktycznej kopii obiektu dopiero wówczas, gdy wymaga tego sytuacja) sprawia, że musisz zachować szczególną czujność podczas kopiowania typów mutowalnych – np. list czy słowników (zob. przykł. 1.13).

Listing 1.13. Aliasing dla typów mutowalnych.

```
>>> colours1 = ["red", "blue"]
>>> colours2 = colours1  # `colours2` i `colours1` to ALIASY
                         #   tego samego obiektu.
>>> print(id(colours1), id(colours2))
14603760 14603760
>>>
>>> colours2[1] = "green"  # Lista to typ mutowalny, zatem zmiana elementu
                           #   jest możliwa i nie powoduje utworzenia
                           #   nowego obiektu
>>> print(id(colours1), id(colours2))
14603760 14603760
>>> print(colours1)
['red', 'green']
>>> print(colours2)
['red', 'green']
```

W przypadku „płytkich” list (tj. niemających zagnieżdżonej struktury), jak w przykładzie 1.13, aby uzyskać faktyczną ich kopię wystarczy użyć *slicingu* (zob. przykł. 1.14).

Problem pojawia się w przypadku struktur zagnieżdżonych, gdyż *slicing* tworzy tzw. **płytką kopię** (ang. shallow copy), czyli dokładną kopię obiektu na poziomie poszczególnych *bitów* pamięci. Ponieważ w języku Python „wszystko jest referencją”, utworzenie takiej płytkiej kopii *zewnętrznej* struktury danych sprawia, że elementy w „kopii” w istocie wskazują na te same obiekty, co elementy w oryginalne (zob.

Listing 1.14. Kopiowanie płytkich list z użyciem *slice*'ingu.

```
>>> list1 = ['a', 'b', 'c', 'd']
>>> list2 = list1[:]  # Slicing powoduje utworzenie PŁYTKIEJ kopii.
>>> print(id(list1), id(list2))
14603474 14603496
>>>
>>> list2[1] = 'x'
>>> print(list2)
['a', 'x', 'c', 'd']
>>> print(list1)
['a', 'b', 'c', 'd']
```

przykł. 1.15). Problem ten nie występował w przykładzie 1.14, gdyż tam elementy kopiowanej listy były typu niemutowalnego (`str`). Rozwiązaniem tego problemu jest wykorzystanie funkcji `deepcopy()` z modułu

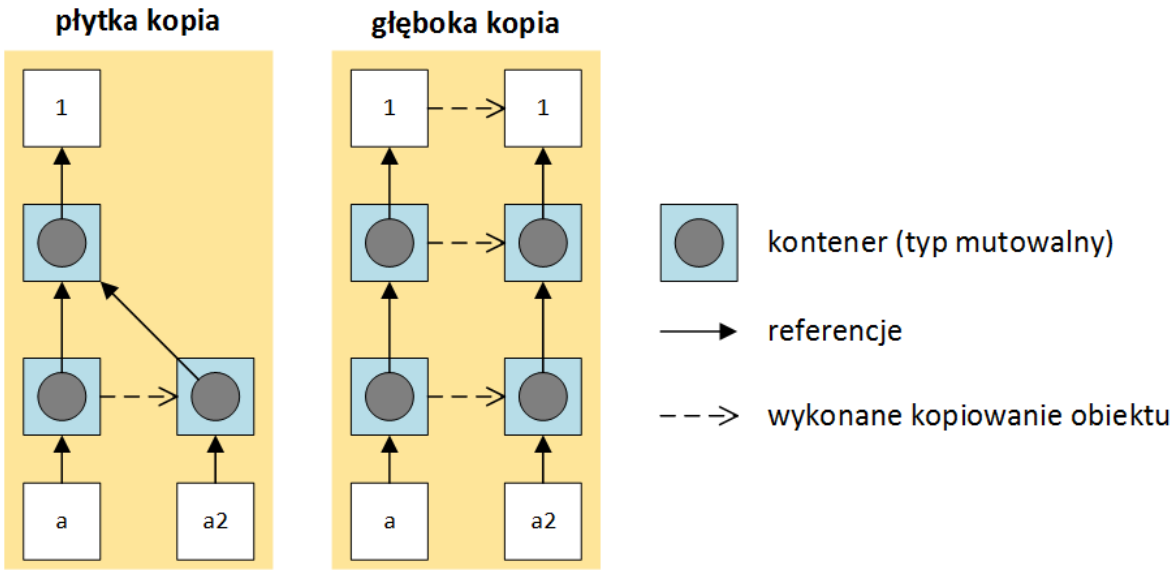
Listing 1.15. Kopiowanie list zagnieżdżonych z użyciem *slicingu* nie działa tak, jak byśmy mogli oczekiwać.

```
>>> lst1 = ['a', 'b', ['ab', 'ba']]  # zagnieżdżona lista
>>> lst2 = lst1[:]  # utworzenie PŁYTKIEJ kopii
>>> lst2[2][1] = 'd'   # Zmiana elementu struktury zagnieżdżonej
                       #   w obrębie PŁYTKIEJ kopii...
>>> print(lst1)
['a', 'b', ['ab', 'd']]
>>> print(lst2)
['a', 'b', ['ab', 'd']]
```

`copy`, która wykonuje tzw. **głęboką kopię** (ang. deep copy), co ilustruje przykład 1.16. Różnica w obu sposobach kopiowania została pokazana na Rys. 1.1.

Listing 1.16. Wykonywanie głębokiej kopii obiektu z użyciem `copy.deepcopy()`.

```
>>> from copy import deepcopy
>>>
>>> lst1 = ['a','b',['ab','ba']]
>>>
>>> lst2 = deepcopy(lst1)
>>>
>>> lst1
['a', 'b', ['ab', 'ba']]
>>> lst2
['a', 'b', ['ab', 'ba']]
>>>
>>> # W ramach każdej pary odpowiadających sobie elementów
>>> #   mają one różne identyfikatory.
>>> print(id(lst1), id(lst2))
139716507600200 139716507600904
>>> print(id(lst1[0]), id(lst2[0]))
139716538182096 139716538182096
>>> print(id(lst2[2]), id(lst1[2]))
139716507602632 139716507615880
```



Rysunek 1.1. Kopia płytka a głęboka w przypadku kontenerów zagnieżdżonych. W tym przykładzie kopiowany kontener to `[[1]]`: zmienna `a` („oryginał") wskazuje zatem na listę złożoną z jednego elementu, który jest referencją do innej listy (zawierającej *referencję* do obiektu `1`) – sam obiekt „wewnętrznej" listy jest przechowywany w innym miejscu w pamięci. W przypadku płytkiej obszar pamięci obiektu wskazywanego przez zmienną jest kopiowany bajt po bajcie, lecz kopiowanie ogranicza się wyłącznie do obszaru pamięci zajmowanego bezpośrednio przez wskazaną zmienną – zatem choć elementy-referencje w kontenerze zewnętrznym w kopii (zmienna `a2`) zajmują inne miejsce w pamięci, to jednak odnoszą się one do tych samych obiektów, co w elementy-referencje w oryginale (gdyż obszar pamięci „wewnętrznego" kontenera nie zostaje skopiowany). W efekcie, jeśli któryś z elementów kontenera zewnętrznego jest typu mutowalnego, zmiana wartości takiego elementu będzie widoczna we wszystkich (płytkich) kopiach tego kontenera zewnętrznego. W przypadku kopii głębokiej kopiowana jest *cała* struktura obiektów – czyli nie tylko obszar pamięci zajmowany bezpośrednio przez wskazywany obiekt, lecz również przez każdy z jego „podobiektów" (np. obiektów wskazywanych przez elementy kontenera) – zatem kopie `a` i `a2` są w pełni niezależne.

Drugą serię przykładów ilustrujących różnicę między kopiami płytkimi a głębokimi stanowią przykłady 1.17, 1.18, 1.19 i 1.20 – chcemy w nich uzyskać listę złożoną z dwóch niezależnych od siebie kopii tego samego początkowego elementu, który jest typu mutowalnego.

Listing 1.17. Wstawienie do struktury danych tego samego elementu – nie rozwiązuje problemu.

```
# Utwórz listę złożoną z jednego elementu MUTOWALNEGO.
nasty_list = [set()]

# Dołącz element na koniec listy bez wykonywania kopii.
nasty_list.append(nasty_list[0])

print('NL  (id={}):'.format(id(nasty_list)), nasty_list)

nasty_list[0].add(1)
# Modyfikacja elementu #0 spowodowała zmianę elementu #1, gdyż oba elementy
# listy to tak naprawdę referencje na ten sam obiekt w pamięci.
print('NL  (id={}):'.format(id(nasty_list)), nasty_list)
```

Dla zainteresowanych...

Więcej przykładów i schematy ilustrujące zasady kopiowania: Python 3 Tutorial: Shallow and Deep Copy

Listing 1.18. Wykonywanie *płytkiej* kopii *całej struktury danych* – nie rozwiązuje problemu.

```
nasty_list = [set()]
nasty_list.append(nasty_list[-1])

# Wykonaj płytką (!) kopię listy.
nasty_list_copy = nasty_list.copy()
nasty_list[0].add(2)
# W przypadku płytkiej kopii tworzony jest nowy obiekt, natomiast zawiera on
#   referencje (!) do oryginalnych referencji.
print('NL  (id={}):'.format(id(nasty_list)), nasty_list)
print('NLC (id={}):'.format(id(nasty_list_copy)), nasty_list_copy)
```

Listing 1.19. Wykonywanie *głębokiej* kopii *całej struktur danych* za pomocą `copy.deepcopy()` – nie rozwiązuje problemu.

```
nasty_list = [set()]
nasty_list.append(nasty_list[-1])

from copy import deepcopy

# Wykonaj głęboką (!) kopię listy.
nasty_list_deepcopy = deepcopy(nasty_list)
nasty_list[0].add(2)
# W przypadku głębokiej kopii tworzony jest nowy obiekt, zawierający
#   kopie (!) oryginalnych elementów...
print('NL   (id={}):'.format(id(nasty_list)), nasty_list)
print('NLDC (id={}):'.format(id(nasty_list_deepcopy)), nasty_list_deepcopy)

# ...natomiast w przypadku naszej listy elementy wciąż są powiązane - nawet
#   po wykonaniu głębokiej kopii - gdyż zostały one powiązane ze sobą już na
#   etapie wstawiania nowego elementu do listy.
nasty_list_deepcopy[0].add(1)
print('NLDC (id={}):'.format(id(nasty_list_deepcopy)), nasty_list_deepcopy)
```

Listing 1.20. Wykonywanie *głębokiej* kopii *wstawianego elementu* za pomocą `copy.deepcopy()` – rozwiązuje problem.

```
nasty_list_edc = [set()]
# Dopiero dołączanie głębokiej kopii elementu sprawi, że będą one od siebie
#   w pełni niezależne.
nasty_list_edc.append(deepcopy(nasty_list_edc[-1]))

print('NL-EDC:', nasty_list_edc)
nasty_list_edc[0].add(1)
print('NL-EDC:', nasty_list_edc)
```

1.9.8 Comprehensions

Konstrukcja **comprehension**, służąca do sprawnego inicjalizowania obiektów wybranych typów struktur danych, to „pythoniczny[19]” odpowiednik stosowanego przez matematyków sposobu definiowania zbiorów, ciągów i (dyskretnych) funkcji. Konstrukcję tę można stosować do tworzenia zbiorów, list i słowników.

[19]czyli zgodny z filozofią języka Python (zob. rozdz. 6.1 Zen of Python)

Dla zainteresowanych...

Materiały źródłowe i dodatkowe:

- 5.1.3. List Comprehensions (PyDocs)
- 5.1.4. Nested List Comprehension (PyDocs)
- Python 3 Tutorial: List Comprehension (Python Course)
- Common Mistake #6: Confusing how Python binds variables in closures (by Martin Chikilian)

Set comprehensions

Ogólna postać konstrukcji *set comprehension*, służącej definiowaniu zbiorów, to:

```
s = {expression for elem in collection [conditional]}
```

gdzie `expression` to *dowolne* wyrażenie (zwykle uwzględniające obiekt `elem`), `collection` to dowolna kolekcja obiektów, a na końcu znajduje się *opcjonalne* wyrażenie warunkowe (`conditional`) służące do „filtrowania" elementów z kolekcji (nawiasy kwadratowe zostały użyte wyłącznie w celu podkreślenia opcjonalności tego członu w ramach konstrukcji *comprehension*).

Przykładowo, w matematyce zbiór kwadratów liczb naturalnych od 0 do 9, S, definiowany jest następująco:

$$S = \{x^2 : x \in \{0, 1, \ldots, 9\}\}$$

W języku Python powyższy zbiór S można utworzyć z użyciem *set comprehension* w poniższy sposób:

```
S = {x**2 for x in range(10)}   # set comprehension
```

Z kolei z użyciem „zwykłej" pętli `for` powyższy zbiór można zdefiniować następująco:

```
S = set()
for x in range(10):
  S.add(x**2)
```

Analogicznie chcąc utworzyć zbiór M zawierający tylko liczby parzyste ze zbioru S:

$$M = \{x : x \in S \text{ jeśli } x \text{ parzyste}\}$$

w języku Python użyjemy następującej instrukcji:

```
M = {x for x in S if x % 2 == 0}  # `if x % 2 == 0` to wyrażenie warunkowe
```

Z kolei z użyciem „zwykłej" pętli `for` powyższy zbiór M można zdefiniować następująco:

```
M = set()
for x in S:
  if x % 2 == 0:
    M.add(x)
```

List comprehensions

Ogólna postać konstrukcji *list comprehension*, służącej definiowaniu list, jest analogiczna do *set comprehension*, z tym że zamiast nawiasów klamrowych stosujemy nawiasy kwadratowe:

```
l = [expression for elem in collection [conditional]]
```

Powiedzmy, że potrzebujemy zamienić listę temperatur wyrażonych w stopniach Celsiusza na stopnie Fahrenheita. Oto jak dokonać tego z użyciem *list comprehension*:

```
def fahrenheit_to_celsius(temperature: float) -> float:
    return (float(9)/5)*temperature + 32

t_celsius = [39.2, 36.5, 37.3, 37.8]
t_fahrenheit = [fahrenheit_to_celsius(t) for t in t_celsius]
# list comprehension
print(t_fahrenheit)
```

Z kolei z użyciem „zwykłej” pętli `for` powyższą listę można zdefiniować następująco:

```
t_fahrenheit = []
for t in t_celsius:
  t_fahrenheit.append(fahrenheit_to_celsius(t))
```

Dict comprehensions

Ogólna postać konstrukcji *dict comprehension*, służącej definiowaniu słowników, jest analogiczna do *set comprehension*, z tym że podajemy dwa wyrażenia – opisujące odpowiednio klucz i wartość – rozdzielone dwukropkiem:

```
d = {key_expression: value_expression for elem in collection [conditional]}
```

Przykładowo, poniższa instrukcja definiuje słownik, w którym kluczami są liczby naturalne 2, 4 i 6, a wartościami ich kwadraty (odpowiednio 4, 16 i 36):

```
d = {x: x**2 for x in (2, 4, 6)}  # równoważnie: d = {2: 4, 4: 16, 6: 36}
```

Z kolei z użyciem „zwykłej” pętli `for` powyższy zbiór można zdefiniować następująco:

```
d = {}
for x in (2, 4, 6):
  d[x] = x**2
```

1.10 Operator `del`

Operator **`del`** może być użyty w dwóch kontekstach – do usunięcia elementów z kolekcji albo do usunięcia całej zmiennej (czyli do „odpięcia” *aliasu* na pewien obiekt w pamięci), przykładowo:

- usuwanie elementów z listy – przez podanie indeksu albo zakresu

```
>>> a = [0, 1, 2, 3, 4]
>>> del a[0]  # usunięcie elementu #0
>>> a
[1, 2, 3, 4]
>>> del a[1:3]  # usunięcie elementów z zakresu <1,3) z powyższej listy
>>> a
[1, 4]
>>> del a[:]  # usunięcie wszystkich (pozostałych) elementów
>>> a
[]
```

- usuwanie elementu ze słownika – przez podanie klucza

```
>>> d = {'a':1, 'b':2, 'c':3}
>>> del d['b']  # usuń wpis związany z kluczem 'b'
>>> d
{'a': 1, 'c': 3}
```

- usuwanie zmiennej – przez podanie nazwy zmiennej; późniejsza próba odwołania się do usuniętej zmiennej skutkuje błędem (chyba, że w międzyczasie nadaliśmy jej nową wartość)

```
>>> a = 1
>>> del a  # "odepnij" alias `a` na obiekt `1`
>>> a
...
Traceback (most recent call last):
File "<input>", line 1, in <module>
NameError: name 'a' is not defined
```

W każdym z powyższych scenariuszy trzeba pamiętać, że operator **`del`** jedynie „odpina” *alias* na pewien obiekt w pamięci, a nie usuwa samego obiektu, przykładowo:

```
>>> a = b = 1
>>> print(id(a), id(b))
139685207020608 139685207020608
>>> del b  # "odpięcie" aliasu `b` na obiekt `1`
>>> id(a)  # alias `a` wciąż wskazuje na ten sam obiekt
139685207020608
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    print(b)
NameError: name 'b' is not defined
```

1.11 Wyrażenia lambda

Słowo kluczowe **lambda** służy do definiowania krótkich anonimowych funkcji, tzw. funkcji lambda, składających się z jednej instrukcji (takie ograniczenie długości to wymóg składni języka Python), np.:

```
lambda a, b: a + b   # anonimowa funkcja zwracająca sumę dwóch składników
```

Funkcje lambda mogą być używane wszędzie tam, gdzie oczekiwany jest obiekt funkcyjny. Podobnie do zagnieżdżonej definicji funkcji, funkcja lambda ma dostęp do zmiennych z zawierającego ją zakresu (zob. przykł. 1.21).

Listing 1.21. Przykład funkcji lambda.

```
>>> def make_incrementor(n):
...     return lambda x: x + n  # Zwraca obiekt funkcyjny zwracający wartość
...                             #   parametru `x` powiększoną o pewną stałą.
>>> f = make_incrementor(42)
>>> f(1)  # Argument `1` odpowiada parametrowi `x`.
43
```

1.12 Dobre praktyki

1.12.1 Style guide

W programowaniu liczy się nie tylko znajomość możliwości danego języka, ale również m.in. stosowanie czytelnego zapisu (np. formatowanie kodu, nazewnictwo) oraz stosowanie się do konwencji. Istotne jest też czytelne komunikowanie przez programistę jego intencji oraz założeń odnośnie tworzonej funkcjonalności programu (np. funkcji, klas).

Poniższe odnośniki stanowią obowiązkową lekturę dla każdego programisty Pythona, który chce pisać kod „w duchu Pythona”:

- PEP 8 – Style Guide for Python Code
- The Best of the Best Practices (BOBP) Guide for Python

1.12.2 Podpowiedzi typów (type hinting)

Jedna z podstawowych zasad filozofii języka Python głosi, że „jawne jest lepsze niż niejawne” (ang. „*explicit is better than implicit*”). Mechanizm **podpowiedzi typu** (ang. type hinting) pozwala na realizację tej zasady w odniesieniu do typów zmiennych, parametrów funkcji oraz obiektów zwracanych przez funkcję – umożliwia jawne określenie, z jakiego typu obiektu będziemy korzystać, lub jakiego typu obiektu oczekujemy (zob. przykł. 1.22):

Ze względu na występujące w języku Python dynamiczne typowanie, próba określenia typu zmiennej w sposób automatyczny (przez narzędzia do statycznej analizy kodu wykorzystywane np. w IDE) kończy się powodzeniem zaledwie w ok. 50% przypadków. Określenie typu nastręcza problemów również samym programistom, zwłaszcza gdy przychodzi im analizować cudzy kod.

