Podstawy języka Python

Skrypt do przedmiotu Informatyka 1

Paweł Kłeczek

v0.3
(2018-04-03)

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.

Materiały źródłowe:

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

1.1 Typy danych

1.1.1 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
```

1.1.2 Łańcuchy znaków

Łańcuchy znaków – obiekty typu `str` – definiuje się za pomocą apostrofów (`'...'`) bądź 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')   # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')   # note the r before the quote
C:\some\name
```

1.1.3 Kontrola typów

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

```
v = 3      # v to alias obiektu typu int
v = 'abc'  # 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 odśmiecacz (zob. rozdz. 6.1), natomiast alias `v` zaczyna odnosić się do innego obiektu.

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

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

1.1.4 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 wypisuje się rozdzielone przecinkami elementy:

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

1.1.5 Typy mutowalne i niemutowalne

Każda dana w Pythonie jest obiektem. 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 mutowalnego jest bardziej kosztowna – wymaga bowiem wykonania kopii.

Próba zmiany obiektu niemutowalnego skutkuje błędem (zob. przykł. 1.2). W przypadku obiektów mutowalnych zmiana istniejącego obiektu jest możliwa (zob. przykł. 1.3).

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'
```

Listing 1.3. Obiekty mutowalne można zmieniać.

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

1.1.6 Typy sekwencyjne

Zarówno łańcuchy znaków jak i listy należą do tzw. **typów sekwencyjnych** (ang. sequence types). Wszystkie typy sekwencyjne udostępniają pewne podstawowe operacje.

Tablica 1.1. Operacje właściwe dla typów sekwencyjnych.

Operacja	Wynik
`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`
`s + t`	konkatenacja `s` i `t`
`s * n , n * s`	konkatenacja n spłytkich kopii `s`
`s[i]`	i-ty element `s`
`s[i:j]`	*slice* z `s` od i do j
`s[i:j:k]`	*slice* z `s` od i do j z krokiem k
`len(s)`	długość `s`

1.1.6.1 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:

```
>>> squares[-1]  # ostatni element
25
>>> squares[-2]  # przedostatni element
16
```

Zwróć uwagę, że ponieważ $-0 = 0$, ujemne indeksy zaczynają się od -1.

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
```

1.1.6.2 Slicing

Operacja **slicing** służy do uzyskiwania podciągów:

```
>>> 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 z zakresu $\langle a, b)$. Dzięki temu zawsze spełniona jest zależność `s[:i] + s[i:] == s`.

W wyniku *slicing* tworzona jest nowa (płyta) kopia ciągu (zob. rozdz. 1.6.7).

Możesz pominąć indeksy (jeden z nich bądź oba): domyślnie pierwszy z nich ma wartość 0, a drugi – długość ciągu (zob. przykł. 1.4).

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

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[3:]
[16, 25]
>>> squares[:3]
[1, 4, 9]
>>> squares[:]
[1, 4, 9, 16, 25]
>>> squares[-2:]  # elementy od przedostatniego do ostatniego
[16, 25]
```

Aby zapamiętać jak działa *slicing* warto wyobrazić sobie indeksy jako „wskaźniki" pomiędzy znakami, przy czym „wskaźnik" na lewą stronę pierwszego znaku ma numer 0, a „wskaźnik" na prawą stronę ostatniego znaku w łańcuchu długości n ma numer n, przykładowo:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Pierwszy wiersz liczb zawiera pozycje indeksów $0 \ldots 6$ w łańcuchu znaków, natomiast drugi wiersz zawiera odpowiadające im indeksy ujemne. *Slice* od i do j zawiera wszystkie znaki pomiędzy krawędziami oznaczonymi odpowiednio i i j.

W przeciwieństwie do indeksowania, w przypadków *slicing* dopuszczalne jest użycie indeksu spoza zakresu – zostaną one „przycięte":

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[3:10]
[16, 25]
>>> squares[-8:2]
[1, 4]
```

Istnieje również możliwość dokonywania *slicingu* z zadanym krokiem k – za pomocą `s[a:b:k]`:

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares[1::2]
[4, 16]
```

1.1.6.3 Konkatenacja i powtarzanie

Konkatenacja (łączenie, ang. concatenation) 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 `*`:

```
>>> 2 * '123'
123123
```

1.1.6.4 Długość ciągu

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

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

1.1.6.5 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
```

1.2 Wartość `None`

Specjalna wartość `None` służy do realizacji koncepcji „elementu neutralnego", który może służyć np. do sygnalizowania brakującej wartości:

```
my_none_variable = None
```

Korzysta się z niego szczególnie często podczas definiowania funkcji, jako domyślnej wartości opcjonalnych parametrów.

1.2.1 Wyrażenia logiczne

Wyrażenie logiczne to takie wyrażenie, któremu można przyporządkować jedną z dwóch wartości – *prawda* bądź *fałsz*. Wyrażenia logiczne w Pythonie działają na podobnej zasadzie, co w C/C++, z tym że operatory logiczne mają postać słowną: **`or`**, **`and`**, **`not`** (natomiast operatory porównania są identyczne w obu przypadkach).

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 referencje odnoszą się do tego samego obiektu (to ma znaczenie w przypadku typów mutowalnych)

Do sprawdzania, czy obiekt jest tożsamy z `Null`, korzystaj zawsze z `is`!

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 bądź jako warunek pętli.

1.3 Instrukcje sterujące

Podstawowa funkcjonalność instrukcji sterujących w Pythonie jest podobna do ich odpowiedników w C/C++, lecz Python udostępnia też pewne dodatkowe możliwości.

1.3.1 Instrukcje warunkowe

Instrukcja warunkowa w Pythonie działa analogicznie do instrukcji warunkowej w C/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 C/C++ instrukcji `switch`.

1.3.1.1 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*).

1.3.1.2 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. Oto składnia wyrażenia warunkowego:

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

przykładowo

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

1.3.2 Pętle

1.3.2.1 Instrukcja `while`

Pętla **`while`** działa analogicznie do pętli w C/C++:

```
>>> t = 1
>>> while t < 10:
...     t = t + 1
```

1.3.2.2 Instrukcja `for`

Działanie pętli **`for`** w Pythonie przypomina pętlę *range-based* **`for`** z C++11 – iteracja następuje po elementach ciągu (np. listy):

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

1.3.2.3 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)
...
0
1
2
```

```
range(2, 5)
   2, 3, 4
range(0, 10, 3)
   0, 3, 6, 9
```

1.3.2.4 Instrukcje `break` i `continue`

Instrukcje **`break`** i **`continue`** zachowują się tak, jak w C/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
```

1.4 Instrukcja `pass`

Instrukcja **`pass`** nie robi nic – może być zatem używa w sytuacjach, gdy ze względów syntaktycznych wymagana jest instrukcja, lecz program nie potrzebuje wykonywać żadnego działania (zob. przykł. 1.5).

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

```
# definicja minimalnej klasy
class MyEmptyClass:
  pass

# robocza definicja funkcji
def init():
  pass    # Remember to implement this!
```

1.5 Definiowanie funkcji

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

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

Definicja funkcji rozpoczyna się słowem kluczowym **`def`**, po którym występuje nazwa funkcji, a na końcu – w nawiasach okrągłych – lista parametrów. Ciało funkcji zaczyna się od kolejnej linijki i musi być wcięte. Do dobrych praktyk należy dokumentowanie działania funkcji za pomocą docstringów.

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

W Pythonie 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. Przyjmując poniższą definicję funkcji `ref_demo()`

```
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
x= 42  id= 41903752
>>> id(x)
41902552
```

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

```
def no_return():
    pass

print(no_return())
```

da następujący wynik

```
None
```

1.5.1 Argumenty domyślne

Poniższa funkcja ma dwa parametry, przy czym drugi posiada argument domyślny:

```
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 jest określana *tylko raz* – w momencie definicji funkcji (a nie przy każdym jej wywołaniu), zatem poniższy program

```
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]
```

Jeśli nie chcesz takiego zachowania, możesz zdefiniować funkcję w poniższy sposób:

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

1.5.2 Argumenty słownikowe

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

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

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

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

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

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

Natomiast kolejność samych argumentów słownikowych nie ma znaczenia.

Argumenty słownikowe warto stosować wówczas, gdy poprawiają one czytelność kodu, 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, numtweets=20,
  popular=True)    # czytelne
```

  bądź 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.