Listing 1.22. Przykład podpowiedzi typu.

```
def is_odd(n: int) -> bool:    # podpowiedź typu parametru `n` (`: int`)
                               #   i wartości zwracanej z funkcji (` -> bool`)
    return n % 2 != 0

r: bool = is_odd(3)    # podpowiedź typu zmiennej
```

Korzystanie z podpowiedzi typu daje zatem szereg korzyści:

- pozwala narzędziom kontroli typu na wychwycenie błędów w programie (np. przekazanie do funkcji argumentu niedozwolonego typu)
- pełni rolę dokumentacji (przydatne dla użytkowników naszego kodu)
- poprawia efektywność korzystania z IDE (które może np. oferować lepsze podpowiedzi autouzupełniania)

Warto jednak podkreślić, że choć korzystanie z podpowiedzi typów jest bardzo wskazane, nie jest to rzecz *wymagana* w programie przez sam język Python – brak takich podpowiedzi nie jest błędem.

Oto podstawowe zasady stosowania podpowiedzi typów:

- Podpowiedź typu dla zmiennej lokalnej (a więc również parametru funkcji) określamy po dwukropku umieszczonym *bezpośrednio* po nazwie zmiennej (tj. przed ewentualnym znakiem „=", także w przypadku argumentów domyślnych):

```
def foo(a: int, b: bool = False):
    pass

x: float = 0.3
```

- Podpowiedź typu dla wartości zwracanej przez funkcję określamy po kombinacji „`->`" umieszczonej *bezpośrednio* po liście parametrów funkcji (tj. przed dwukropkiem kończącym wiersz z nagłówkiem funkcji:

```
def foo() -> int:
  return 1
```

- W celu określenia, że dana wartość będzie kontenerem, stosuje się specjalne podpowiedzi z modułu `typing`. Zwykle nazwy tych typów odpowiadają nazwom typów wbudowanych, tyle że są zapisane dużą literą:

```
from typing import Tuple, List, Dict, Set

l: List[int] = [1, 2]                  # List[elem_type]
t: Tuple[float, str] = (5.6, 'abc')    # Tuple[elem1_type, elem2_type, ...]
d: Dict[int, str] = {1: 'a', 2: 'b'}   # Dict[key_type, value_type]
s: Set[int] = {1, 2}                   # Set[elem_type]
```

  przy czym podpowiedzi można „zagnieżdżać" – przykładowo, podpowiedź typu dla słownika o kluczach typu całkowitego i wartościach z nimi skojarzonych typu „lista elementów typu logicznego" będzie mieć postać `Dict[int, List[bool]]`.
- Jeśli dana wartość może być „pewną wartością typu `T` *albo wartością* `None`, zastosuj podpowiedź `Optional[T]`.
- Możesz w prosty sposób definiować własne „typy" podpowiedzi typów korzystając z funkcji `NewType()` – przykładowo, podpowiedź typu dla dwuwymiarowej tablicy (in. macierzy) liczb całkowitych, reprezentowanej za pomocą listy list, możesz zdefiniować jako:

```
Matrix = NewType('Matrix', List[List[int]])
m: Matrix = [[1, 2], [3, 4]]  # użycie własnego "typu"
```

  Czasem warto zdefiniować alias dla istniejącego typu, jeśli w danym kontekście ma on specjalne znaczenie semantyczne – przykładowo, typ całkowity może służyć do wyrażania zarówno liczby elementów kontenera (jego rozmiaru), jak i okresu (różnicy między dwoma momentami w czasie),

jednak od strony *semantycznej* wartość 1 w odniesieniu do rozmiaru oznacza coś innego, niż 1 w przypadku okresu:

```
Size = NewType('Size', int)
Period = NewType('Period', int)
s: Size = Size(1)
p: Period = Period(1)
```

Wówczas narzędzie do statycznej kontroli typów ostrzeże Cię przed sytuacjami, w których istnieje ryzyko pomieszania ze sobą danych reprezentujących różne *semantycznie* wartości:

```
x = s + p  # OSTRZEŻENIE: dodawanie wartości różnych typów
```

mimo, że obie wartości są w pamięci komputera wyrażane za pomocą typu całkowitego `int`.

Dla zainteresowanych...

Więcej informacji o efektywnym i precyzyjnym stosowaniu podpowiedzi typów przeczytasz tu:

- `typing` — Support for type hints
- PEP 484 – Type Hints
- PEP 526 – Syntax for Variable Annotations
- Type Hints – Guido van Rossum (PyCon 2015)

1.13 `if __name__ == "__main__"`

Gdy interpreter języka Python odczytuje plik źródłowy, dokonuje dwóch rzeczy:

- ustawia kilka zmiennych pomocniczych (np. `__name__`) oraz
- wykonuje wszystkie polecenia znalezione w pliku.

Gdy dany moduł (plik `.py`) zostaje uruchomiony jako główny program, np. poleceniem

```
python foo.py
```

wówczas na czas interpretacji tego modułu interpreter przypisuje zmiennej `__name__` predefiniowany łańcuch znaków `"__main__"`.

Z kolei gdy dany moduł (tu: moduł `pkg`) zostaje zaimportowany przez inny moduł, np. poleceniem

```
import pkg
```

wówczas na czas interpretacji tego modułu interpreter przypisuje zmiennej `__name__` nazwę modułu – czyli w tym przykładzie łańcuch znaków `"pkg"`.

Zatem dodając w programie warunek:

```
if __name__ == "__main__":
  # ...
```

sprawiamy, że kod nim objęty zostanie wykonany tylko wówczas, gdy dany moduł zostanie wykonany jako program główny – co przydaje się choćby do uruchamiania testów jednostkowych, gdy moduł jest tylko importowany (testy nie powinny być uruchomione podczas „normalnego” wykonania programu przez użytkownika końcowego):

```
import unittest

if __name__ == "__main__":
  unittest.main()  # Wykonaj testy tylko wówczas, gdy moduł
                   #   uruchomiony jako główny program.
```

1.14 `*args` i `**kwargs`

Funkcja może (ale nie musi) posiadać choć jeden z dwóch specjalnych parametrów zapisywanych odpowiednio jako `*args` oraz `**kwargs`[20]:

- `args` przechowuje krotkę złożoną z niewykorzystanych argumentów pozycyjnych – czyli tych, które „nie mieszczą się” w liście „regularnych” parametrów (o nazwach nie poprzedzonych gwiazdkami)

[20] od ang. *keyword arguments* – argumenty słownikowe

- `kwargs` przechowuje słownik złożony z niedopasowanych argumentów słownikowych – czyli tych argumentów słownikowych, których klucz nie odpowiada żadnej z nazw parametrów

przy czym na liście parametrów `*args` musi występować przed `**kwargs`, gdyż argumenty pozycyjne zawsze są przekazywane przed argumentami słownikowymi.

Przykładowo, dla poniższej funkcji

```
def foo(x, y, *args, **kwargs):
  print('args:', args)
  print('kwargs:', kwargs)
```

w zależności od wywołania otrzymasz takie przykładowe wyniki:

- gdy liczba argumentów pozycyjnych dokładnie odpowiada liczbie „regularnych" parametrów...

```
>>> foo(1, 2)
args: ()
kwargs: {}
```

- gdy nadmiar argumentów pozycyjnych...

```
>>> foo(1, 2, 3)
args: (3,)
kwargs: {}
```

- gdy niedopasowane argumenty słownikowe...

```
>>> foo(1, 2, q=0)
args: ()
kwargs: {'q': 0}
```

- gdy nadmiar argumentów pozycyjnych i niedopasowane argumenty słownikowe...

```
>>> foo(1, 2, 3, q=0)
args: (3,)
kwargs: {'q': 0}
```

- gdy niedomiar argumentów pozycyjnych, ale choć część argumentów słownikowych dopasowana...

```
>>> foo(1, y=2, q=0)
args: ()
kwargs: {'q': 0}

>>> foo(x=1, y=2, q=0)
args: ()
kwargs: {'q': 0}
```

Użycie powyższych specjalnych parametrów, w połączeniu z dwoma operatorami:

- `*` – rozpakowanie kontenera sekwencyjnego (listy, krotki)
- `**` – rozpakowanie kontenera asocjacyjnego (słownika)

pozwala m.in. definiować tzw. dekoratory[21] oraz efektywnie wywoływać metody inicjalizacyjne dla klas w hierarchii dziedziczenia[22].

Oto przykład dekoratora, czyli generycznej funkcji „opakowującej" wywołanie innej funkcji w celu wykonania pewnych operacji przed albo po wywołaniu pewnej innej funkcji – zwróć uwagę na zastosowanie parametru `*args` w celu akceptowania dowolnej liczby argumentów pozycyjnych:

```
from typing import Callable

def check_positive(func: Callable) -> Callable:
    """Sprawdź, czy wszystkie argumenty są liczbami dodatnimi."""
    def func_wrapper(*args):
        for arg_inx, arg in enumerate(args):
            if type(arg) is int or type(arg) is float:
```

[21] zob. rozdz. 6.3 Dekoratory
[22] zob. rozdz. 3.1.2 Dziedziczenie

```
                if arg <= 0:
                    raise Exception(f'Function `{func.__name__}()` takes only'
                                    f' positive arguments (arg'
                                    f' #{arg_inx} = `{arg}`).')
            else:
                raise Exception(f'Arguments of `{func.__name__}()` must be'
                                f' numbers (T[arg #{arg_inx}] = {type(arg)}).')
        return func(*args)  # rozpakuj krotkę `args` jako argumenty pozycyjne

    return func_wrapper  # funkcja zwracająca inną funkcję

Number = Union[int, float]

@check_positive  # "Opakuj" wywołania funkcji `area` w wywołania
                 #   funkcji `check_positive`.
def area(base: Number, height: Number):
    """Oblicz pole trójkąta o zadanych długościach podstawy i wysokości."""
    return base * height / 2
```

W zależności od przekazanych argumentów, wywołanie funkcji `area()` da następujące wyniki:

```
>>> print(area(1, 2))
1.0

>>> print(area(-1, 2))
Exception: Function `area()` takes only positive arguments (arg #0 = `-1`).

>>> print(area(-1, 'x'))
Exception: Arguments of `area()` must be numbers (T[arg #1] = <class 'str'>).
```

Rozdział 2

Klasy: Podstawy

Dla zainteresowanych...

Materiały źródłowe:

- The Python Tutorial: Classes (PyDocs)
- Python 3 Tutorial (Python Course)
- Object-Oriented Programming in Python: Classes
- Improve Your Python: Python Classes and Object Oriented Programming (by Jeff Knupp)

2.1 Wprowadzenie

Klasa to podstawowy element konstrukcyjny kodu w języku Python, pełniący identyczną funkcję, co klasy w języku C++ – grupują dane i funkcje operujące na tych danych. Co jednak ważne, w języku Python *wszystko* jest instancją jakiejś klasy, każdy obiekt dziedziczy (bezpośrednio albo pośrednio) po wbudowanej klasie `object`[1]. Typy klasowe i typy wbudowane same w sobie również są obiektami – instancjami klasy `type`. Możesz uzyskać informację o typie danego obiektu z użyciem funkcji wbudowanej `type()`:

```
type(any_object)
```

Oto przykład ilustrujący fakt, że *wszystko* (nawet typy wbudowane i klasowe) dziedziczą po wbudowanej klasie `object`:

```
>>> isinstance(int, object)
True
>>> isinstance(type(object), object)
True
```

W języku Python dane przechowywane wewnątrz obiektu nazywamy **atrybutami**[2] (ang. attributes), a funkcje powiązane z obiektami – **metodami** (ang. methods).

Przykład 2.1 pokazuje definicję prostej klasy służącej do przechowywania danych osobowych oraz sposób wykorzystania takiej klasy. Poszczególne elementy klasy zostały omówione w kolejnych rozdziałach.

Przy czym minimalna definicja klasy ma poniższą postać:

```
# definicja minimalnej klasy
class MinimalClass:
  pass  # Ciało klasy nie może być puste - musi składać się z co najmniej
        #   jednej instrukcji.
```

2.2 Parametr `self`

Podczas wywoływania metody potrzebujemy w jej ciele dostępu do informacji, na rzecz jakiego obiektu ta metoda została wywołana[3]. W języku Python taka informacja przekazywana jest jawnie, jako pierwszy

[1] We wcześniejszych wersjach języka Python istniał podział na typy wbudowane i klasy zdefiniowane przez użytkownika, lecz obecnie podział ten zanikł (natomiast podobny podział wciąż istnieje w języku C++).

[2] W języku C++ takie dane nazywamy *polami*.

[3] W języku C++ służy do tego niejawny wskaźnik `this`, którego programista nigdzie sam nie definiuje – jest to rolą kompilatora.

Listing 2.1. Przykład definiowania i korzystania z klasy.

```
import datetime

# definicja klasy
class Person:

  # definicja metody inicjalizującej instancję klasy
  def __init__(self, name, surname, birthdate):
    # zdefiniowanie trzech atrybutów
    self.name = name
    self.surname = surname
    self.birthdate = birthdate

  # definicja metody związanej z daną instancją
  def age(self):
    today = datetime.date.today()
    age = today.year - self.birthdate.year

    if today < datetime.date(today.year, self.birthdate.month,
        self.birthdate.day):
      age -= 1

    return age

# konstruowanie obiektu
person = Person("Jane", "Doe", datetime.date(1992, 3, 12))

# dostęp do atrybutów i metod
print(person.name)
print(person.age())
```

parametr metody[4]. Choć parametr ten nie musi nazywać się `self`, taka została przyjęta powszechna konwencja. Sam *argument* `self` jest najczęściej przekazywany niejawnie – choć teoretycznie możemy wywołać „pełną” wersję metody, w której przekażemy go jawnie (zob. przykł. 2.2).

Listing 2.2. Przekazywanie argumentu dla parametru `self`.

```
person = Person("Jane", "Doe", datetime.date(1992, 3, 12))

# przekazywanie niejawne -- poprzez wywołanie metody dla instancji klasy
#   => sposób PREFEROWANY
print(person.age())

# przekazywanie jawne -- poprzez wywołanie metody dla typu klasowego
#   => sposób DOPUSZCZALNY, lecz mniej czytelny
print(Person.age(person))
```

Ponieważ wszelkie odwołania do obiektu możliwe są jedynie z użyciem `self`, w Pythonie zwykło nazywać się parametry metod (np. konstruktora) identycznie jak nazwy pól – mamy bowiem gwarancję, że nie wystąpi znany z języka C++ problem kolizji oznaczeń.

Ważne

Pierwszy parametr metody często nazywa się `self`. To tylko konwencja, nazwa `self` nie ma w języku Python absolutnie żadnego specjalnego znaczenia. Jednak nie stosując się do tej konwencji uczynisz swój program mniej czytelnym dla innych programistów języka Python, a także utrudnisz pracę

[4] Z wyjątkiem metod statycznych i klasowych, o czym później. . .

narzędziom typu „przeglądarka klas” dostępnym w Twoim IDE (które często polegają na wspomnianej konwencji).

Dla zainteresowanych...

W przypadku poniższej klasy i jej instancji

```
class MyClazz:
  def foo(self, x):
    pass

obj = MyClazz()
```

stosowany powszechnie zapis wywołania metody, `obj.foo(x)`, oznacza tak naprawdę `MyClazz.foo(obj, x)`; proces takiej zamiany dokonywany jest automatycznie przez interpreter języka Python.

2.3 Metoda `__init__`

Metoda `__init__()` służy do *inicjalizacji* składowych utworzonego obiektu (pełni rolę analogiczną do połączenia listy inicjalizacyjnej i ciała konstruktora w języku C++).

Ważne

Metodę `__init__()` czasem określa się mianem „konstruktora”, lecz jest to błędne od strony technicznej – metoda ta jest tak naprawdę *inicjalizatorem*. Za sam proces tworzenia obiektu (przydzielania mu zasobów) odpowiada metoda `__new__()`, której w typowych sytuacjach nie musisz przeciążać[5]. W momencie wywołania metody `__init__()` obiekt został już utworzony, zatem możemy się do niego odwoływać poprzez `self`.

Dla zainteresowanych...

Więcej o roli `__new__()` i `__init__()`: Understanding `__new__` and `__init__`

2.3.1 Zapewnianie niezmienników klasy

Po wykonaniu metody `__init__()` użytkownik może (słusznie) zakładać, że obiekt jest gotowy do użytku. Rozważ (anty)przykład 2.3, w którym definiujemy klasę do obsługi depozytu bankowego.
Używając tak zdefiniowanej klasy `Account` należy pamiętać, aby – zgodnie z „dokumentacją” klasy – przed dokonaniem pierwszej wpłaty/wypłaty wywołać metodę `set_balance()` w celu utworzenia atrybutu `self.balance`. Nie ma jednak żadnej możliwości, aby *wymusić* na użytkownikach tej klasy stosowania się do takiego rozwiązania, przez co łatwo o błędy wykonania programu (nowo utworzone obiekty klasy `Account` nie będą „w pełni” zainicjalizowane).

Dobre praktyki

*Do dobrych praktyk należy definiowanie atrybutów klasy **wyłącznie** w ciele metody `__init__()`, w przeciwnym razie użytkownik klasy dostanie obiekt nie w pełni zainicjalizowany. Zdefiniowanie wszystkich atrybutów wewnątrz metody `__init__()` zwiększa również czytelność kodu.*

Ta reguła odnosi się do szerszego pojęcia **spójności obiektu** (ang. object consistency): nie powinien istnieć taki ciąg wywołań metod, które wprowadzą obiekt w stan, który nie ma sensu (np. ujemne prawdopodobieństwo). Te tzw. **niezmienniki** (ang. invariants) powinny być spełnione zarówno w chwili rozpoczynania wykonania metody, jak i jej opuszczania – samo wywoływanie metod nie może prowadzić do naruszenia niezmienników. Oczywiście w tym celu musimy zapewnić, że już na samym początku (tj. po wywołaniu metody `__init__()`) obiekt znajduje się w poprawnym stanie, stąd zasada inicjalizacji wszystkich atrybutów wewnątrz metody `__init__()`.

[5] O sytuacjach, w których możesz rozważać przeciążenie `__new__()` przeczytasz na Stack Overflow

Listing 2.3. (Anty)przykład definiowania klasy – brak zapewnienia jej niezmienników.

```
class Account:
  # UWAGA: Poniższa implementacja klasy `Account` wymaga, aby przed
  #   pierwszym użyciem metod związanych z obsługą rachunku (tj. withdraw()
  #   lub deposit()) wywołać metodę `set_balance()` inicjalizującą rachunek.

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

    def set_balance(self, balance=0.0):
        self.balance = balance  # utwórz atrybut `balance`

    def withdraw(self, amount):
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance.')
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance
```

2.4 Kontrola dostępu

W języku Python formalnie nie istnieje pojęcie kontroli dostępu do składowych – wszystkie składowe są zawsze dostępne dla użytkowników klasy (a więc „publiczne"). Istnieją jednak dwie konwencje nazywania atrybutów, które informują o pożądanym poziomie dostępu[6]:

- nazwy zaczynające się od pojedynczego podkreślenia (np. `_protected_attr`) odpowiadają polom chronionym (dostępnym tylko w obrębie klasy definiującej ten atrybut oraz jej klas pochodnych), natomiast
- nazwy zaczynające się od dwóch podkreśleń (np. `__private_attr`) odpowiadają polom prywatnym (dostępnym wyłącznie w obrębie klasy definiującej ten atrybut).

Atrybut powinien być "prywatny" wyłącznie wtedy, gdy potrzebujemy dokonać weryfikacji lub transformacji danych (np. w celu zapewnienia niezmienników klasy).

Z zagadnieniem kontroli dostępu związany jest również dekorator `@property`[7], który w język Python stanowi rozwiązanie problemu konieczności definiowania specjalnych par metod zapewniających dostęp do danych w trybach odpowiednio „tylko do odczytu" oraz „w celu modyfikacji"[8] – co do zasady w języku Python nie ma konieczności definiowania takich metod.

2.5 Atrybuty i metody

W poniższym rozdziale pokazane zostanie jak podczas definiowania klas wykorzystać kilka wbudowanych **dekoratorów**, czyli funkcji „opakowujących" inne funkcje i zmieniające ich działanie – co pozwala na efektywniejsze definiowanie klas. (Na chwilę obecną nie będzie Ci potrzebna dokładna znajomość działania dekoratorów[9]).

2.5.1 Atrybuty klasowe

Oprócz **atrybutów powiązanych z konkretną instancją klasy** (ang. instance attributes), definiowanych w metodzie `__init__()`, możemy także definiować atrybuty powiązane z samą klasą – te tzw. **atrybuty klasowe** (ang. class attributes) zachowują się podobnie do znanych z języka C++ statycznych pól klasy.

[6] zob. The Python Tutorial: Classes, Private, protected and public in Python (radek.io)
[7] zob. rozdz. 2.5.3 Właściwości
[8] Czyli tzw. *getterów* i *setterów* – zob. rozdz. 2.5.3 Właściwości.
[9] Dekoratory zostaną omówione w rozdziale 6.3 Dekoratory.

Atrybuty klasowe definiuje się poza wszelkimi metodami, zwykle na samym początku ciała klasy – tuż pod nagłówkiem klasy (zob. przykł. 2.4).

Listing 2.4. Przykład definiowania i korzystania z atrybutów klasowych.

```
class Car:
    wheels = 4    # atrybut klasowy

    def __init__(self, make, model):
        self.make = make
        self.model = model

mustang = Car('Ford', 'Mustang')

print(mustang.wheels)   # Dostęp do atrybutu klasowego poprzez instancję klasy.
                        #    (= dostęp "tylko do odczytu")
print(Car.wheels)       # Dostęp do atrybutu klasowego poprzez typ klasowy.
                        #    (= dostęp umożliwiający i odczyt, i modyfikację)
```

Metody powiązane z instancjami klasy[10] mogą odwołać się do wartości atrybutów klasowych w sposób identyczny jak do „zwykłych" atrybutów – za pomocą `self` (np. `self.wheels` dla klasy `Car` z przykł. 2.4). Jest to dostęp „tylko do odczytu", gdyż próba zmiany wartości atrybutu klasowego np. w poniższy sposób:

```
class Car:
    wheels = 4    # atrybut klasowy

  # ...

    def try_to_change_wheels(self):
        self.wheels = 5

mustang = Car('Ford', 'Mustang')
mustang.try_to_change_wheels()
print(mustang.wheels)
print(Car.wheels)
```

spowoduje wypisanie na konsolę:

```
5
4
```

Jak widzisz, próba przypisania wartości do `self.wheels` powoduje nie tyle zmianę atrybutu klasowego, co utworzenie *nowego* atrybutu *powiązanego z instancją klasy* – przy czym atrybut powiązany z instancją klasy zaczyna przesłaniać atrybut klasowy. Aby faktycznie zmienić wartość atrybutu klasowego musisz użyć notacji `ClassName.attribute_name = new_value`, czyli w tym przypadku `Car.wheels = 5`.

2.5.2 Metody klasowe

Metoda klasowa (ang. class method) w języku Python jest powiązana z klasą (a nie z instancją klasy) i umożliwia wyłącznie dostęp do atrybutów klasowych[11]. Do definiowania metod klasowych służy dekorator `@classmethod`.

Typowy przykład użycia metody klasowej to tzw. **metoda wytwórcza** (ang. factory method), służąca do konstruowania obiektów danej klasy po dokonaniu pewnej wstępnej obróbki otrzymanych danych – dzięki temu kod metody inicjalizacyjnej jest bardziej czytelny (zob. przykł. 2.5).

Metoda klasowa, podobnie jak metoda związana z instancją klasy, jako pierwszy argument przyjmuje obiekt, na rzecz którego została wywołana. Metodę klasową możemy wywołać na dwa sposoby – albo poprzez instancję klasy, albo poprzez samą klasę – jednak w obu przypadkach pierwszy argument będzie

[10] Z innym rodzajem metod, tj. z metodami powiązanymi z typem klasowym, zapoznasz się w rozdziale 2.5.2 Metody klasowe.

[11] Metoda klasowa zachowuje w zbliżony sposób do metody statycznej w języku C++.

Listing 2.5. Metoda wytwórcza.

```
from datetime import date

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

    @classmethod
    def from_birth_year(cls, name, birth_year):
        return cls(name, date.today().year - birth_year)
```

obiektem wyrażającym typ klasowy. W związku z tym zwyczajowo pierwszy parametr metody klasowej nazywa się `cls` (a nie `self`) – chodzi o podkreślenie tego, że obiekt przekazany jako pierwszy argument nie umożliwia dostępu do atrybutów związanych z instancją (zob. przykł. 2.6)

Listing 2.6. Sposoby wywołania metody klasowej.

```
p = Person("Jan", 20)

p1 = Person.from_birth_year("Adam", 1992)   # preferowany sposób
p2 = p.from_birth_year("Adam", 1992)        # dopuszczalny sposób
```

Dla zainteresowanych. . .

Każde umieszczone w kodzie wywołanie metody klasowej, np. `from_birth_year()` z powyższego przykładu:

```
p = Person.from_birth_year("Adam", 1992)
```

jest *wewnętrznie* zamieniane przez interpreter języka Python na poniższą formę:

```
from_birth_year(Person, "Adam", 1992)
```

a zatem parametr `cls` ma wartość `Person` i tym samym metoda ta zwraca tak naprawdę nowo utworzony obiekt klasy `Person`:

```
class Person:

  # ...

 @classmethod
 def from_birth_year(cls, name, birth_year):  # wartością `cls` jest
                                              #   typ `Person`
     return cls(name, date.today().year - birth_year)
     # równoważne:
     #   return Person(name, date.today().year - birth_year)
```

2.5.3 Właściwości

Rozważ następujący przypadek: Początkowo zaprojektowano klasę `Person` posiadającą jeden atrybut – „pełne imię i nazwisko” (`fullname`) – do którego użytkownicy klasy `Person` odwoływali się bezpośrednio. Okazało się jednak, że klasę należy przeprojektować w taki sposób, aby trzymać imię i nazwisko w osobnych atrybutach (`name` i `surname`):

```
# dotychczasowy dostęp
person.fullname = "John Smith"

# nowy dostęp
```

```
person.name = "John"
person.surname = "Smith"
```

W wielu językach programowania (np. C++, Java) taka zmiana projektowa powoduje naruszenie interfejsu programistycznego (API) i wymusza gruntowną refaktoryzację kodu. W związku z tym języki te (zwłaszcza Java) zachęcają do stosowania prywatnych atrybutów oraz tzw. **getterów** (ang. getters) i **setterów** (ang. setters)[12] – specjalnych metod służących odpowiednio wyłącznie do odczytu oraz wyłącznie do modyfikacji danych przechowywanych jako składowe. Ponieważ pełnią one rolę pośredników w dostępie do danych (stanowią warstwę abstrakcji), można dowolnie zmieniać reprezentację danych wewnątrz klasy w sposób niezauważalny dla użytkowników tej klasy – innymi słowy, *zmiana implementacji nie pociąga za sobą zmiany interfejsu*.

Użycie setterów bywa jednak niewygodne, gdyż m.in. uniemożliwia zastosowanie złożonego operatora przypisania, co czyni zapis mniej przejrzystym (zob. przykł. 2.7). Oczywiście można by napisać specjalną metodę `change_height()` przyjmującą jako parametr zmianę wzrostu w centymetrach, lecz to by tylko niepotrzebnie „zaciemniało" interfejs klasy.

Listing 2.7. Enkapsulacja z użyciem setterów – rozwiązanie mało wygodne...

```
class Person:
    def __init__(self, height):
        self.__height = height

    def get_height(self):
        return self.__height

    def set_height(self, height):
        self.__height = height

jane = Person(153)  # początkowy wzrost Jane to 153 cm

jane.set_height(jane.get_height() + 1)  # Jane rośnie o 1 cm - mało wygodne
# jane.height += 1  # Wolelibyśmy móc użyć takiego zapisu...
```

Wróćmy teraz do wcześniejszego problemu. Język Python udostępnia mechanizm, który pozwala na wygodne przejście od reprezentacji „`fullname`" do „`name` i `surname`" bez konieczności dokonywania refaktoryzacji kodu – mowa o dekoratorze `@property`. Dekorator `@property` sprawia, że metoda zachowuje się niczym dynamiczny atrybut, czyli można z niej korzystać jak z atrybutu – bez podawania nawiasów okrągłych oznaczających normalnie wywołanie funkcji (zob. przykł. 2.8).

Listing 2.8. Definiowanie właściwości za pomocą dekoratora `@property` (na razie tylko funkcjonalność gettera)

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

    @property
    def fullname(self):
        return "{self.name} {self.surname}".format(self=self)

jane = Person("Jane", "Smith")
print(jane.fullname)  # nie potrzeba nawiasów!
```

Możemy również określić metodę realizującą funkcjonalność settera[13] za pomocą dekoratora `@property_name.setter` (gdzie `property_name` to nazwa symulowanego atrybutu), przy czym na-

[12] Te nazwy pochodzą oczywiście od angielskich czasowników: to get, to set.

[13] A także tzw. deleter, służący do usuwania atrybutu z obiektu.

zwa tak udekorowanej metody musi być identyczna z nazwą metody udekorowanej za pomocą `@property` (zob. przykł. 2.9)

Listing 2.9. Przykład właściwości z funkcjonalnością settera.

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

    @property
    def fullname(self):
        return f"{self.name} {self.surname}"

    @fullname.setter
    def fullname(self, value):
        name, surname = value.split(" ", 1)
        self.name = name
        self.surname = surname
```

Użycie klasy z przykładu 2.9 da następujący wynik:

```
>>> jane = Person("Jane", "Smith")
>>> jane.fullname
Jane Smith

>>> jane.fullname = "Jane Doe"
>>> jane.fullname
Jane Doe
>>> jane.name
Jane
>>> jane.surname
Doe
```

Mechanizm właściwości służy zatem zapewnieniu enkapsulacji: możemy zacząć od najprostszej implementacji atrybutu – od atrybutu publicznego, dostępnego bezpośrednio – a w razie potrzeby „opakować” go później w mechanizm kontroli dostępu (co nie wiąże się ze zmianą interfejsu klasy). To dlatego w języku Python obowiązuje konwencja domyślnego definiowania atrybutów jako publicznych. Atrybuty powinny być definiowane jako prywatne wyłącznie w sytuacji, gdy stanowią tzw. *wewnętrzną sprawę klasy* (ang. class internals), czyli gdy stanowią szczegóły implementacyjne klasy, do których użytkownik klasy nie powinien mieć dostępu (ani w ogóle wiedzieć o ich istnieniu. . .). Dzięki definiowaniu atrybutów jako publicznych przede wszystkim unikamy niepotrzebnej „eksplozji” interfejsu klasy – wprowadzania do jej interfejsu wielu w gruncie rzeczy niepotrzebnych metod (m.in. getterów i setterów, służących wyłącznie „przepychaniu” danych), które tylko zaciemniają rzeczywistą funkcjonalność klasy!

Ważne

Zgodnie z filozofią języka Python atrybuty powinny być domyślnie „publiczne”.

Dla zainteresowanych. . .

Wyczerpujące omówienie kwestii projektowych związanych z właściwościami: Python 3 Tutorial: Properties vs. Getters and Setters
Materiały źródłowe:

- Python Properties (by Scott Robinson)

2.5.4 Metody statyczne

Język Python udostępnia dekorator `@staticmethod`, który *teoretycznie* powinien być stosowany w następującej sytuacji: chcesz napisać funkcję pomocniczą powiązaną (semantycznie) z daną klasą, ale nie korzystającą *żadnych* jej składowych (ani powiązanych z instancją, ani z typem klasowym)[14].

Warto zadać sobie jednak pytanie – jaki miałby być cel takiego „dopinania" luźnej funkcji do danej klasy? Język Python udostępnia przecież inne metody grupowania funkcjonalności – moduły[15]! W praktyce nie zachodzi nigdy potrzeba tworzenia metod statycznych[16].

Zresztą sam twórca języka Python Guido van Rossum stwierdził w 2012 r., że:
„Wszyscy wiemy, jak ograniczone są możliwości metod statycznych. Te metody stanowią rodzaj «wypadku przy pracy»w czasie mojej pracy nad językiem Python 2.2, kiedy to starałem się opracować nowy styl pisania klas i ich deskryptorów. Chciałem zaimplementować mechanizm metod klasowych, lecz z początku nie miałem jeszcze dobrego wyczucia, jak powinny dokładnie funkcjonować, i przypadkiem *w pierwszej kolejności zaimplementowałem metody statyczne. Potem było już za późny, aby je usunąć i pozostawić wyłącznie metody klasowe.*

2.6 Metoda `__del__`

Każda klasa zawiera domyślnie zdefiniowaną metodę `__del__`, wykonywaną podczas niszczenia obiektu. Ponieważ zamiast mechanizmu RAII język Python posiada mechanizm tzw. odśmiecania[17], w praktyce nie zachodzi konieczność *własnoręcznego* definiowania destruktora. Oznacza to jednak również, że nie mamy gwarancji odnośnie tego, kiedy destruktor zostanie wykonany[18]. W związku z tym obsługa zasobów systemowych (np. połączeń z plikami, połączeń z bazą danych, połączeń sieciowych) powinna się odbywać poprzez tzw. **menedżery kontekstu** (ang. context managers), które gwarantują poprawne zwolnienie takich zasobów – także w przypadku wystąpienia błędów wykonania programu[19].

2.7 Nazwane krotki

Nazwana krotka (ang. *named tuple*) to obiekt przypominający zwykłą krotkę (w szczególności jest on niemutowalny), lecz udostępniający m.in.:

- mechanizm odwoływania się do poszczególnych swoich elementów za pomocą notacji właściwej dla atrybutów klasy (tj. z użyciem kropki) oraz
- przyjazną reprezentację tekstową przechowywanych danych w formacie listy złożonej z pozycji `nazwa=wartość`.

Sposób definiowania krotki (począwszy od wersji Python 3.6) przedstawia przykład 2.10 (w starszych wersjach języka Python służyła do tego funkcja `namedtuple()`).

Listing 2.10. Nazwana krotka reprezentująca pracownika, zawierająca dwa atrybuty.

```
# Python >=3.6
from typing import NamedTuple

class Employee(NamedTuple):
    name: str
    id: int

# Python <3.5
# Employee = collections.namedtuple('Employee', ['name', 'id'])
```

[14] To inna sytuacja niż np. w języku C++, gdzie metoda statyczna to metoda potencjalnie korzystająca z pól statycznych – czyli pól powiązanych z typem klasowym.

[15] zob. rozdz. 4 Programowanie modułowe

[16] Dokładniejsze omówienie takiego podejścia: When to use Static Methods in Python? Never (webucator)

[17] zob. rozdz. 7.1 Garbage collector

[18] W wersjach Pythona starszych niż 3.4, w przypadku zależności cyklicznych, destruktor mógł w ogóle nie być wykonany. To ograniczenie zostało usunięte w wersji 3.4 (zob. PEP 442 – Safe object finalization).

[19] Menedżery kontekstu zostały omówione w rozdziale 5.6 Instrukcja `with`.

Korzystanie z tak zdefiniowanej krotki nie różni się wiele od korzystania ze zwykłej klasy, której atrybuty są „tylko do odczytu". Z nazwanych krotek warto zatem korzystać w sytuacjach, gdy grupujemy dane nieulegające zmianom, do których jednocześnie jesteśmy w stanie przyporządkować opisowe „etykiety" (np. obiekt reprezentujący samochód danej marki i w danym kolorze). Oto przykład zachowania krotki z przykładu 2.10:

```
>>> employee = Employee('Guido', 3)
>>> employee.name
Guido
>>> employee
Employee(name='Guido', id=3)
```

Wykaz nazw pól nazwanej krotki przechowywany jest w specjalnym atrybucie `_fields`[20], w postaci „zwykłej" krotki, przykładowo:

```
>>> employee._fields
('name', 'id')
```

2.8 Metody magiczne

Każda klasa umożliwia zdefiniowanie pewnych metod, które później są wywoływane nie-wprost – stąd ich określenie „**metody magiczne**" (ang. magic methods). Przykładowo metoda `MyClazz.__init__(self)` wywoływana jest automatycznie przez interpreter języka Python w momencie tworzenia nowego obiektu klasy `MyClazz` za pomocą instrukcji `MyClazz()`; metoda ta jest wywoływana po przydzieleniu miejsca w pamięci na przechowanie takiego obiektu za pomocą innej metody magicznej, `MyClazz.__new__(self)`, której programista również nie wywołuje własnoręcznie i którą w praktyce w ogóle rzadko kiedy własnoręcznie (re)definiuje.

Innymi przykładami przydatnych metod magicznych są m.in.:

- `__str__(self)` – pozwala zwrócić reprezentację obiektu jako łańcuch znaków
  Wywołanie funkcji wbudowanej `str(obj)` jest tłumaczone na wywołanie powyższej metody, czyli na `obj.__str__()`.
- `__eq__(self, other)` – służy do emulacji operatora równości (zwraca wartość logiczną)
  Użycie operatora równości `obj1 == obj2` jest tłumaczone na wywołanie powyższej metody, czyli na `obj1.__eq__(obj2)`[21].

Dla zainteresowanych...

Oto przykład demonstrujący dlaczego chcąc sprawdzić, czy dany alias odnosi się do `None`, *nie należy* stosować operatora równości ==. Chodzi o to, że klasa może zdefiniować metodę `__eq__()` choćby w następujący sposób:

```
class Negator:
    def __eq__(self, other):
        return not other
```

przez co sprawdzenie *równości* danej instancji `Negator` z wartością `None` nie daje intuicyjnych wyników – w przeciwieństwie do sprawdzenia *tożsamości* z `None`:

```
thing = Negator()
print(thing == None)   # True <= wywołanie thing.__eq__(None)
print(thing is None)   # False
```

Ponieważ obiekt `None` jest unikalny w skali programu (tj. istnieje tylko jeden taki egzemplarz), wszystkie aliasy na `None` wskazują w istocie na ten sam obiekt (zatem operator **`is`**, którego nie można przedefiniować, zwróci poprawną wartość).