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)

1.5.3 Przeciążanie funkcji

W Pythonie nie ma możliwości przeciążenia funkcji – ponowne zdefiniowanie funkcji (tym razem z inną liczbą parametrów) spowoduje *nadpisanie* 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, możemy uzyskać pożądany efekt za pomocą argumentów domyślnych:

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

1.6 Struktury danych

1.6.1 Listy

Aby dodać element na koniec listy, korzystaj z metody `append()`:

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

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']
>>> # zmnień kilka wartości
>>> letters[1:3] = ['B', 'C']
>>> letters
['a', 'B', 'C', 'd']
>>> # usuń kilka wartości
>>> letters[1:3] = []
>>> letters
['a', 'd']
>>> # usuń wszystkie elementy - zastępując je pustą listą
>>> letters[:] = []
>>> letters
[]
```

1.6.2 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 krotki po jej utworzeniu. W przeciwieństwie do listy, krotkę zwykle stosuje się do przechowywania elementów *różnego* typu. Krotkę definiuje się rozdzielając wartości przecinkami, przy czym dla czytelności warto wartości te umieścić w nawiasach okrągłych:

```
>>> 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).

1.6.2.1 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.

Listing 1.6. Tworzenie pustej krotki i krotki jednoelementowej

```
>>> empty = ()    # pusta krotka
>>> len(empty)
0
>>> singleton = ('hello',)     # <-- note trailing comma
>>> len(singleton)
1
>>> singleton
('hello',)
```

Listing 1.7. Tuple unpacking.

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

1.6.3 Zbiory

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, przy czym zbiór pusty definiuje się za pomocą funkcji `set()`[1] (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
{}
```

1.6.4 Słowniki

Python udostępnia wbudowany typ `dict` realizujący funkcjonalność **słownika**[2] (ang. dictionary). 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 za podając w nawiasach klamrowych rozdzieloną przecinkami listę par klucz–wartość (zob. przykł. 1.9)

1.6.5 Struktury zagnieżdżone

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

[1] Zapis `{}` oznacza pusty słownik – typ danych realizujący kontener asocjacyjny.
[2] in. tablicy asocjacyjnej, mapy

Listing 1.9. Tworzenie słowników i operacje na słownikach.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['jack']   # odczytaj wartość skojarzoną z kluczem
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
```

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
```

1.6.6 Modyfikacja struktur podczas iteracji

Jeśli wewnątrz pętli potrzebujesz dokonać modyfikacji ciągu po którym iterujesz (np. chcesz dodać bądź usunąć pewne elementy), warto wcześniej utworzyć kopię takiego ciągu (zob. przykł. 1.11) W przeciwnym

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']
```

razie w pętli z przykładu 1.11 wpadniemy w pętlę nieskończoną, która będzie dodawać wciąż nowe wyrazy `'windows'` do listy, po której następuje iteracja.

1.6.7 Kopiowanie obiektów

W Pythonie zmienne domyślnie zachowują się jak referencje w C++ – mówimy o tzw. **aliasingu** (ang. aliasing), czyli sytuacji, gdy kilka nazw odnosi się do tego samego obiektu. Zmiennej przydzielany jest osobny obszar pamięci dopiero wówczas, gdy przypiszemy do niej nową wartość (zob. przykł. 1.12).

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). 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 nie są one faktycznie kopiowane – *slicing* tworzy tzw. **płytką kopię** (ang. shallow copy) (zob. przykł. 1.15). Rozwiązaniem tego problemu jest wykorzystanie funkcji `deepcopy()` z modułu `copy`, która wykonuje tzw. **głęboką kopię** (ang. deep copy) (zob. przykł. 1.16).