Więcej informacji o metodach magicznych (wraz z licznymi przykładami) znajdziesz na stronie A Guide to Python's Magic Methods (Rafe Kettler).

2.9 Testy jednostkowe w języku Python

[20] Atrybut `_fields` stanowi część API kolekcji `namedtuple`.

[21] Zatem istnieje m.in. możliwość zdefiniowania nieintuicyjnej wersji operatora przypisania.

Ważne

Przed przystąpieniem do czytania tego rozdziału zapoznaj się z ogólnymi informacjami odnośnie testowania oprogramowania na wiki: Testowanie oprogramowania oraz Testy jednostkowe.

Python posiada dobre wbudowane wsparcie dla testów jednostkowych – w postaci modułu `unittest` – zatem nie ma konieczności korzystania z zewnętrznych frameworków[22].

Testy zwykle organizuje się w następujący sposób: w głównym katalogu projektu tworzy się osobny katalog `test/`, w którym umieszcza się moduły (pliki `.py`) z testami – przy czym każdemu modułowi z logiką programu powinien odpowiadać osobny moduł z testami. Przykładowo:

```
my_projest/
    ...
    module1.py
    module2.py
    test/
        ...
        test_module1.py
        test_module2.py
```

Przyjmując, że w programie istnieje moduł `mymath.py` definujący funkcję `factorial(n: int) -> int` zwracającą silnię n, moduł `test_mymath.py` mógłby zawierać poniższe scenariusze testowe zgrupowane w klasę `TestFactorial`:

```
import unittest                 # Zaimportuj modułu testów jednostkowych.
from mymath import factorial    # Zaimportuj funkcjonalności do przetestowania.

# zbiór testów (tj. klasa dziedzicząca po `unittest.TestCase`)
class TestFactorial(unittest.TestCase):

    # Każdy przypadek testowy to osobna metoda klasy-zbioru testów.

    def test_n_zero(self):
      # Dostęp do makr testowych - poprzez obiekt `self` (gdyż są one
        #   zdefiniowane w klasie `unittest.TestCase`).
        # unittest.TestCase.assertEqual(val_act, val_exp):
        #   * val_act -- wartość faktyczna,
        #   * val_exp -- wartość oczekiwana
        #   (kolejność ma znaczenie pod kątem komunikatów diagnostycznych!)
        self.assertEqual(factorial(0), 1)

    def test_n_positive(self):
        self.assertEqual(factorial(1), 1)
        self.assertEqual(factorial(3), 6)
```

Dany zbiór testów – czyli klasa dziedzicząca po klasie `unittest.TestCase` – pełni rolę kontenera na testy grupuje przypadki testy odnoszące się do danej funkcjonalności (np. funkcji albo klasy), przy czym każdy przypadek testowy jest zaimplementowany jako osobna metoda takiej klasy-zbioru.

Wewnątrz testu możesz weryfikować poprawność testowanej funkcjonalności za pomocą tzw. **metod asercji** (ang. assertion methods), czyli specjalnych metod z klasy `unittest.TestCase`, które oferują bogatą diagnostykę w przypadku niespełnienia warunku. Co może być przedmiotem warunku testowego? Przykładowo:

- relacja (nie)równości między dwiema wartościami
- tożsamość dwóch wartości
- zawieranie się wartości (np. czy kontener zawiera pewien element)
- czy został rzucony wyjątek danego typu

[22] Przykładowo, języki C i C++ nie posiadają takiej funkcjonalności – wymagają one korzystania z zewnętrznych, niestandardowych narzędzi.

Jeśli dowolna z asercji nie będzie spełniona, cały test „nie przechodzi” – dalsze wykonanie przypadku testowego jest przerywane[23].

Zbiór testów może definiować pewne specjalne metody o ustalonych nazwach, które są wywoływane „automatycznie” w odpowiednich momentach, przykładowo:

- `setUp(self)` – metoda wywoływana bezpośrednio *przed* wywołaniem każdej z metod-przypadków testowych
- `tearDown(self)` – metoda wywoływana bezpośrednio *po* wywołaniu każdej z metod-przypadków testowych

Jeśli wszystkie Twoje testy wymagają pewnej jednakowej, specjalnej konfiguracji wstępnej (np. utworzenia kilku obiektów o konkretnym stanie), możesz umieścić te instrukcje w implementacji metody `setUp()`.

Dla zainteresowanych...

Materiały źródłowe[24]:

- `unittest` – Unit testing framework (PyDocs)
- D. Phillips. *Python 3 Object Oriented Programming* (2010), rozdz. 11
- T. Ziade. *Expert Python Programming* (2008), rozdz. 11
- M. Alchin. *Pro Python* (2010)
- Python 3 Tutorial: Python Tests

[23] Moduł `unittest` nie udostępnia możliwości analogicznych do frameworku Google Testing Framework, w którym można korzystać z makr `EXPECT_XXX`, których niespełnienie oznaczało cały test jako „nie przechodzi”, ale jednocześnie pozwalały na dalsze jego wykonanie (i potencjalne zebranie dalszych przydatnych informacji diagnostycznych).

[24] Informacje zawarte w trzech podanych książkach w dużym stopniu się pokrywają.

Rozdział 3

Klasy: Rozszerzanie funkcjonalności

W niniejszym rozdziale poznasz zasady i dostępne w języku Python mechanizmy służące realizacji koncepcji rozszerzania funkcjonalności w kontekście programowania zorientowanego na obiekty.

Dla zainteresowanych...

Materiały źródłowe:

- Object-Oriented Programming in Python: Object-oriented programming
- Python 3 Tutorial (Python Course)
- Python 3 OOP Part 3 – Delegation: composition and inheritance (by Leonardo Giordani)
- Object-Oriented Programming in Python 1 (kod źródłowy miejscami przestarzały!))
- Python Design Patterns: For Sleek And Fashionable Code (by Andrei Boyanov)

3.1 Relacje między klasami

W języku Python istnieją dwa główne typy relacji między klasami: kompozycja i dziedziczenie.

3.1.1 Kompozycja

Relacja kompozycji to najprostszy rodzaj powiązania między dwiema klasami – polega po prostu na dodaniu do klasy `B` składowej będącej instancją klasy `A`, przy czym instancja `B` posiada instancję `A` „na wyłączność" (zob. przykł. 3.1). Oznacza to, że:

- instancja `A` utworzona podczas inicjalizacji instancji `B` nie może istnieć w oderwaniu od instancji `B`, oraz że
- klasa `B` może korzystać wyłącznie z publicznego interfejsu udostępnianego przez klasę `A`.

Listing 3.1. Przykład relacji kompozycji.

```
class A:
    pass

class B:
    def __init__(self):
        self.a = A()  # relacja kompozycji
                      #   (instancja `A` jako składowa instancji `B`)
```

3.1.2 Dziedziczenie

Klasy powiązane relacją dziedziczenia tworzą hierarchię – klasa macierzysta definiuje funkcjonalność wspólną dla obu klas, natomiast klasa pochodna rozszerza tą funkcjonalność o właściwe sobie składowe (zob. przykł. 3.2).

Listing 3.2. Przykład relacji dziedziczenia. Użycie funkcji `super()` oraz parametrów `*args` `**kwargs` pozwala na wygodne wywołanie metody inicjalizacyjnej klasy macierzystej, bez konieczności znajomości jej listy parametrów; gdy dana zdefiniowana przez Ciebie klasa dziedziczy wyłącznie po klasie `object`, w znakomitej większości przypadków możesz pominąć wywołanie funkcji `super()`[1].

```
class BaseClass:
    pass

class DerivedClass(BaseClass):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)  # dobra praktyka: inicjalizacja
                                           #   atrybutów związanych z klasą
                                           #   macierzystą (BaseClass)
```

3.1.3 Podpowiedzi typu dla klas w hierarchii dziedziczenia

Czasem chcemy wyrazić za pomocą podpowiedzi typów fakt, że dany obiekt może być obiektem pewnej klasy `T` *albo jednej z jej klas pochodnych*. W tym celu należy zdefiniować własny typ podpowiedzi za pomocą klasy `TypeVar` z modułu `typing` – metoda inicjalizacyjna klasy `TypeVar` oprócz argumentu pozycyjnego, którego wartość musi być taka sama jak nazwa definiowanej właśnie podpowiedzi typu, przyjmuje m.in. argument słownikowy `bound` określający wspomniane „dolne ograniczenie” (zob. przykł. 3.3).

Listing 3.3. Przykład definiowania podpowiedzi typu odnoszącej się do klas w hierarchii dziedziczenia

```
from typing import TypeVar

class BaseClass:
    pass

class DerivedClass(BaseClass):
    pass

# Pierwszy argument metody `TypeVar.__init__()` to łańcuch znaków, który musi
#   być zgodny z nazwą definiowanej podpowiedzi typu (czyli nazwą po lewej
#   stronie operatora przypisania).
# Argument słownikowy `bound=T` oznacza, że podpowiedź typu oznacza "typ T lub
#   dowolny z jego typów pochodnych".
BaseOrDerivedT = TypeVar('BaseOrDerivedT', bound=BaseClass)

obj1: BaseOrDerivedT = BaseClass()      # OK: typ `BaseClass`
obj2: BaseOrDerivedT = DerivedClass()   # OK: typ pochodny dla `BaseClass`
```

Nadpisywanie metod

Mechanizm nadpisywania metod sprawia, że możemy w klasie potomnej zmienić implementację metody zdefiniowanej uprzednio w klasie macierzystej. Natomiast wciąż mamy możliwość odwołania się do metody zdefiniowanej w klasie macierzystej z użyciem notacji `BaseClass.method_name(derived_obj)` (zob. przykł. 3.4).

Z mechanizmu nadpisywania korzystaliśmy m.in. podczas definiowania metody `__init__()`, jak w przykładzie 3.2 (przy czym do klasy macierzystej odnosiliśmy się za pomocą funkcji `super()`) – ten przypadek dotyczy każdej klasy, gdyż każda klasa niejawnie dziedziczy po klasie `object`[2].

[1] Funkcja `super()` została dokładniej opisana w rozdziałach 3.2.1 Problem diamentu #2: Wielokrotne wywoływanie metod z klasy macierzystej i 3.2.1 Dziedziczenie wielokrotne a `__init__`. Szczegółowe omówienie tego, kiedy i jak wywoływać metodę `object.__init__()`, znajdziesz w dyskusji When inheriting directly from ‘object‘, should I call super().__init__()?

[2] W języku Python 2 należało jawnie zdefiniować dziedziczenie po klasie `object` (zob. Stack Overflow).

Listing 3.4. Przykład nadpisania metody

```
class A:
    def m(self):
        pass

class B(A):
    # nadpisanie metody `m()` z klasy A
    def m(self):
        pass

obj_b = B()
obj_b.m()    # wywołanie metody `m()` z klasy B  (dla obiektu `obj_b`)
A.m(obj_b)   # wywołanie metody `m()` z klasy A  (dla obiektu `obj_b`)
```

Informacja

*Niektórzy stosują błędny termin „*overwriting*” zamiast poprawnego „*overriding*” – choć w obu przypadkach chodzi o ten sam mechanizm (nadpisywania metod).*

3.2 Więcej o dziedziczeniu...

Zagadnienia omówione w tym rozdziale stanowią podstawę stosowaną w zasadzie w każdym średniej wielkości projekcie.

3.2.1 Dziedziczenie wielokrotne

Język Python umożliwia dziedziczenie wielokrotne (zob. przykł. 3.5).

Listing 3.5. Przykład relacji dziedziczenia wielokrotnego.

```
class A:
    pass

class B:
    pass

# klasa C dziedziczy zarówno po A, jak i po B
class C(A,B):
    pass
```

Przekonanie, że dziedziczenie wielokrotne samo w sobie jest „niebezpieczne” bądź „złe”, można potraktować w przypadku języka Python jako uprzedzenie – język Python posiada bowiem dobrze zaprojektowany mechanizm rozwiązywania typowych problemów pojawiających się w przypadku wielokrotnego dziedziczenia. Poniżej opisano dwa tzw. problemy diamentu oraz sposoby ich rozwiązania w Pythonie.

Poniższe przykłady zaczerpnięto z Python 3 Tutorial: Multiple Inheritance

Problem diamentu #1: „Przypadkowa” kolejność wywołań metod

W przykładzie 3.6 klasa `C` nadpisuje metodę `m()` z klasy `A`. W przypadku wywołania metody `m` dla instancji klasy `D` o nazwie `x` (`x.m()`) dostaniemy wynik:

```
m of A called
```

natomiast gdybyśmy zamienili kolejność klas macierzystych w nagłówku klasy `D`:

```
class D(C,B):
    pass
```

wynikiem wywołania `x.m()` będzie

```
m of C called
```

Listing 3.6. Przykład relacji dziedziczenia, w których występuje tzw. problem diamentu.

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    pass

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    pass
```

Jest to efekt działania wspomnianego wcześniej mechanizmu MRO, za pomocą którego Python „rozwiązuje" problem wielokrotnego dziedziczenia – tj. w jaki sposób przeszukiwać klasy macierzyste w poszukiwaniu składowej[3].

Aby jawnie określić, która metoda ma zostać wywołana, możemy użyć następującej notacji:

```
x = D()
B.m(x)   # wywołanie metody `m` z klasy `B`
C.m(x)   # wywołanie metody `m` z klasy `C`
```

Problem diamentu #2: Wielokrotne wywoływanie metod z klasy macierzystej

Twórca kodu z przykładu 3.7 chciał uzyskać następujący efekt: metoda `m()` z klasy `D` powinna wywoływać metody `m()` z *wszystkich* klas macierzystych.

Listing 3.7. Przykład relacji dziedziczenia, w których występuje tzw. problem diamentu.

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")

class C(A):
    def m(self):
        print("m of C called")

class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
        A.m(self)
```

W tym przypadku implementacja wygląda rozsądnie. Jednak problem pojawiłby się w sytuacji, gdybyśmy w każdej z klas potomnych chcieli nadpisać metodę `m()` (z klasy `A`) w ten sposób, aby wywoływała ona również swoją wersję ze wszystkich bezpośrednich klas macierzystych – sytuację taką ilustruje przykład 3.8.

[3] Mechanizm MRO został omówiony w rozdziale 7.4 Method Resolution Order.

Listing 3.8. Problem wielokrotnego wywoływania metody z klasy macierzystej.

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")
        A.m(self)

class C(A):
    def m(self):
        print("m of C called")
        A.m(self)

class D(B,C):
    def m(self):
        print("m of D called")
        B.m(self)
        C.m(self)
        A.m(self)
```

W tym drugim przypadku wywołanie metody `m()` dla instancji klasy `D` spowoduje aż trzykrotne wywołanie metody `m()` z klasy `A` (raz pośrednio poprzez metodę `B.m()`, raz pośrednio poprzez metodę `C.m()` oraz raz bezpośrednio w metodzie `D.m()`):

```
>>> D().m()
m of D called
m of B called
m of A called
m of C called
m of A called
m of A called
```

Optymalne rozwiązanie polega na stosowaniu wbudowanej funkcji `super()`[4]. Jej użycie, zaprezentowane w przykładzie 3.9, zapewnia wywołanie wszystkich metod `m()`, we właściwej kolejności, zdefiniowanej przez reguły MRO, a także wywołanie metody ze wspólnej klasy macierzystej tylko raz; w tym przypadku:

```
>>> D().m()
m of D called
m of B called
m of C called
m of A called
```

Zwróć uwagę, że kolejność wywołań metody `m()` z klasy `B` i `C` odpowiada kolejności tych klas na liście klas macierzystych w definicji klasy `D`.

Dziedziczenie wielokrotne a `__init__`

W każdej z nadpisanych metod `__init__()` korzystamy z tych parametrów metody, które są właściwe dla zawierającej ją klasy, a resztę przekazujemy do metody `__init__()` klasy macierzystej. Mechanizm MRO sprawia, że powinniśmy przekazać dalej nie tylko argumenty oczekiwane przez metodę bezpośredniego rodzica, ale *wszystkie* niewykorzystane argumenty – gdyż nie wiemy, metody której klasy (a więc o jakiej dokładnie liście parametrów) zostaną wywołane w dalszej kolejności.

Aby zapewnić takie przekazywanie, powszechną konwencją jest dodawanie parametrów właściwych dla danej podklasy na początku listy parametrów, a zdefiniowanie wszystkich pozostałych za pomocą argu-

[4]Składnia funkcji `super()` różni się między wersjami 2 i 3 języka Python.

Listing 3.9. Rozwiązanie problemu diamentu z użyciem funkcji `super()`.

```
class A:
    def m(self):
        print("m of A called")

class B(A):
    def m(self):
        print("m of B called")
        super().m()

class C(A):
    def m(self):
        print("m of C called")
        super().m()

class D(B,C):
    def m(self):
        print("m of D called")
        super().m()
```

mentów pozycyjnych (`*args`) oraz słownikowych (`**kwargs`) – dzięki temu klasa potomna nie musi znać żadnych szczegółów związanych z parametrami metody `__init__()` klasy macierzystej (zob. przykł. 3.10).

Listing 3.10. Użycie argumentów pozycyjnych (`*args`) oraz słownikowych (`**kwargs`) w metodach `__init__()`.

```
class B1:
    def __init__(self, p, *args, **kwargs):
        print("p = ", p)
        super().__init__(*args, **kwargs)

class B2:
    def __init__(self, q, *args, **kwargs):
        print("q = ", q)
        super().__init__(*args, **kwargs)

class D1(B1, B2):
    def __init__(self, r, *args, **kwargs):
        print("r = ", r)
        super().__init__(*args, **kwargs)

-------------------------------------------------------------------------
>>> D1(1, p=3, q=2)
r =  1
p =  3
q =  2
```

Takie rozwiązanie gwarantuje również, że dodając nowy parametr do metody `__init__()` klasy nadrzędnej będziemy musieli zmienić wyłącznie wyrażenia tworzące obiekty tej klasy oraz kod metody `__init__()` w każdej z jej *bezpośrednich* klas potomnych – nie będziemy natomiast musieli zmieniać definicji metody `__init__()` we *wszystkich* klasach potomnych.

Ważne

Co do zasady, w metodzie `__init__()` należy *zawsze* wywoływać `super().__init__()` (nawet, gdy klasa dziedziczy tylko po `object`), gdyż tylko wtedy mamy gwarancję, że wszystkie fragmenty klasy zostaną poprawnie zainicjalizowane[5].

Aby nie musieć przejmować się tym, jakie konkretnie parametry posiadają metody inicjalizacyjne klas macierzystych (jednej lub wielu), zwykle warto przekazać w wywołaniu `super().__init__()` *wszystkie* otrzymane argumenty – za pomocą `super().__init__(*args, **kwargs)`. W ten sposób wiele z ewentualnych zmian w nagłówku metodach inicjalizacyjnych klas macierzystych (w szczególności: dodanie nowych parametrów) nie zaburzy funkcjonowania Twojego kodu.

Dla zainteresowanych...

Warto mieć świadomość problemów wynikających z nieumiejętnego stosowania funkcji `super()`:

- Python's Super is nifty, but you can't use it (by James Knight)
- The wonders of cooperative inheritance, or using super in Python 3 (by Michele Simionato)

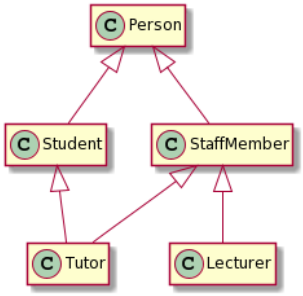
Materiały źródłowe:

- Python's super() considered super! (by Raymond Hettinger)

3.2.2 Mix-iny

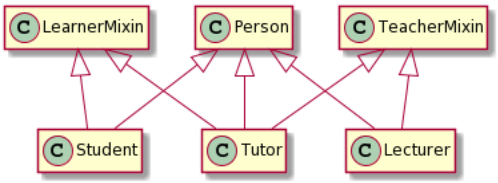
Rozważ następujący problem: Tworzysz system do zarządzania uczelnią – w szczególności jej „zasobami ludzkimi", czyli studentami i prowadzącymi zajęcia. Zarówno studenci, jak i prowadzący, to osoby – zatem byty posiadające pewne dane osobowe. Co więcej, student może być jednocześnie prowadzącym (np. student studiów doktoranckich) – taką rolę określimy jako „tutor".

Aby zamodelować taki system, zamiast tworzyć mało przejrzystą pionową hierarchię dziedziczenia (Rys. 3.1), lepiej podzielić funkcjonalność na mniejsze „bloczki" – tzw. mix-iny – dodawane w razie potrzeby do klas za pomocą (wielokrotnego) dziedziczenia. **Mix-in** to klasa, która w założeniu nie będzie używana samodzielnie – zamiast tego rozszerza ona funkcjonalność istniejącej klasy poprzez dziedziczenie wielokrotne.



Rysunek 3.1. Pionowa hierarchia dziedziczenia (niezalecane).

W przytoczonym problemie taką wydzieloną funkcjonalnością będzie „uczący się" (klasa `LearnerMixin`) oraz „nauczający" (klasa `TeacherMixin`) – zob. przykład 3.11 – za pomocą których możemy efektywnie wyrazić koncept studenta, prowadzącego zajęcia, oraz tutora (studenta-prowadzącego). Rysunek 3.2 przedstawia hierarchię klas po zastosowaniu mix-inów.



Rysunek 3.2. Płaska hierarchia dziedziczenia z użyciem mix-inów (zalecane).

Zwróć uwagę, że nowa hierarchia klas jest bardziej płaska oraz że nie występuje w niej tzw. problem diamentu[6] (jak w przypadku klasy `Tutor` z rysunku 3.1).

[5] zob. też rozdz. 3.2.1 Dziedziczenie wielokrotne a `__init__`

[6] Problem diamentu został omówiony w rozdziale 3.2.1 Dziedziczenie wielokrotne.

Listing 3.11. Mix-iny `LearnerMixin` oraz `TeacherMixin`.

```
class Person:
    def __init__(self, name, surname, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.name = name
        self.surname = surname


class LearnerMixin:
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.classes = []

    def enrol(self, course):
        self.classes.append(course)


class TeacherMixin:
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.courses_taught = []

    def assign_teaching(self, course):
        self.courses_taught.append(course)
```

Nie wszystkie mix-iny definiują nowe atrybuty, część z nich bazuje na atrybutach już istniejących. Podobnie część mix-inów korzysta z metod udostępnianych przez klasę, która dziedziczy dany mix-in.

Powiedzmy, że tworzymy model sieci komputerowej i potrzebujemy mechanizmu umożliwiającego serializację i deserializację[7] pythonowych obiektów reprezentujących elementy wspomnianej sieci z użyciem formatu JSON. Ponieważ taka generyczna funkcjonalność będzie nam potrzebna w przypadku kilku różnych typów obiektów (np. węzłów i połączeń), zatem rozsądnie będzie stworzyć odpowiedni mix-in (zob. przykł. 3.12).

Listing 3.12. Przykład mix-inu do serializacji i deserializacji danych w formacie JSON.

```
import json

class JsonMixin:
    @classmethod
    def from_json(cls, data):
        kwargs = json.loads(data)
        return cls(**kwargs)

    def to_json(self):
        return json.dumps(self.to_dict())  # skorzystaj z metody
                                           # klasy "domieszkowanej"
```

Jak widzisz, pisanie mix-inów jest proste, gdyż język Python udostępnia mechanizm sprawdzania istnienia składowych obiektu niezależnie od jego typu[8] oraz jest dynamicznie typowany.

Po utworzeniu innego mix-inu, `ToDictMixin`, udostępniającego metodę `to_dict()`, możemy efektywnie tworzyć klasy wymagające (de)serializacji – dziedzicząc po obu mix-inach (zob. przykł. 3.13).

Oto wskazówki, które pozwolą Ci zorientować się, czy mix-iny mają zastosowanie w danym przypadku:

[7] **Serializacja** (ang. serialization) to proces tłumaczenia struktur danych lub stanu obiektów do takiego formatu, który będzie mógł być przechowywany (np. jako plik na dysku) lub przesyłany (np. poprzez łącze sieciowe), w celu późniejszej rekonstrukcji – czyli **deserializacji** (ang. deserialization).

[8] zob. rozdz. 3.2.2 `getattr()`, `setattr()` i `hasattr()`

Listing 3.13. Tworzenie klas korzystających z wielu mix-inów.

```
class Node(ToDictMixin, JsonMixin):
    # ...

class Link(ToDictMixin, JsonMixin):
    # ...
```

- Czy wydzielona funkcjonalność zostanie *od razu* zastosowana w różnych niepowiązanych ze sobą klasach?
  - Jeśli tak, to *być może* warto zastosować mix-in.
  - Jeśli wydzielona funkcjonalność miałaby być użyta tylko w jednej klasie – nie wydzielaj jej.
- Czy potrafisz streścić wydzieloną funkcjonalność za pomocą przymiotnika (a nie rzeczownika lub czasownika), np. *Movable*? – Jeśli nie, to zapewne nie chodzi Ci o utworzenie mix-ina.
- Czy wydzielona funkcjonalność stanowi udoskonalenie lub specjalizację istniejącego typu? – Jeśli tak, to *być może* lepiej zastosować „zwykłe" dziedziczenie.
- Czy masz problem ze skutecznym wydzieleniem mix-ina? – Jeśli tak, to użyj kompozycji. Relacja kompozycji jest zawsze bezpieczniejsza, gdyż łatwiej ją utworzyć i łatwiej z niej korzystać. Lepiej użyć kompozycji niż stworzyć słabo przemyślany mix-in.

Unikaj dołączania zbyt wielu mix-inów do klasy, gdyż w ten sposób tworzysz coś w rodzaju boskiej klasy, co jest sprzeczne z zasadą pojedynczej odpowiedzialności. Co więcej, takie podejście zwiększa zależności w systemie i utrudnia wprowadzanie zmian (choćby dlatego, że całe API każdej z klas macierzystych staje się interfejsem klasy pochodnej) – w przypadku kompozycji ryzyko wystąpienia takich problemów jest znacznie mniejsze.

Ważne

Unikaj tworzenia rozbudowanych hierarchii klas (w szczególności korzystających z dziedziczenia wielokrotnego), jeśli mix-in'y mogą zapewnić identyczną funkcjonalność.

Pamiętaj, że w wielu przypadkach stosowanie mix-inów jest rezultatem złego zaprojektowania programu – często istnieją lepsze alternatywy dla mix-inów. Przykładami mix-inów z biblioteki standardowej są klasy `ForkingMixIn` oraz `ThreadingMixIn` w module `socketserver`[9].

Dla zainteresowanych...

O tym, kiedy lepiej zrezygnować z mix-inów na rzecz innych rozwiązań przeczytasz w poniższych materiałach:

- Mixins considered harmful/1 (by Michele Simionato)
- Mixins considered harmful/2 (by Michele Simionato)

`getattr()`, `setattr()` i `hasattr()`

Podczas pisania mix-inów przydatne są funkcje wbudowane `hasattr()`, `getattr()` oraz `setattr()`[10]:

- Funkcja `hasattr()` pozwala sprawdzić, czy dany obiekt posiada atrybut o zadanej nazwie.
- Funkcja `getattr()` pozwala dla danego obiektu odczytać wartość jego atrybutu o zadanej nazwie oraz zapewnia (opcjonalny) mechanizm obsługi sytuacji, gdy obiekt nie posiada atrybutu o takiej nazwie.
- Funkcja `setattr()` pozwala dla danego obiektu zmienić wartość jego atrybutu o zadanej nazwie – o ile obiekt dopuszcza taką możliwość – oraz zapewnia (opcjonalny) mechanizm obsługi sytuacji, gdy obiekt nie posiada atrybutu o takiej nazwie.

[9] zob. `socketserver` – A framework for network servers
[10] zob. `getattr`, `setattr` and `hasattr` (OOPinP)

3.2.3 Klasy abstrakcyjne i interfejsy

Język Python nie wspiera bezpośrednio tworzenia klas abstrakcyjnych[11] – możemy jednak w tym celu skorzystać z funkcjonalności udostępnianych przez standardowy moduł `abc`. W tym celu definiowana klasa abstrakcyjna powinna dziedziczyć po standardowej klasie `ABC`[12] oraz posiadać choć jedna metodę udekorowaną jako `@abstractmethod`.

Przykład 3.14 zawiera definicję klasy abstrakcyjnej `Shape` będącej szablonem dla kształtów (dwuwymiarowych figur). Próba utworzenia obiektu takiej klasy skutkuje błędem:

```
>>> Shape()
Traceback (most recent call last):
  File "...", line 10, in <module>
    Shape()
TypeError: Can't instantiate abstract class Shape with abstract methods area
```

Listing 3.14. Abstrakcyjna klasa `Shape`.

```
from abc import ABC, abstractmethod

class Shape(ABC):

    @abstractmethod
    def area(self):
        pass
```

Przykład 3.15 zawiera definicję klasy `Square` dziedziczącej po klasie `Shape` i implementującej wszystkie metody abstrakcyjne, czyli **klasy konkretnej** (ang. concrete class). Próba utworzenia obiektu klasy `Square` nie powoduje wystąpienia błędu:

```
>>> Square(2)
>>>
```

Listing 3.15. Konkretna klasa `Square`.

```
class Square(Shape):
    def __init__(self, width):
        self.width = width

    def area(self):
        return self.width ** 2
```

Aby uniknąć błędów w sytuacji, gdy interfejs klasy macierzystej uległ zmianie (np. w wyniku refaktoryzacji zmieniła się nazwa metody abstrakcyjnej), a zapomniano o odpowiedniej zmianie klas potomnych, można do metod „nadpisywanych" w klasach potomnych dodać odpowiedni własnoręcznie zdefiniowany dekorator[13].

3.2.4 Zastępowanie dziedziczenia kompozycją

Ponieważ język Python jest językiem typowanym dynamicznie (a nie statycznie, jak np. język C++), polimorfizm można osiągnąć bez konieczności stosowania dziedziczenia – wystarczy, że dany obiekt posiada wszystkie niezbędne atrybuty i/lub metody[14].

Choć język Python zapewnia dobre wsparcie dla dziedziczenia wielokrotnego, dobrą praktyką jest mimo wszystko w miarę możliwości unikanie go – aby nie tworzyć niepotrzebnych powiązań między klasami. Zamiast uciekania się do dziedziczenia, często możemy osiągnąć pożądaną funkcjonalność za pomocą kompozycji.

[11] W języku C++ taki mechanizm jest wbudowany w jądro języka.
[12] ABC to akronim od ang. Abstract Base Class – abstrakcyjna klasa macierzysta.
[13] Takie rozwiązanie zostało przedstawione w dyskusji In Python, how do I indicate I'm overriding a method? (Stack Overflow).
[14] zob. rozdz. 3.3 Duck Typing

Ważne

W języku Python kompozycja jest bardziej elegancka i naturalna niż dziedziczenie.

Przykład 3.16 zawiera definicje klas reprezentujących zwykłe drzwi (klasa `Door`) oraz drzwi antywłamaniowe (klasa `SecurityDoor`). W tym (nieoptymalnym) przykładzie kompozycji musieliśmy napisać metodę delegującą `close()`, natomiast odwołania do atrybutów `number` i `status` są możliwe wyłącznie z użyciem składni

```
security_door = SecurityDoor(1, 'open')
print(security_door.door.number)
print(security_door.door.status)
```

Listing 3.16. Relacja kompozycji (zaimplementowana nieoptymalnie)

```
class Door:
    def __init__(self, number, status):
        self.number = number
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

class SecurityDoor:
    def __init__(self, number, status):
        self.door = Door(number, status)
        self.locked = True

    def open(self):
        if self.locked:
            return
        self.door.open()

    def close(self):
        self.door.close()
```

Możemy uniknąć wspomnianych problemów nadpisując w klasie `SecurityDoor` metodę magiczną `__getattr__()`, wywoływaną gdy żądana składowa nie została znaleziona w obiekcie (zob. przykł. 3.17).

3.3 Duck Typing

Mechanizm „duck typing” służy do realizacji zasady:

Zasada

Program to an interface, not an implementation.

W myśl tej zasady nie powinna interesować nas natura obiektu (*czym tak naprawdę jest dany obiekt?*), lecz jedynie to, czy obiekt taki posiada pożądaną przez nas funkcjonalność.

Przykład 3.18 demonstruje użycie duck typingu w języku Python[15].

[15] W języku C++ przykładem realizacji idei *duck typingu* są szablony.

Listing 3.17. Relacja kompozycji (zaimplementowana optymalnie). Nadpisanie metody specjalnej `__getattr__()` ułatwia dostęp do atrybutów.

```
class SecurityDoor:
    def __init__(self, number, status):
        self.door = Door(number, status)
        self.locked = True

    def open(self):
        if self.locked:
            return
        self.door.open()

    def __getattr__(self, attr):
        return getattr(self.door, attr)

# ---

>>> security_door = SecurityDoor(1, 'open')
>>> print(security_door.number)
1
>>> print(security_door.status)
'open'
>>> security_door.close()
>>>
```

Listing 3.18. Przykład użycia duck typingu.

```
class A:
    def foo(self):
        print("A.foo")

class B:
    def foo(self):
        print("B.foo")

class C:
    pass

def bar(obj):
    obj.foo()   # Funkcja `bar()` nie ma ŻADNEJ gwarancji, że
                #   obiekt `obj` posiada metodę `foo()`...

# -----

>>> bar(A())
A.foo
>>> bar(B())
B.foo
>>> bar(C())
Traceback (most recent call last):
  File "...", line 21, in <module>
    bar(C())
  File "...", line 16, in bar
    obj.foo()
AttributeError: 'C' object has no attribute 'foo'
```

Rozdział 4

Programowanie modułowe

Już na wiele lat przed pojawieniem się komputerów inżynierowie zauważyli, że łatwiej tworzyć złożone systemy z mniejszych, wyspecjalizowanych, funkcjonalnych elementów, tzw. *modułów*, które można rozwijać niezależnie od siebie i z których zwykle można korzystać w różnych systemach.
Wyróżnikami systemu modułowego (ze względu na wspomniane cechy) są:

- podział funkcjonalności na niewielkie, skalowalne, mające zastosowanie w różnych sytuacjach moduły, oraz
- ściśle przestrzegana zasada tworzenia dobrze zaprojektowanych interfejsów – w oparciu o normy przemysłowe.

W idealnym systemie modułowym poszczególne moduły są od siebie w pełni niezależne i dopiero pewna „wyższa warstwa" łączy je ze sobą – przykładowo w przypadku podzespołów komputera rolę takiej „wyższej warstwy" pełni płyta główna (zapewnia komunikację między poszczególnymi podzespołami).

Programowanie modułowe (ang. modular programming) to technika projektowania oprogramowania oparta na wspomnianej zasadzie tworzenia systemu modułowego.

4.1 Czym jest moduł w języku Python?

W języku Python moduł służy do grupowania elementów realizujących powiązaną (np. tematycznie) funkcjonalność, co pozwala uniknąć „zaśmiecania" globalnej przestrzeni nazw (elementami tymi mogą być dowolne obiekty – klasy, funkcje, atrybuty globalne itd.)[1].

Tworzenie modułów nie wymaga korzystania z żadnej specjalnej składni. Każdy plik posiadający rozszerzenie `.py` i zawierający poprawny składniowo kod napisany w języku Python traktowany jest jako moduł.

4.2 Importowanie modułów

Aby skorzystać z funkcjonalności danego modułu, należy go najpierw zaimportować. W najprostszy sposób można to zrobić następująco:

```
import module_name  # zaimportuj moduł `module_name`
```

przykładowo

```
import typing  # zaimportuj moduł `typing`
```

przy czym co do zasady instrukcje **`import`** powinny znajdować się na samym początku pliku.

Rozważmy kwestie związane z importowaniem modułów na przykładzie modułu `math` – należy on do biblioteki standardowej i udostępnia różne stałe oraz funkcje matematyczne, np. π (`math.pi`) oraz sinus (`math.sin()`). Po zaimportowaniu tego modułu z użyciem instrukcji

```
import math
```

możemy korzystać z jego funkcjonalności za pomocą nazwy *kwalifikowanej*, czyli poprzedzając nazwę funkcjonalności nazwą modułu:

[1] Moduł w języku Python stanowi odpowiednik połączenia pliku nagłówkowego i przestrzeni nazw w języku C++.

```
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
```

Gdy potrzebujemy skorzystać tylko z kilku wybranych funkcjonalności danego modułu, a jednocześnie naszym zdaniem nazwy takich funkcjonalności nie wejdą w konflikt z innymi istniejącymi w programie nazwami (tj. uważamy, że nazwy funkcjonalności z modułu są unikalne w skali programu), możemy zaimportować te funkcjonalności bezpośrednio: za pomocą instrukcji

```
from module_name import entity1[, entity2, ...]
```

przykładowo

```
from math import pi, sin  # po wykonaniu tej instrukcji tylko
                          # `pi` i `sin` będą dostępne
```

a następnie korzystać z zaimportowanych funkcjonalności już bez kwalifikacji:

```
>>> pi
3.141592653589793
>>> sin(pi/2)
1.0
```

Istnieje również możliwość bezpośredniego zaimportowania *wszystkich* obiektów danego modułu, za pomocą gwiazdki (`*`):

```
from module_name import *
```

przykładowo

```
from math import *
```

jednak z tej opcji powinno się korzystać *wyłącznie* podczas pracy w interaktywnej konsoli języka Python.

Ważne

Bezpośrednie importowanie wszystkich obiektów z modułu, za pomocą instrukcji postaci

```
from module_name import *
```

prowadzi do **zaśmiecenia przestrzeni nazw** (ang. namespace pollution) zwiększając ryzyko konfliktu oznaczeń – np. gdy zarówno moduł `A` jak i moduł `B` definiują własną funkcję `foo()`.

Wiele osób korzysta z powyższego zapisu „`from module_name import *`” dla wygody, aby uniknąć pisania długich kwalifikowanych nazw. Język Python pozwala jednak rozwiązać ten problem w inny sposób – poprzez przemianowywanie przestrzeni nazw podczas importowania modułu:

```
import module_name as alias  # przemianuj przestrzeń nazw
                             #   `module_name` na `alias`
```

Przykładowo, niepisana zasada mówi, aby moduł `numpy` importować za pomocą:

```
import numpy as np  # przemianuj przestrzeń nazw `numpy` na `np`
```

dzięki czemu możemy korzystać z obiektów w `numpy` za pomocą nazw kwalifikowanych z użyciem „`np.`” (zamiast „`numpy.`”):

```
>>> np.e
2.718281828459045
```

4.3 Projektowanie i kodowanie modułów

Jak wspomniano, tworzenie modułów nie wymaga stosowania specjalnej składni. Utwórzmy zatem moduł `fibonacci` zawierający funkcje do obliczania n-tego wyrazu ciągu Fibonacciego w sposób rekurencyjny (`fib(n)`) i iteracyjny (`fibi(n)`) – w tym celu umieścimy w pliku `fibonacci.py` następujący kod:

```
# fibonacci.py

def fib(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

def fibi(n: int) -> int:
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Tak utworzony moduł `fibonacci` jest gotowy do użytku, wystarczy że zaimportujemy go do innego modułu lub skryptu:

```
>>> import fibonacci
>>> fibonacci.fib(7)
13
>>> fibonacci.fibi(20)
6765
```

4.4 Pakiety

Gdy kilka modułów zawiera powiązaną ze sobą funkcjonalność, warto zgrupować je w postaci **pakietu** (ang. package). Stosowanie pakietów pozwala na utworzenie hierarchii podobnej do zagnieżdżonych przestrzeni nazw w języku C++.