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
                          #    obiektu (nadany przez interpreter Pythona)
9251744 9251744
>>> y = 4   # zmiana obiektu - 'y' odnosi się do nowego obszaru pamięci
>>> print(id(x), id(y))
9251744 9251776
```

Listing 1.13. Aliasing dla typów mutowalnych.

```
>>> colours1 = ["red", "blue"]
>>> colours2 = colours1
>>> print(id(colours1), id(colours2))
14603760 14603760
>>> colours2[1] = "green"
>>> print(id(colours1), id(colours2))
14603760 14603760
>>> print(colours1)
['red', 'green']
>>> print(colours2)
['red', 'green']
```

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

```
>>> list1 = ['a','b','c','d']
>>> list2 = list1[:]
>>> list2[1] = 'x'
>>> print(list2)
['a', 'x', 'c', 'd']
>>> print(list1)
['a', 'b', 'c', 'd']
>>>
```

Listing 1.15. Kopiowanie list zagnieżdżonych z użyciem *slicingu* nie działa.

```
>>> lst1 = ['a','b',['ab','ba']]
>>> lst2 = lst1[:]
>>> lst2[2][1] = 'd'   # zmiana elementu struktury zagnieżdżonej
>>> print(lst1)
['a', 'b', ['ab', 'd']]
>>> print(lst2)
['a', 'b', ['ab', 'd']]
```

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

1.7 Instrukcja `del`

Instrukcja **`del`** może być użyta w dwóch kontekstach – do usunięcia elementów z kolekcji bądź do usunięcia całej zmiennej.

W pierwszym przypadku istnieje możliwość usunięcia m.in. pojedynczego elementu bądź zakresu elementów oraz pary klucz–wartość ze słownika (zob. przykł. 1.17).

W drugim przypadku po prostu podajemy nazwę zmiennej, która ma zostać usunięta (a w zasadzie nazwę aliasu obiektu, który powinien zostać „odpięty"): **`del`** `a`. Późniejsza próba odwołania się do usuniętej zmiennej skutkuje błędem – chyba, że w międzyczasie nadaliśmy jej nową wartość (zob. przykł. 1.18).

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']]
>>> 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
```

Listing 1.17. Usuwanie pojedynczego elementu za pomocą **`del`**.

```
>>> a = [0, 1, 2, 3, 4]
>>> del a[0]
>>> a
[1, 2, 3, 4]
>>> del a[1:3]
>>> a
[1, 4]
>>> del a[:]
>>> a
[]
>>>
>>> d = {'a':1, 'b':2, 'c':3}
>>> del d['b']
>>> d
{'a': 1, 'c': 3}
```

Listing 1.18. Usuwanie nazwy zmiennej za pomocą **`del`**.

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

1.8 Wyrażenia lambda

Słowo kluczowe **`lambda`** służy do definiowania krótkich anonimowych funkcji, tzw. funkcji lambda, składających się z jednej instrukcji (wymóg składni Pythona), np.: **`lambda`** `a, b: a+b`. Funkcje lambda mogą być używane wszędzie tam, gdzie oczekiwany jest obiekt funkcyjny. Podobnie jak zagnieżdżone definicje funkcji, funkcja lambda ma dostęp do zmiennych z zawierającego ją zakresu (zob. przykł. 1.19).

Listing 1.19. Przykład funkcji lambda.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(1)
43
```

1.9 Dobre praktyki

1.9.1 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 również 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

1.9.2 Type hinting

Jedna z podstawowych zasad filozofii Pythona głosi, że „jawne jest lepsze niż niejawne” (ang. *explicit is beter 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, bądź jakiego typu obiektu oczekujemy (zob. przykł. 1.20):

Listing 1.20. Przykład podpowiedzi typu.

```
from typing import List

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

r: bool = is_odd(3)    # podpowiedź typu zmiennej
l: List = []    # podpowiedź typu zmiennej (kolekcja - wymaga importu)
```

Ze względu na występujące w Pythonie dynamiczne typowanie, próba określenia typu zmiennej w sposób automatyczny (przez IDE) kończy się powodzeniem ledwie w ok. 50% przypadków. Określenie typu nastręcza problemów również samym programistom, zwłaszcza gdy przychodzi im analizować cudzy kod.

Korzystanie z podpowiedzi typu daje 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 – brak takich podpowiedzi nie jest błędem.

Materiały źródłowe:

- PEP 484 – Type Hints
- PEP 526 – Syntax for Variable Annotations
- Type Hints – Guido van Rossum (PyCon 2015)

1.10 Formatowanie wyjścia

We współczesnym Pythonie zalecanym sposobem formatowania wyjścia jest skorzystanie z metody `str.format()`.

Materiały źródłowe:

- Fancier Output Formatting (PyDocs)
- `str.format()`
- PEP 3101 – Advanced String Formatting

1.11 Funkcje wyższego rzędu: filter, map, reduce...

Funkcje wyższego rzędu (ang. higher order functions) – czyli funkcje przyjmujące jako parametr inne funkcje – to jeden z kluczowych konceptów programowania funkcyjnego, umożliwiający m.in. zrównoleglanie operacji (zob. model MapReduce).

Zapoznaj się we własnym zakresie z poniższymi materiałami.

Materiały źródłowe:

- Python 3 Tutorial: Lambda, filter, reduce and map (Python Course)
- map, filter, and reduce (BogoToBogo)

Rozdział 2

Klasy

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, pełnią identyczną funkcję jak klasy w C++ – grupują dane i funkcje operujące na tych danych. Co jednak ważne, w Pythonie *wszystko* jest obiektem – wszystko jest instancją jakiejś klasy, każdy obiekt dziedziczy (bezpośrednio bądź pośrednio) po typie `object`[1]. Klasy i typy same w sobie są obiektami – typu `type`. Możesz uzyskać informację o typie danego obiektu z użyciem funkcji `type`:

```
type(any_object)
```

W języku Python dane przechowywane wewnątrz obiektu nazywamy **atrybutami** (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.

2.2 Parametr `self`

Podczas wywoływania metody potrzebujemy w jej ciele dostępu do informacji, na rzecz jakiego obiektu metoda została wywołana – w C++ służy do tego niejawny wskaźnik `this` (nie pojawia się on w nagłówku metody). W języku Python informacja ta przekazywana jest jawnie, jako pierwszy parametr metody[2]. 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).

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 C++ problem kolizji oznaczeń.

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

[1] We wcześniejszych wersjach języka istniał podział na typy wbudowane i klasy zdefiniowane przez użytkownika, lecz obecnie podział ten zanikł.

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

Listing 2.1. Przykład definiowania i korzystania z klasy.

```
import datetime

# definicja klasy
class Person:

  # definicja funkcji inicjalizującej obiekt
  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())
```

Listing 2.2. Przekazywanie argumentu dla parametru `self`.

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

# przekazywanie niejawne - preferowany sposób
print(person.age())

# przekazywanie jawne - dopuszczalne, lecz mniej czytelne
print(Person.age(person))
```

2.3 Metoda `__init__`

Metoda `__init__()` pełni rolę *ciała* konstruktora znanego z języka C++ – służy do *inicjalizacji* składowych utworzonego obiektu.

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[3]. W chwili wywołania metody `__init__()` obiekt został już utworzony, zatem możemy się do niego odwoływać poprzez `self`.

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

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

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. W przy-

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

```
class Account(object):

    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
```

kładzie tym musimy pamiętać, aby wywołać metodę `set_balance()` przed dokonaniem pierwszej wpłaty/wypłaty, aby utworzyć atrybut `self.balance`. Nie mamy jednak żadnej możliwości, aby *wymusić* na użytkownikach naszej klasy stosowania się do takiego rozwiązania, przez co łatwo o błędy wykonania programu (nowo utworzone obiekty klasy `Account` nie będą „w pełni" zainicjalizowane).

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 obiekt znajduje się w poprawnym stanie, stąd zasada inicjalizacji wszystkich atrybutów wewnątrz metody `__init__()`.