Pakietem jest każdy znajdujący się w ścieżce interpretera języka Python[2] katalog zawierający plik `__init__.py`, który to plik wykonywany jest podczas importowania pakietu. Plik `__init__.py` *może* zawierać instrukcje służące konfiguracji pakietu – ustawieniu wartości, zaimportowaniu innych pakietów i modułów (natomiast może też być pusty).

Sama instrukcja importowania pakietu nie różni się niczym od instrukcji importowania „zwykłego” modułu.

Prosty przykład

Aby utworzyć przykładowy pakiet `simple_package` zawierającą moduły `a` oraz `b`, musimy utworzyć katalog `simple_package` zawierający pliki `__init__.py`, `a.py` oraz `b.py`.
Przyjmijmy, że zawartość pliku `a.py` będzie następująca:

```
def bar():
    print("Called 'bar' from module 'a'")
```

a zawartość pliku `b.py` następująca:

```
def foo():
    print("Called 'foo' from module 'b'")
```

natomiast plik `__init__.py` pozostawimy pusty.

Sprawdźmy działanie naszego pakietu w interaktywnej konsoli języka Python:

[2] zob. rozdz. 7.3 Python Search Path

```
>>> import simple_package
>>>
>>> simple_package
<module 'simple_package' from '.../simple_package/__init__.py'>
>>>
>>> simple_package.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
>>> simple_package.b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Jak widać, choć sama paczka `simple_package` została zaimportowana, to jednak ani moduł `a`, ani moduł `b`, nie zostały załadowane! Aby nie było konieczności ręcznego importowania każdego z modułów za pomocą poniższej instrukcji

```
from simple_package import a, b  # Uciążliwe, jeśli zwykle potrzebujemy
                                 #   skorzystać z obu modułów naraz...
```

którą należałoby dodać w każdym pliku, w którym korzystamy z paczki `simple_package`, najlepiej umieścić stosowny kod odpowiedzialny za ładowanie wspomnianych modułów w pliku `__init__.py`:

```
import simple_package.a
import simple_package.b
```

Po wprowadzeniu takiej zmiany możemy od razu korzystać z pełnej funkcjonalności paczki `simple_package`:

```
>>> import simple_package
>>>
>>> simple_package.a.bar()
Called 'bar' from module 'a'
>>>
>>> simple_package.b.foo()
Called 'foo' from module 'b'
```

Rozdział 5

Błędy i wyjątki

We wcześniejszych rozdziałach występowały już wiadomości o błędach (np. `TypeError`), choć nie były szerzej omawiane. W Pythonie wyróżniamy (co najmniej) dwa typy błędów: **błędy składniowe** (ang. syntax errors) oraz **wyjątki** (ang. exceptions).

Dla zainteresowanych...

Materiały źródłowe:

- The Python Tutorial: Errors (PyDocs)
- The definitive guide to Python exceptions (by Julien Danjou)

5.1 Błędy składniowe

Błędy składniowe, in. **błędy parsowania** (ang. parsing errors), są związane z napisaniem fragmentu kodu niezgodnie z gramatyką języka, przykładowo:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                    ^
SyntaxError: invalid syntax
```

Parser wyświetla problematyczną linijkę oraz pokazuje małą „strzałkę" w najwcześniejszym miejscu, w którym błąd został wykryty – zatem błąd został spowodowany przez coś, co stoi po lewej stronie od „strzałki" (w tym przypadku przez brakujący dwukropek po warunku pętli). Oprócz tego w komunikacie o błędzie wyświetlana jest nazwa pliku oraz numer linii pliku, co ułatwia lokalizację błędu w skrypcie.

5.2 Wyjątki

Nawet wyrażenie poprawne składniowo może spowodować błąd wykonania – takie błędy nazywamy **wyjątkami** (ang. exceptions). Wyjątki niekoniecznie muszą powodować przerwanie dalszego wykonania programu, możemy je obsłużyć. Oto przykład komunikatu o nieobsłużonym wyjątku:

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Komunikat zawiera m.in. informację o typie błędu – w powyższym przykładzie: `TypeError`.

Komunikaty zawierają również rozwinięty **stos wywołań** (ang. stack traceback), dzięki któremu widzimy, w którym momencie wykonania programu wystąpił błąd. Przykładowo, próba uruchomienia poniższego programu:

```
def foo():
    return 1 / 0  # UWAGA: Dzielenie przez 0!

def bar():
    foo()
```

```
bar()
```

da następujący rezultat:

```
Traceback (most recent call last):
  File "...", line 7, in <module>
    bar()
  File "...", line 5, in bar
    foo()
  File "...", line 2, in foo
    return 1/0
ZeroDivisionError: division by zero
```

Warto podkreślić, że (w przeciwieństwie do języka C++) w języku Python wyjątki są stosowane powszechnie. Wynika to z filozofii języka Python, zgodnie z którą „lepiej prosić o przebaczenie, niż o pozwolenie".

5.3 Obsługa wyjątków

Wyjątki możesz obsłużyć za pomocą instrukcji **try** zawierającej choć jedną klauzulę **except**, przykładowo:

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Oops! Can't divide by zero.")
```

Schemat wykonania instrukcji **try** zawierającej klauzule **except** jest analogiczny do instrukcji **try** z klauzulami **catch** w języku C++:

- Najpierw wykonywana jest klauzula **try**.
- Jeśli nie wystąpiły żadne wyjątki, klauzule **except** są pomijane i wykonanie programu będzie kontynuowane po instrukcji **try**.
- Jeśli w trakcie wykonywania klauzuli **try** wystąpił błąd, dalsze instrukcje wewnątrz klauzuli **try** są pomijane. Zaczyna się przeglądanie klauzul **except** w poszukiwaniu pierwszej takiej, która wychwytuje rzucony typ wyjątku – czyli takiej, która specyfikuje ten sam typ wyjątku albo jego nadklasę (tj. dowolną z klas macierzystych); np. **except** `BaseException` wychwyci w szczególności wszystkie obiekty wyjątków wbudowanych typów wyjątków.
  - Jeśli odpowiednia klauzula **except** zostanie znaleziona, zostanie ona wykonana, a następnie wykonanie programu będzie kontynuowane po instrukcji **try** (dalsze klauzule **except** nie będą rozpatrywane).
  - Jeśli wyjątek nie został obsłużony przez żadną z klauzul **except**, zostaje on przekazany do zewnętrznej instrukcji **try** (o ile takowa istnieje). Jeśli nie ma już więcej zewnętrznych instrukcji **try**, a wyjątek wciąż nie został obsłużony, wykonanie programu zostaje przerwane – z odpowiednim komunikatem o błędzie.

Chcąc w ramach jednej klauzuli obsłużyć kilka wyjątków możemy po **except** umieścić krotkę zawierającą nazwy wychwytywanych typów wyjątków[1], przykładowo:

```
try:
    # ...
except (RuntimeError, TypeError, NameError):
    pass
```

Instrukcja **try** może posiadać klauzulę **else** umieszczaną po klauzulach **except** – klauzula **else** wykonywana jest wówczas, gdy wewnątrz klauzuli **try** nie wystąpił wyjątek (zob. przykł. 5.1). Takie rozwiązanie jest zdecydowanie lepsze od umieszczania dodatkowego kodu wewnątrz klauzuli **try**, gdyż wówczas moglibyśmy przypadkowo wychwycić (i obsłużyć) wyjątki, które nie zostały rzucone przez kod oryginalnie przeznaczony do ochrony przez **try**-**except**.

[1] W wersji języka Python 2 składnia **except** `Exception, e` służyła do powiązania wyjątku z opcjonalnym parametrem (w tym przypadku o nazwie `e`), który pozwalał na późniejszą dalszą inspekcję wyjątku.

Listing 5.1. Instrukcja `try` z klauzulą `else`.

```
try:
    f = open(arg, 'r')
except OSError:
    print('cannot open', arg)
else:
    print(arg, 'has', len(f.readlines()), 'lines')
    f.close()
```

5.4 Rzucanie wyjątków

Programista może wymusić rzucenie wyjątku (obiektu klasy pochodnej względem `BaseException`) za pomocą instrukcji `raise`:

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Instrukcja ta przyjmuje tylko jeden argument, określający obiekt wyjątku lub klasę wyjątku (w tym drugim przypadku zostanie domyślnie wywołany konstruktor bezargumentowy):

```
raise ValueError('X')    # rzuć jawnie utworzoną instancję klasy `ValueError`
raise ValueError         # równoważne: `raise ValueError()`
raise 1                  # NIEZALECANE: obiekt wyjątku nie dziedzczy
                         #   po `Exception`
```

5.5 Wyjątki zdefiniowane przez użytkownika

W programie możesz definiować własne typy wyjątków, przy czym przestrzegaj wówczas kilku opisanych poniżej zasad i dobrych praktyk.

Twój własny typ wyjątku powinien dziedziczyć (bezpośrednio albo pośrednio) przynajmniej po standardowej klasie `Exception`[2]:

```
class MyOwnError(Exception):
    pass
```

Nazwa typu wyjątku zazwyczaj kończy się członem `Error` (taka konwencja została przyjęta dla standardowych typów wyjątków).

Gdy tworzysz moduł, który może rzucać kilka różnych (własnych) wyjątków, do dobrych praktyk należy zdefiniowanie własnej klasy nadrzędnej w stosunku do wszystkich tych wyjątków – wówczas użytkownicy tego modułu mogą w wygodny sposób wychwytywać wszystkie te wyjątki (podając nazwę klasy nadrzędnej). Przykładowo:

```
class ShoeError(Exception):
    """Podstawowa klasa wyjątku rzucana przy złym korzystaniu z butów."""

class UntiedShoelace(ShoeError):
    """Niezawiązane buty."""

class WrongFoot(ShoeError):
    """But założony na złą nogę."""
```

Jeśli to ma sens, twoje klasy błędów powinny również dziedziczyć po standardowych typach wyjątków (zob. rozdz. 5.9), przykładowo:

```
class CarError(Exception):
    """Podstawowa klasa wyjątku rzucana przez samochody."""
```

[2] zob. *5.9 Hierarchia wbudowanych wyjątków*

```
class InvalidColor(CarError, ValueError):
    """Rzucany, gdy samochód ma niepoprawny kolor."""
```

W ten sposób programy korzystające z twojego modułu mogą wychwytywać błędy w sposób bardziej ogólny – w szczególności nie muszą być świadome istnienia zdefiniowanych przez ciebie typów błędów (w powyższym przypadku wystarczy, że będą wychwytywać standardowy typ `ValueError`).

Zwykle definicja nowego wyjątku jest zwięzła i zawiera tylko dodatkowe atrybuty niezbędne do skutecznej diagnostyki błędu i jego obsługi (zob. przykł. 5.2). W przypadku definiowania własnej metody `__init__()` należy wywołać metodę `__init__()` klasy macierzystej. W nadrzędnej klasie (w przykładzie 5.2: w klasie `CarError`) należy wywołać ją z jednym argumentem – wartością, która zostanie wypisana przez `BaseException.__str__()` jako komunikat diagnostyczny.

Listing 5.2. Definiowanie nowego wyjątku we własnym module (własnej bibliotece).

```
class CarError(Exception):
    """Podstawowa klasa wyjątku rzucana przez samochody."""

    def __init__(self, car, msg=None):
        if msg is None:
            # Ustaw domyślny użyteczny (!) komunikat
            msg = f"An error occured with car {car}"
        super().__init__(msg)  # wywołanie konstruktora klasy 'Exception'
        self.car = car

class CarCrashError(CarError):
    """Gdy jechałeś zbyt szybko i miałeś kolizję."""

    def __init__(self, car, other_car, speed):
        super().__init__(
            car, msg=f"Car crashed into {other_car} at speed {speed}")
        self.speed = speed
        self.other_car = other_car
```

5.6 Instrukcja `with`

Czasem zachodzi potrzeba, aby pewne akcje „sprzątające” zostały wykonane niezależnie od powodzenia innych operacji – m.in. chcemy zawsze po zakończeniu korzystania z zasobów systemowych zwolnić te zasoby.

(Anty)przykład 5.3 zawiera program wypisujący zawartość pliku tekstowego na konsolę. W przykładzie tym instrukcja **`try`** (z klauzulą **`finally`**) została użyta tylko po to, aby mieć gwarancję zamknięcia pliku. Choć program ten jest poprawny, jego czytelność pozostawia sporo do życzenia.

Listing 5.3. (Anty)przykład korzystania z zasobów systemowych.

```
f = open("myfile.txt")
try:
    for line in f:
        print(line, end="")
finally:
    f.close()
```

Pod kątem takich przypadków język Python udostępnia słowo kluczowe **`with`** służące do definiowania **kontekstu** (ang. context) wykonania dla zadanej instrukcji. Zwykle **`with`** występuje w połączeniu z klauzulą **`as`**, która pozwala na wykorzystanie wyników zwracanych przez instrukcję powiązaną z **`with`** wewnątrz kontekstu (zob. przykł. 5.4).

Cała obsługa wyjątków jest zawarta w kodzie zarządzającym **`with`**, natomiast tzw. **menedżer kontekstu** (ang. context manager) powiązany z instrukcją następującą po **`with`** definiuje co ma się dziać podczas wcho-

Listing 5.4. Przykład korzystania z zasobów systemowych z użyciem `with`.

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

dzenia do kontekstu i podczas opuszczania kontekstu – w przykładzie 5.4 menedżer kontekstu powiązany z funkcją `open()` odpowiednio otwiera i zamyka zadany plik. Istnieje możliwość definiowania własnych menedżerów kontekstu.

Dla zainteresowanych...

Materiały źródłowe i dodatkowe:

- Python with Context Managers (by Jeff Knupp)
- The `with` statement (PyDocs)
- Context Manager Types (PyDocs)
- `contextlib` – Utilities for `with`-statement contexts (PyDocs)
- Python in the real world: Context Managers (Arnav Khare)

5.7 Rzucaj wyjątki zamiast zwracać `None`

Ponieważ język Python udostępnia specjalną wartość `None`, początkującym programistom piszącym w tym języku wydaje się kusząca perspektywa zwracania takiej wartość w przypadku wystąpienia błędu w funkcji. Przykładowo, funkcja dzieląca dwie liczby mogłaby zwracać `None` w przypadku, gdy dzielnik wynosi 0 (co wydaje się naturalnym, ponieważ wartość operacji dzielenia nie jest wówczas zdefiniowana):

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None  # ZŁA PRAKTYKA: zwracanie `None` w celu
                     #   sygnalizowania błędów
```

Z takiej funkcji można korzystać w poniższy sposób:

```
result = divide(x, y)
if result is None:
    print('Invalid input')
```

Jednak wówczas przerzucamy na użytkownika odpowiedzialność za pamiętanie o sprawdzaniu wartości zwróconej przez `divide()` i odpowiednim obsługiwaniu takich szczególnych przypadków – co łatwo prowadzi do dalszych błędów, przykładowo wykonanie poniższego programu:

```
distance = 1
velocity = 0
ride_time = divide(distance, velocity)
print('Ride time: ', str(ride_time))
```

zwróci następujący wynik:

```
Ride time:  None
```

Zamiast „*zwracać* obiekt `None`" zdecydowanie lepszym rozwiązaniem (czytelnym i pozwalającym zmniejszyć ryzyko błędów) jest „*rzucić* odpowiedni wyjątek" – w ten sposób użytkownik, który zapomni obsłużyć wyjątek, zostanie o tym niezwłocznie poinformowany. Przykład 5.5 prezentuje przykładową implementacja funkcji `divide()` z użyciem wyjątków – wyjątek `ZeroDivisionError` zostaje wychwycony i zamieniony na `ValueError`, aby wskazać użytkownikowi prawdziwą przyczynę błędu (zła wartość argumentu).

Ważne

Pamiętaj, że informacja o potencjalnie rzucanych wyjątkach powinna się znaleźć w dokumentacji funkcji!

Listing 5.5. Rzucanie wyjątków z funkcji.

```
def divide(a, b):
    """Divide a by b.

    # sekcje `Args` i `Returns` ...

    Raises:
        ValueError: If b = 0.
    """
    try:
        return a / b
    except ZeroDivisionError as e:
        # Rzuć nowy obiekt wyjątku na podstawie wychwyconego obiektu wyjątku.
        raise ValueError('Divisor cannot be zero.') from e

# ----
try:
    result = divide(1, 2)
except ValueError:
    print('Invalid input')
else:
    print('Result is %.1f' % result)
```

5.8 Luźne uwagi

- Wyjątki służą do sygnalizowania sytuacji, które nie są częścią normalnego (oczekiwanego) przebiegu wykonania programu.
  Przykładowo, funkcja `str.find()` zwraca wartość `-1`, gdy wzorzec nie zostanie znaleziony w łańcuchu znaków (gdyż to normalna sytuacja), natomiast próba odwołania się do elementu o indeksie spoza zakresu powoduje zgłoszenie wyjątku.
- Nigdy nie stosuj mechanizmu wyjątków do kontroli przepływu sterowania w programie – np. zamiast warunku pętli.
- Zawsze zadawaj sobie pytanie: „czy to właściwe miejsce na obsługę danego wyjątku?”
  Wyjątki powinny być obsługiwane w miejscu, w którym mamy wystarczająco dużo informacji, aby skutecznie je obsłużyć. Przykładowo, obsługę błędów programistycznych (np. `IndexError`, `TypeError`, `NameError` itd.) zwykle najlepiej zostawić programiście/użytkownikowi, gdyż próba ich „obsługi” jedynie ukryje faktyczną przyczynę problemów.

5.9 Hierarchia wbudowanych wyjątków

Projektując własne typy wyjątków zapoznaj się z hierarchią typów dostępnych w bibliotece standardowej (na stronie Built-in Exceptions: Exception hierarchy (PyDocs)) i wybierz możliwie wyspecjalizowany typ odpowiadający Twojej sytuacji.

Rozdział 6

Be pythonic!

Można programować w języku Python tkwiąc mentalnie w rozwiązaniach rodem z języków C i C++. Można też przestawić się na sposób myślenia programisty języka Python, ale nie znać sposobów na zwięzłą i efektywną realizację pożądanych operacji.

Niniejszy rozdział ma na celu uzupełnić Twoją wiedzę o zagadnienia, których znajomość świadczy o byciu „prawdziwym" programistą języka Python.

6.1 Zen of Python

Prawdopodobnie najlepsze zestawienie zasad filozofii Pythona zostało stworzone przez Tima Petersa, wieloletniego współtwórcę języka Python i aktywnego użytkownika grupy dyskusyjnej `comp.lang.python`. Ten „poemat", tzw. *Zen of Python*, w zwięzły sposób ujmuje najważniejsze kwestie związane z tworzeniem programu zgodnego z filozofią Pythona, w związku z tym nie tylko został mu poświęcony osobny PEP[1], ale także sam język pozwala na szybki dostęp do *Zen* (ot, taki *easter egg*. . .):

```
>>> import this
The Zen of Python, by Tim Peters
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Ponieważ dla adepta języka Python sama lektura *Zen of Python* ma niewielką wartość poznawczą, poniżej pokrótce omówiono wybrane wersy. Pełniejsze omówienie tych zasad znajdziesz w książce *Pro Python: Advanced coding techniques and tools* pióra Marty'ego Alchina.

[1] zob. PEP 20 – The Zen of Python

6.1.1 Beautiful Is Better Than Ugly

„Piękno” to dość subiektywne kryterium, jednak spoglądając na przykład 6.1, zawierający warianty metod służących do walidacji danych formularza, możemy z całą pewnością stwierdzić, że pierwsza propozycja jest „piękniejsza” od drugiej – nazwa funkcji `is_valid()` od razu sugeruje typ zwracanej wartości (wartość logiczna), natomiast nazwa `validate()` nawet nie sugeruje zwracania wartości.

Listing 6.1. „Piękno” kodu – na przykładzie nazewnictwa.

```
# wariant "piękny" - czytelny, preferowany
is_valid = form.is_valid(include_hidden_fields=True)

# wariant "brzydki" - mało czytelny
is_valid = form.validate(include_hidden_fields=True)
```

6.1.2 Explicit Is Better Than Implicit

Programista powinien możliwie jasno komunikować swoją intencję. Przykładowo, podczas wywołania funkcji warto korzystać z argumentów słownikowych, gdyż wtedy od razu wiadomo, do którego parametru odnosi się dany argument (tj. gdy nie trzeba szukać takiej informacji w dokumentacji) – zob. przykł. 6.2.

Listing 6.2. Komunikowanie intencji – na przykładzie argumentów funkcji.

```
# wariant "jawny" - czytelny, preferowany
is_valid = form.is_valid(include_hidden_fields=True)

# wariant "niejawny" - mało czytelny
is_valid = form.is_valid(True)  # jakiemu parametrowi odpowiada `True'?!
```

6.1.3 Simple Is Better Than Complex

Typowym przykładem zastosowania tej zasady jest skorzystanie w instrukcjach warunkowych z faktu, że większości wyrażeń można przyporządkować wartość logiczną (`False` albo `True`) bez konieczności jawnego testowania[2] (zob. przykł. 6.3).

Listing 6.3. Złożoność rozwiązania – na przykładzie instrukcji warunkowej.

```
# wariant "złożony" - mało czytelny
if value is not None and value != '':
  ...

# wariant "prosty" - czytelny, preferowany
if value:
  ...
```

6.1.4 Complex Is Better Than Complicated

Przyjmijmy następujące rozróżnienie między rozwiązaniem „złożonym” a „skomplikowanym”:

- rozwiązanie *złożone* (ang. complex) – składa się z wielu wzajemnie powiązanych elementów,
- rozwiązanie *skomplikowane* (ang. complicated) – jest tak złożone, a powiązania między elementami tak zagmatwane, że aż ciężko takie rozwiązanie zrozumieć.

Aby uniknąć tworzenia rozwiązań skomplikowanych, powinniśmy m.in. umiejętnie dzielić problem na małe fragmenty i grupować je logiczne (pomaga tu stosowanie spójnego, przemyślanego nazewnictwa obiektów).

[2] Zagadnienie to zostało omówione dokładniej w rozdziale 6.5.1 Unikaj bezpośredniego porównywania z `True`, `False`, oraz `None`.

6.1.5 Flat Is Better Than Nested

Zasada „spłaszczania” (tj. *„«płaskie» jest lepsze niż zagnieżdżone”*) odnosi się m.in. do hierarchii klas, struktury kodu, oraz do sposobu organizacji pakietów.

Przykład 6.4 zawiera zagnieżdżoną, nieczytelną strukturę. Można ją „spłaszczyć”, jak w przykładzie 6.5, co pozwala łatwiej zorientować się w przepływie sterowania.

Listing 6.4. Zagnieżdżona (mało czytelna) struktura instrukcji warunkowych.

```
if x > 0:
    if y > 100:
        raise ValueError("Value for y is too large.")
    else:
        return y
else:
    if x == 0:
        return False
    else:
        raise ValueError("Value for x cannot be negative.")
```

Listing 6.5. Płaska (przejrzysta) struktura instrukcji warunkowych, równoważna strukturze z przykładu 6.4.

```
if x > 0 and y > 100:
    raise ValueError("Value for y is too large.")
elif x > 0:
    return y
elif x == 0:
    return False
else:
    raise ValueError("Value for x cannot be negative.")
```

6.1.6 Sparse Is Better Than Dense

Należy formatować kod źródłowy w taki sposób, aby był czytelny[3] – z jednej strony nie ma sensu na siłę zagęszczać elementów (zob. przykł. 6.6), a z drugiej strony nie należy zbyt hojnie umieszczać odstępów (zob. przykł. 6.7).

Listing 6.6. Zbytnie zagęszczenie kodu może zmniejszać jego czytelność.

```
if x == 4: print('OK')  # źle - zbytnie zagęszczenie

if x == 4:
    print('OK')  # dobrze ("ciało" w osobnym wierszu)
```

Listing 6.7. Zbytnie rozstrzelenie kodu może zmniejszać jego czytelność.

```
spam(ham[1], {eggs: 2})  # OK
spam( ham[ 1 ], { eggs: 2 } )  # źle - za dużo odstępów
```

[3] zob. PEP 8 – Style Guide for Python Code

6.1.7 Readability Counts

Czytelność kodu to również dość subiektywne pojęcie, odnoszące się do bardzo wielu kwestii: nazewnictwa obiektów, formatowania kodu, podziału odpowiedzialności między klasy itd. Chodzi o to, aby mieć na uwadze, że kod jest analizowany nie tylko przez komputery, ale też przez inne osoby (które muszą go utrzymywać – np. dopisywać nowe funkcjonalności). Czytelność ma sprawić, aby ludziom łatwiej było pracować z naszym kodem – obojętnie, czy będą to inni programiści, czy my sami (za kilka tygodni bądź miesięcy).

Zagadnieniu czytelności kodu poświęcony jest PEP 8 – Style Guide for Python Code.

6.1.8 There Should Be One – and Preferably Only One – Obvious Way to Do It

Projektując moduły, klasy bądź funkcje zwracaj uwagę na to, aby użytkownik miał jasność, jakiej funkcjonalności użyć w jakim przypadku. Przykładowo, choć do wartości ze słownika (typu wbudowanego `dict`) możemy odwołać się zarówno poprzez `my_dict['key']`, jak i z użyciem metody `dict.get()`, tylko pierwszy sposób jest oczywisty (i promowany w dokumentacji). Odwołanie się z użyciem nawiasów kwadratowych zakłada wariant optymistyczny i nie dokonuje dodatkowych sprawdzeń, czy klucz faktycznie znajduje się w słowniku. Dopiero gdy ktoś potrzebuje obsługiwać takie szczególne przypadki (np. brak klucza w słowniku), korzysta z metody `get()`. W tym przypadku nie mamy zatem do czynienia z problemem „wielu sposobów na realizację tej samej operacji”, gdyż każdy z tych sposobów będzie użyty w zupełnie odmiennej sytuacji.

6.1.9 Although That Way May Not Be Obvious at First Unless You’re Dutch

Ponieważ dla różnych ludzi różne rzeczy są „oczywiste”, dobrą praktyką jest solidne dokumentowanie swojej pracy. (Wzmianka o „byciu Holendrem” odnosi się do twórcy języka Python, Guido van Rossuma).

6.1.10 Now Is Better Than Never

Rozwiązując problemy warto unikać prokrastynacji, gdyż w miarę upływu czasu zapominamy o różnych istotnych szczegółach (założenia, ograniczenia itp.), które należy uwzględnić w rozwiązaniu. W przypadku języka Python chodzi również o iteracyjne podejście do rozwoju oprogramowania – język Python umożliwia *prototypowanie*, dzięki czemu nie musimy mieć od pierwszej chwili gruntownie przemyślanego całego rozwiązania (ewentualne refaktoryzacje nie są zwykle kosztowne).

6.1.11 If the Implementation is Hard to Explain, It’s a Bad Idea

Oczywiście – „moja racja jest racja najmojsza”[4] – jednak zwykle lepiej skonsultować swój pomysł z inną osobą. Jeśli wytłumaczenie sposobu rozwiązania zajmuje dużo czasu i wysiłku intelektualnego, to jest to najprawdopodobniej „złe” (skomplikowane) rozwiązanie.

6.1.12 If the Implementation is Easy to Explain, It May Be a Good Idea

Pamiętaj jednak, że łatwość wytłumaczenia rozwiązania niekoniecznie oznacza automatycznie, że jest to rozwiązanie „dobre” (poprawne)! Zwykle dopiero rzetelna ocena przez innych programistów pozwala na wypracowanie optymalnego rozwiązania.

6.2 Generatory

Zagadnienie generatorów wykracza poza ramy niniejszego skryptu.

Dla zainteresowanych. . .

Materiały źródłowe:

- Python 3 Tutorial: Generators (Python Course)
- Tutorial – Python List Comprehension With Examples (by Aarshay Jain)

[4] cytat z filmu „Dzień Świra” (2002) w reż. Marka Koterskiego

6.3 Dekoratory

W języku Python **dekoratorem** (ang. decorator) jest każdy dający się wywołać obiekt, który służy do zmiany zachowania funkcji lub klasy – referencja do (oryginalnej) funkcji `func` lub klasy `C` jest przekazywana do dekoratora, który z kolei zwraca zmodyfikowaną funkcję bądź klasę (zwykle zmodyfikowana wersja zawiera odpowiednio „opakowane" użycie oryginalnej funkcji `func` bądź klasy `C`). W związku z tym rozróżniamy dekoratory funkcji i dekoratory klas, mimo iż w obu przypadkach chodzi o to samo pojęcie.

Poniższe podrozdziały omawiają krok po kroku kluczowe aspekty języka Python pozwalające zdefiniować prosty dekorator i użyć go. Szczegóły dotyczące sposobu implementowania dekoratorów znajdziesz w dokumencie PEP 318 – Decorators for Functions and Methods.

Dla zainteresowanych...

Materiały źródłowe:

- Python 3 Tutorial: Decorators (Python Course)

6.3.1 Aliasing

Pamiętaj, że w języku Python nazwy funkcji to w rzeczywistości referencje do obiektów funkcyjnych, oraz że możemy nadać temu samemu obiektowi funkcyjnemu kilka nazw (aliasów):

```
def succ(x):
  return x + 1

successor = succ  # `successor` i `succ` to identyfikatory odnoszące się
                  #   do tego samego obiektu funkcyjnego
```

6.3.2 Zagnieżdżanie funkcji

W języku Python mamy możliwość definiowania **funkcji zagnieżdżonej** (ang. nested function), czyli funkcji zdefiniowanej wewnątrz innej funkcji (zob. przykł. 6.8, 6.9).

Listing 6.8. Funkcja `g()` jest zagnieżdżona w funkcji `f()`.

```
def f():
    def g():
        print("This is the function 'g'")

    print("This is the function 'f'")
    g()

# --- Python Console ---
>>> f()
This is the function 'f'
This is the function 'g'
```

Listing 6.9. Zagnieżdżenie funkcji zwracających wartość.

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32

    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

# --- Python Console ---
>>> print(temperature(20))
It's 68.0 degrees!
```

6.3.3 Funkcje jako parametry

Ponieważ każdy parametr funkcji jest referencją na obiekt, a funkcje też są obiektami, możemy przekazywać funkcje (a ściślej: referencje do funkcji) jako argumenty funkcji (zob. przykł. 6.10).

Listing 6.10. Przekazywanie funkcji jako argumentu innej funkcji.

```
import math

def foo(func):
    return sum([func(x) for x in [1, 2, 2.5]])

# --- Python Console ---
>>> print(foo(math.sin))
2.3492405557375347
>>> print(foo(math.cos))
-0.6769881462259364
```

6.3.4 Funkcje zwracające funkcje

Ponieważ wartość zwracana przez funkcję to referencja do *dowolnego* obiektu, funkcje mogą zwracać referencje do innych obiektów funkcyjnych (zob. przykł. 6.11).

Listing 6.11. „Fabryka" wielomianów 2. stopnia (tj. wielomianów postaci $W = ax^2 + bx + c$).

```
def polynomial_deg2_factory(a, b, c):
    def polynomial(x):
        return a * x ** 2 + b * x + c
    return polynomial

# --- Python Console ---
>>> p1 = polynomial_deg2_factory(2, 3, -1)
>>> p2 = polynomial_deg2_factory(-1, 2, 1)
>>> x = 2
>>> print("p1({x}) = {p1v}, p2({x}) = {p2v}".format(x=x, p1v=p1(x), p2v=p2(x)))
p1(2) = 13, p2(2) = 1
```

6.3.5 Prosty dekorator

Mamy obecnie opanowane wszystkie niezbędne podstawy, aby zdefiniować nasz pierwszy prosty dekorator (zob. przykł. 6.12).

Listing 6.12. Definicja prostego dekoratora funkcji wypisującego komunikat przed wywołaniu funkcji i po nim.

```
def our_decorator(func: Callable[[Any], Any]):
    def function_wrapper(x: Any):
      # obiekty funkcyjne zawierają atrybut `__name__`
      #   przechowujący identyfikator funkcji
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper
```

Po uruchomieniu programu testującego dekorator:

```
def foo(x):
```

```
    print("Hi, foo has been called with " + str(x))

print("We call foo before decoration:")
foo("Hi")

print("We now decorate foo with f:")
foo = our_decorator(foo)

print("We call foo after decoration:")
foo(42)
```

otrzymamy poniższy wynik:

```
Hi, foo has been called with Hi
We now decorate foo with f:
We call foo after decoration:
Before calling foo
Hi, foo has been called with 42
After calling foo
```

Zwróć uwagę, że po zastosowaniu dekoracji

```
foo = our_decorator(foo)
```

uchwyt `foo` staje się referencją do obiektu funkcyjnego `function_wrapper`. Obiekt funkcyjny `foo` wciąż będzie wywołany wewnątrz `function_wrapper`, lecz zarówno przed tym wywołaniem, jak i po nim zostanie wykonany dodatkowy kod (w tym przypadku: wypisane zostaną informacje diagnostyczne).

6.3.6 Typowa składnia dekoratorów w języku Python

W języku Python zwykle unikamy dekorowania za pomocą przypisania:

```
foo = our_decorator(foo)
```

gdyż ten sposób sprawia, że w programie istnieją dwie wersje danego obiektu funkcyjnego – przed i po dekoracji. Zamiast tego dekorację wykonuje się w linii poprzedzającej nagłówek funkcji – za pomocą symbolu „`@`”, po którym następuje nazwa dekoratora, np.:

```
@our_decorator
def foo(x):
    ...
```

Ponieważ sygnatura funkcji `our_decorator` ma postać

```
def our_decorator(func: Callable[[Any], Any]):
```

za pomocą tego dekoratora możemy udekorować każdą inną *jednoparametrową* funkcję, przykładowo:

```
@our_decorator
def succ(n):
    return n + 1

succ(10)
```

Rezultat wykonania powyższego programu:

```
Before calling succ
11
After calling succ
```

6.3.7 Przykład zastosowania dekoratorów

Powiedzmy, że Twoim zadaniem jest stworzenie biblioteki jednoargumentowych funkcji matematycznych. Część z nich (np. silnia) wymaga, aby argument był liczbą naturalną – podanie niedozwolonej liczby powinno skutkować rzuceniem wyjątku odpowiedniego typu. Aby uniknąć zaciemniania kodu takich funkcji matematycznych wywołaniem pomocniczej funkcji weryfikującej, możesz napisać i zastosować odpowiedni dekorator (zob. przykł. 6.13).

Listing 6.13. Dekorator weryfikujący poprawność argumentu funkcji.

```
def argument_test_natural_number(f):
    def helper(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("The argument is not an integer")
    return helper

@argument_test_natural_number
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

6.4 Docstrings

Materiały źródłowe:

- PEP 257 – Docstring Conventions
- Google vs NumPy style

6.5 Idiomatyczny język Python

Oto znany cytat dotyczący pisania kodu w taki sposób, aby był on łatwy w utrzymaniu:

```
Always code as if the guy who ends up maintaining your code
will be a violent psychopath who knows where you live.
--John Woods comp.lang.c++
```

Któż spośród nas nie był zmuszony choć raz w życiu „przegryzać się” przez cudzy kod, klnąc pod nosem na zastosowane w nim karkołomne konstrukcje?

Choć język Python został zaprojektowany tak, aby w naturalny sposób wspierać łatwe zarządzanie kodem, jego nieumiejętne stosowanie będzie skutkowało tworzeniem kodu tak samo nieczytelnego, jakim będzie źle napisany kod w każdym innym języku. Pisanie czytelnego kodu wymaga od nas, programistów, pewnego wysiłku. Co zatem robić, aby zmniejszyć cierpienie w skutek analizowania słabego kodu?

Pisz. Idiomatyczny. Kod.

Idiomy w odniesieniu do języków programowania to rodzaj powszechnie zrozumiałego języka ułatwiającego przyszłym czytelnikom kodu precyzyjne zrozumienie naszych intencji. Luźne przyrównanie do sytuacji z życia codziennego: idiomatycznym jest podawanie ceny w formie „5 złotych i 18 groszy”, a nie „518 groszy”. Teoretycznie nie ma nic złego w podawaniu ceny w groszach, lecz prawdopodobnie większość z nas byłaby zaskoczona i zdezorientowana widząc taki zapis. . . Podobnie stosowanie idiomów programistycznych pozwala łatwiej zorientować się w tym, co dzieje się w kodzie – zmniejsza wysiłek poznawczy.

(Poniższe przykłady pochodzą głównie z książki *Writing Idiomatic Python*, Jeff Knupp (2013).)

6.5.1 Unikaj bezpośredniego porównywania z `True`, `False`, oraz `None`

Z każdym obiektem – zarówno wbudowanym jak i zdefiniowanym przez użytkownika – związana jest jego „prawdziwość" (ang. "truthiness"). Podczas sprawdzania warunków logicznych staraj się polegać na niejawnej „prawdziwości" obiektu, zwłaszcza że zasady określania „prawdziwości" są dość jasne.

Wartość logiczna każdego z poniższych wyrażeń to `False`:

- `None`
- `False`
- zero dla typów numerycznych (tj. `int` i `float`)
- puste kontenery (tj. `[]`, `{}`, and `()`)
- puste łańcuchy znaków (np. `''`)

natomiast wszelkie inne wyrażenia są traktowane jako posiadające wartość logiczną `True` (zatem „prawdziwość" w większości przypadków wynosi `True`).

Instrukcja `if` korzysta z „prawdziwości" wyrażeń w sposób niejawny – poniższe dwa warunki są równoważne:

```
# szkodliwe
if foo == True:
    pass

# idiomatyczne
if foo:
    pass
```

jednak lepiej stosować sposób idiomatyczny – głównie ze względu na łatwość wprowadzania zmian projektowych. Przykładowo, jeśli typ `foo` zmieni się z wartości logicznej na typ całkowity, sposób idiomatyczny pozostanie poprawny (w przeciwieństwie do sposobu wykorzystującego bezpośrednie porównanie do obiektu `True`, którego wartość całkowita wynosi `1`).

W szczególności „prawdziwość" wyrażenia można wykorzystać do zwięzłego sprawdzania, czy kontener zawiera elementy – zamiast wywołania funkcji `__len__()`:

```
c = [1]  # pewien kontener

# szkodliwe
if len(c) > 0:
    pass

# idiomatyczne
if c:
    pass
```

Zwróć uwagę, że w niektórych przypadkach jesteśmy zmuszeni do bezpośredniego porównywania z `None` – w szczególności w funkcjach przyjmujących argument o domyślnej wartości `None`, aby sprawdzić czy został on jawnie określony:

```
def insert_value(value, position=None):
    """Inserts a value into my container, optionally at the
    specified position"""
    if position is not None:
        ...
```

Użycie warunku `if position:` nie będzie poprawne, gdyż wówczas przekazanie argumentu 0 również zostanie potraktowane jak wartość domyślna (gdyż zarówno 0 jak i `None` są traktowane jak `False`).

6.5.2 Stosuj słowo kluczowe `in` do iterowania po `iterable`

Pamiętaj, że w języku Python do iterowania po zakresie służy słowo kluczowe `in` (tzw. styl *for each*) – a *nie* zmienna indeksująca (por. przykł. 6.14 i 6.15).

Listing 6.14. Iterowanie za pomocą zmiennej indeksującej – **szkodliwe**.

```
my_list = ['Larry', 'Moe', 'Curly']
index = 0
while index < len(my_list):
    print (my_list[index])
    index += 1
```

Listing 6.15. Iterowanie za pomocą `in` – **idiomatyczne**.

```
my_list = ['Larry', 'Moe', 'Curly']
for element in my_list:
    print (element)
```

6.5.3 Unikaj powtarzania nazwy zmiennej w złożonych warunkach

Aby sprawdzić, czy zmienna ma jedną z kilku wartości, stosuj słowo kluczowe `in` w połączeniu z obiektem typu `iterable` zawierającym te wartości (por. przykł. 6.16 i 6.17). Dzięki temu unikniesz zaciemniania kodu zbędnymi powtórzeniami nazwy zmiennej.

Listing 6.16. Powtarzanie nazwy zmiennej w złożonym warunku – **szkodliwe**.

```
name = 'Tom'
if name == 'Tom' or name == 'Dick' or name == 'Harry':
    pass
```

Listing 6.17. Korzystanie z `iterable` oraz z `in` – **idiomatyczne**.

```
name = 'Tom'
if name in ('Tom', 'Dick', 'Harry'):
    pass
```

6.5.4 Unikaj umieszczania kodu rozgałęzienia w linii z dwukropkiem

Aby zwiększyć czytelność kodu, umieszczaj instrukcje do wykonania w ramach danego rozgałęzienia w osobnych wierszach – z użyciem wcięcia. Symbol dwukropka (`:`) powinien być ostatnim znakiem w danym wierszu (por. przykł. 6.18 i 6.19).

Listing 6.18. Kod rozgałęzienia w wierszu z dwukropkiem – **szkodliwe**.

```
name = 'Jeff'
address = 'New York, NY'
if name: print(name)
print(address)
```

Listing 6.19. Kod rozgałęzienia w osobnym wierszu, z wcięciem – **idiomatyczne**.

```
name = 'Jeff'
address = 'New York, NY'
if name:
    print(name)
print(address)
```

6.5.5 Stosuj w pętlach funkcję `enumerate()` zamiast tworzenia zmiennej indeksującej

W wielu językach programowania chcąc iterować po kolekcji, a jednocześnie mieć informację o indeksie elementu, zmuszeni jesteśmy do korzystania ze zmiennej indeksującej, np. w języku C++:

```
for (std::size_t i = 0; i < container.size(); ++i) {
    // Do stuff
}
```

W języku Python służy do tego wbudowana funkcja `enumerate()` (por. przykł. 6.20 i 6.21).

Listing 6.20. Uzyskiwanie informacji o indeksie za pomocą zmiennej indeksującej – **szkodliwe**.

```
my_container = ['Larry', 'Moe', 'Curly']
index = 0
for element in my_container:
    print ('{} {}'.format(index, element))
    index += 1
```

Listing 6.21. Uzyskiwanie informacji o indeksie za pomocą `enumerate` – **idiomatyczne**.

```
my_container = ['Larry', 'Moe', 'Curly']
for index, element in enumerate(my_container):
    print ('{} {}'.format(index, element))
```

6.5.6 Stosuj pętle z klauzulą `else`

Pętle mogą zawierać klauzulę **`else`**, wykonywaną:

- w przypadku pętli **`for`** – gdy lista wartości zostanie wyczerpana,
- w przypadku pętli **`while`** – gdy warunek staje się fałszywy,

ale *nie* w sytuacji, gdy pętla zostanie przerwana instrukcją **`break`** albo instrukcją **`return`**.

Oto przykład pętli służącej znajdowaniu liczb pierwszych (z zakresu $[2, 6]$):

```
for n in range(2, 6):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

Rezultat wykonania powyższego programu:

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
```

W poniższym scenariuszu chcemy sprawdzić, czy któryś z adresów e-mail podanych przez użytkownika jest błędny (każdy użytkownik może podać więcej niż jeden adres). Dzięki zastosowaniu konstrukcji **`for-else`** unikamy konieczności stosowania flagi `has_malformed_email_address`, przez co kod staje się krótszy i bardziej przejrzysty (por. przykł. 6.22 i 6.23).

Listing 6.22. Użycie flagi do sprawdzenia trybu wyjścia z pętli – **szkodliwe**.

```
for user in get_all_users():
  has_malformed_email_address = False
  print ('Checking {}'.format(user))
  for email_address in user.get_all_email_addresses():
    if email_is_malformed(email_address):
      has_malformed_email_address = True
      print ('Has a malformed email address!')
      break
  if not has_malformed_email_address:
    print ('All email addresses are valid!')
```

Listing 6.23. Użycie pętli `for` z klauzulą `else` – **idiomatyczne**.

```
for user in get_all_users():
    print ('Checking {}'.format(user))
    for email_address in user.get_all_email_addresses():
        if email_is_malformed(email_address):
            print ('Has a malformed email address!')
            break
    else:
        print ('All email addresses are valid!')
```

6.6 Iterowanie po strukturach danych

W przypadku intensywnej pracy ze strukturami danych warto znać techniki efektywnego iterowania po nich – poniższe odnośniki omawiają to zagadnienie w sposób dość wyczerpujący.

Materiały źródłowe:

- Looping Techniques (PyDocs)
- Transforming Code into Beautiful, Idiomatic Python (by Raymond Hettinger, pycon US 2013)

Istotne różnice między wersjami Python 2 a Python 3:

- Znane z Pythona 2 funkcje `izip()` oraz `xrange()` to odpowiednio funkcje `zip()` i `range()` w Pythonie 3.
- W Pythonie 3 słownik nie posiada metody `iteritems()`, natomiast metoda `items()` zachowuje się podobnie do dawnego `iteritems()` (zob. dokumentację).

Rozdział 7

Zagadnienia dodatkowe

Dokładna znajomość poniższych zagadnień zapewne nie będzie Ci potrzeba w projektach o niewielkiej złożoności, jednak warto mieć świadomość, że „o! coś takiego istnieje!” – dzięki temu później (w potrzebie) wiadomo w ogóle, czego szukać.

Materiały źródłowe:

- The Python Tutorial (PyDocs)
- Python 3 Tutorial (Python Course)

7.1 Garbage collector

Materiały źródłowe:

- Things you need to know about garbage collection in Python (by Artem Golubin)

7.2 Zasięgi i przestrzenie nazw

Przestrzeń nazw to mapowanie nazw na obiekty[1]. Przykładowe przestrzenie nazw to:

- zbiór nazw wbudowanych (obejmuje m.in. nazwy typów prostych, nazwy funkcji w stylu `abs()`),
- zbiór nazw globalnych w danym module, oraz
- zbiór nazw lokalnych w danym wywołaniu funkcji.

W pewnym sensie zbiór atrybutów obiektu również tworzy przestrzeń nazw. Co istotne, nie ma absolutnie żadnej zależności między nazwami w różnych przestrzeniach nazw – w dwóch modułach może istnieć tak samo nazwana funkcja, lecz to nie prowadzi do niejasności, gdyż użytkownik modułu otrzymuje dostęp do właściwej funkcji dopiero po poprzedzeniu jej nazwy nazwą odpowiedniego modułu (podobnie jak w przypadku elementów z danej przestrzeni nazw w języku C++).

Przestrzenie nazw są tworzone w różnych momentach wykonania programu oraz mają różny „czas życia”. Przestrzeń nazw zawierająca nazwy wbudowane tworzona jest w chwili uruchomienia interpretera języka Python i nigdy nie jest usuwana. Globalna przestrzeń nazw dla modułu tworzona jest w momencie wczytania definicji modułu i zwykle również zostaje zachowana w pamięci do momentu zamknięcia interpretera. Instrukcje wykonywane na najwyższym poziomie interpretera (zarówno odczytywane ze skryptu jak i dostarczane w trybie interaktywnym) są traktowane jako część modułu o nazwie `__main__`, zatem posiadają swoją własną globalną przestrzeń nazw (na dobrą sprawę nazwy wbudowane również „żyją” w module o nazwie `builtins`).

Lokalna przestrzeń nazw dla funkcji tworzona jest w momencie wywołania funkcji i usuwana w chwili, gdy funkcja zwraca wartość bądź gdy zgłoszony zostanie wyjątek nieobsłużony wewnątrz tej funkcji. Oczywiście każde z wywołań rekurencyjnych danej funkcji posiada swoją własną, odrębną przestrzeń nazw.

Zasięg (ang. scope) to fragment kodu programu w języku Python (jako tekst), w którym przestrzeń nazw jest bezpośrednio dostępna, przy czym termin „bezpośrednio dostępna” oznacza, że podanie niekwalifikowanego odniesienia do nazwy będzie skutkować próbą znalezienia nazwy w tej właśnie przestrzeni.

Choć zasięgi są określane statycznie, ich użycie jest dynamiczne. W każdym momencie wykonywania programu dostępne są bezpośrednio przestrzenie nazw co najmniej trzech zasięgów:

[1] Większość przestrzeni nazw jest obecnie zaimplementowanych jako słowniki (typ `dict`), lecz dla użytkownika nie ma to znaczenia – implementacja może też ulec zmianie w kolejnych wersjach Pythona.

- zasięg najgłębszy – przeszukiwany w pierwszej kolejności – zawiera nazwy lokalne
- zasięgi wszelkich funkcji „otaczających”[2] – przeszukiwane w kolejności począwszy od ostatnio wywołanej – zawierają nazwy, które nie są ani lokalne, ani globalne
- zasięg „przedostatni” – zawiera nazwy globalne aktualnego modułu
- zasięg zewnętrzny – przeszukiwany jako ostatni – to przestrzeń nazw zawierająca nazwy wbudowane

Jeśli dana nazwa zostanie zadeklarowana jako globalna, wtedy wszystkie odniesienia i przypisania są umieszczane w zasięgu pośrednim, zawierającym nazwy globalne modułu. Aby zmienić powiązanie zmiennych „żyjących” w zakresie innym niż najgłębszy, należy użyć instrukcji `nonlocal` – w przeciwnym razie te zmienne będą traktowane jako „tylko do odczytu” (próba zapisu do takiej zmiennej po prostu spowoduje utworzenie *nowej* zmiennej lokalnej w najgłębszym zakresie, pozostawiając identycznie nazwaną zmienną w „zewnętrznym” zakresie w niezmienionej postaci).

Zwykle zasięg lokalny odnosi się do nazw lokalnych utworzonych wewnątrz obecnej funkcji (w ujęciu kodu programu). Poza funkcją zasięg lokalny odnosi się do tej samej przestrzeni nazw, co zasięg globalny – do przestrzeni nazw modułu. Definicje klas również tworzą osobne zasięgi lokalne.

Należy podkreślić, że zasięgi są określane na podstawie *kodu* (tekstu) programu – zasięg globalny funkcji zdefiniowanej w module jest tożsamy z przestrzenią nazw tego modułu, niezależnie skąd i z użyciem jakiego aliasu została wywołana taka funkcja. Z drugiej strony faktyczne poszukiwanie nazw odbywa się dynamicznie, w czasie działania programu – jednak język Python ewoluuje w stronę statycznego tłumaczenia nazw, w czasie „kompilacji”, zatem nie polegaj na dynamicznym tłumaczeniu nazw! (W rzeczywistości zmienne lokalne są określane statycznie już obecnie.)

Pewnym szczególnym „dziwactwem” języka Python jest to, że – w przypadku gdy nie użyjemy w danym momencie instrukcji **`global`** – przypisania do nazw są zawsze umieszczane w najgłębszym zasięgu. Przypisania nie powodują skopiowania danych – one jedynie przyporządkowują nowe, dodatkowe nazwy (aliasy) obiektom. To samo odnosi się do operacji usuwania – instrukcja „**`del`** `x`” usuwa nazwę (alias) `x` z przestrzeni nazw do której odnosi się dany zasięg lokalny. W rzeczywistości wszystkie operacje, które wprowadzają nowe nazwy, korzystają z zasięgu lokalnego – w szczególności instrukcje **`import`** oraz definicje funkcji wiążą nazwę modułu bądź funkcji z zasięgiem lokalnym.

Instrukcja **`global`** może być użyta do wskazania, że dana zmienna „żyje” w zasięgu globalnym i powinna być „przepięta” (ang. rebound) tamże, z kolei instrukcja `nonlocal` wskazuje, że dana zmienna „żyje” w zasięgu otaczającym i że powinna zostać „przepięta” tamże.

Wykonanie poniższego programu, ilustrującego zachowanie się zasięgów:

```
import sys

def foo():
    print(x)    # zmienna `x` pochodzi z zewnętrznego zasięgu
    sys.stdout.flush()

x = 3
foo()
del x
foo()
```

da następujący wynik:

```
3
Traceback (most recent call last):
  File "...", line 10, in <module>
    foo()
  File "...", line 4, in foo
    print(x)    # zmienna `x` pochodzi z zewnętrznego zasięgu
NameError: name 'x' is not defined
```

Z kolei wykonanie poniższego programu, również ilustrującego zachowanie się zasięgów:

```
spam = "global spam"  # zmienna globalna
```

[2] tj. funkcji znajdujących się w danym momencie na stosie wywołań

```
def scope_test():
    def do_local():
        spam = "local spam"  # Ta zmienna `spam` przesłania zmienną
                             #   z zasięgu globalnego.

    def do_nonlocal():
        nonlocal spam  # Od tego momentu `spam` w zakresie funkcji
                       # `do_nonlocal()` będzie odnosić się do zmiennej
                       #   z najbliższego zewnętrznego zasięgu - w tym
                       #   przypadku do zmiennej `spam` z poziomu funkcji
                       #   `scope_test()`. Jeśli taka zmienna nie istnieje
                       #   w zewnętrznym zasięgu, wystąpi błąd.
        spam = "nonlocal spam"

    def do_global():
        global spam  # Od tego momentu `spam` w zakresie funkcji
                     #   `do_global()` będzie odnosić się do zmiennej
                     #   `spam` z zasięgu globalnego.
        spam = "new global spam"

    spam = "local spam (function)"  # Zmienna lokalna dla funkcji
                                    #   `scope_test()`.
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

da następujący wynik:

```
After local assignment: local spam (function)
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: new global spam
```

7.3 Python Search Path

Materiały źródłowe:

- How does python find packages? (by Lee Mendelowitz)

7.4 Method Resolution Order

W jaki sposób funkcja wbudowana `super()` rozwiązuje dylemat „metodę której klasy wywołać”? Służy do tego metoda `mro()` (od **method resolution order**, MRO), której sercem jest **algorytm linearyzacji C3** (ang. C3 superclass linearisation algorithm). Algorytm ten określamy jako „algorytm linearyzacji”, gdyż drzewiasta struktura klas zostaje zamieniona na porządek liniowy.

Przykładowo, dla poniższej struktury klas:

```
class A:
    def __init__(self):
        pass

class B(A):
    def __init__(self):
        super().__init__()

class C(A):
```

```
    def __init__(self):
        super().__init__()

class D(B,C):
    def __init__(self):
        super().__init__()
```

za pomocą metody `mro()` uzyskamy poniższe listy:

```
>>> from super_init import A,B,C,D
>>> D.mro()
[<class 'super_init.D'>, <class 'super_init.B'>,
 <class 'super_init.C'>, <class 'super_init.A'>,
 <class 'object'>]
>>> B.mro()
[<class 'super_init.B'>, <class 'super_init.A'>,
 <class 'object'>]
>>> A.mro()
[<class 'super_init.A'>, <class 'object'>]
```

Rozważ następującą hierarchię klas:

```
class A:
    x = 1

class B(A):
    pass

class C(A):
    pass
```

oraz ciąg instrukcji:

```
>>> print(A.x, B.x, C.x)
1 1 1

>>> B.x = 2
>>> print(A.x, B.x, C.x)
1 2 1

>>> A.x = 3
>>> print(A.x, B.x, C.x)
3 2 3
```

O ile pierwsze dwa wyniki nie budzą wątpliwości, trzeci może być zaskoczeniem – dlaczego zmiana `A.x` spowodowała zmianę `C.x`?! Ma to związek z zasadami wybierania składowej – jeśli składowa o danej nazwie nie zostanie znaleziona w aktualnej klasie, przeszukiwane są po kolei jej klasy macierzyste[3] zgodnie z kolejnością określoną przez metodę `mro()`.

W powyższym przykładzie instrukcja `B.x = 2` spowodowała utworzenie nowego atrybutu klasowego w klasie `B` i dlatego podczas późniejszego odwołania do `B.x` wybrany został ten nowo utworzony atrybut. Z kolei klasa `C` nie posiada własnego atrybutu `x`, zatem odwołanie do `C.x` wybierze atrybut `x` z klasy `A`.

Dla zainteresowanych...

Materiały źródłowe:

- Python 3 Tutorial: Multiple Inheritance (Python Course)
- Method Resolution Order (by Guido van Rossum)
- Common Mistake #2: Using class variables incorrectly (by Martin Chikilian)

[3] Język Python wspiera wielokrotne dziedziczenie.

Podziękowania

Korekta

Jacek Ankowski
Dawid Bugajny
Filip Gacek
Michał Krzyszczuk
Jakub Mazur
Szymon Majewski
Kacper Moździerz
Aleksander Nagaj
Artur Połeć
Dominika Przewłocka
Igor Ratajczyk
Agata Suliga
Błażej Szargut
Rafał Węgrzyn
Bartosz Więcek
Olaf Zdziebko