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[4], które informują o pożądanym poziomie dostępu:

- nazwy zaczynające się od pojedynczego podkreślenia (`_attr`) odpowiadają polom chronionym (dostępnym tylko w obrębie tej klasy oraz jej klas potomnych), oraz
- nazwy zaczynające się od dwóch podkreśleń (`__attr`) odpowiadają polom prywatnym (dostępnym wyłącznie w obrębie tej klasy).

Atrybut powinien być "prywatny" wyłącznie wtedy, gdy potrzebujemy dokonać weryfikacji bądź transformacji danych.

Z zagadnieniem kontroli dostępu związany jest również dekorator `@property` (zob. rozdz. 2.5.3).

[4] zob. The Python Tutorial: Classes, Private, protected and public in Python (radek.io)

2.5 Atrybuty i metody

W poniższym rozdziale pokazane zostanie wykorzystanie kilku wbudowanych **dekoratorów** – funkcji „opakowujących" inne funkcje i zmieniające ich działanie – podczas definiowania klas. Na chwilę obecną nie będzie Ci potrzebna dokładna znajomość działania dekoratorów, zostaną one omówione w rozdziale 5.4.

2.5.1 Atrybuty klasowe

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

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(object):

    wheels = 4   # atrybut klasowy

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

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

Metody powiązane z instancjami klasy[5] posiadają dostęp do wartości atrybutów klasowych (mogą je odczytywać, ale nie zmieniać!) w sposób identyczny jak do „zwykłych" atrybutów – za pomocą `self` (np. `self.wheels` dla klasy `Car` z przykł. 2.4).

Chcąc zmienić wartość atrybutu klasowego musisz użyć notacji `ClassName.AttributeName` – w przeciwnym razie utworzysz nowy atrybut instancji, który przesłoni atrybut klasowy[6].

2.5.2 Metody klasowe

Metoda klasowa (ang. class method) w języku Python zachowuje w zbliżony sposób do metody statycznej w języku C++ – jest powiązana z klasą (a nie z instancją klasy) i umożliwia wyłącznie dostęp do atrybutów klasowych. 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).

Listing 2.5. Metoda wytwórcza.

```
from datetime import date

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

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

[5] Z metodami klasowymi zapoznasz się w rozdziale 2.5.2.

[6] Przykład: Python 3 Tutorial: Class and Instance Attributes

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 – poprzez instancję klasy bądź poprzez samą klasę – jednak w obu przypadkach pierwszy argument będzie obiektem klasowym. W związku z tym zwyczajowo pierwszy parametr metody klasowej nazywa się `cls` (a nie `self`) – chodzi o podkreślenie, że nie mamy dostępu do atrybutów związanych z instancją poprzez obiekt przekazany jako pierwszy argument (zob. przykł. 2.6)

Listing 2.6. Sposoby wywołania metody klasowej.

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

p1 = Person.fromBirthYear("Adam", 19)   # preferowany sposób
p2 = p.fromBirthYear("Adam", 19)        # dopuszczalny sposób
```

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`).

W wielu językach programowania (np. C++, Java) taka zmiana projektowa powoduje naruszenie interfejsu (API) i wymusza gruntową refaktoryzację kodu:

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

# nowy dostęp
person.name = "John"
person.surname = "Smith"
```

i dlatego języki te (zwłaszcza Java) zachęcają do stosowania prywatnych atrybutów oraz tzw. **getterów** (ang. getters) i **setterów** (ang. setters)[7] – specjalnych metod służących odpowiednio do odczytu i do modyfikacji danych przechowywanych jako składowe. Ponieważ pełnią one rolę pośredników w dostępie do danych (stanowią warstwę abstrakcji), możemy dowolnie zmieniać wewnętrzną reprezentację danych w sposób niezauważalny dla użytkowników klasy – zmiana implementacji nie pociąga za sobą zmiany interfejsu.

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

Język Python udostępnia mechanizm, który pozwala na wygodne przejście od reprezentacji „`fullname`” do „`name` i `surname`” bez konieczności refaktoryzacji – mowa o dekoratorze `@property`. Dekorator `@property` sprawia, że metoda zachowuje się jak dynamiczny atrybut (zob. przykł. 2.8).

Możemy również określić metodę realizującą funkcjonalność settera[8] za pomocą dekoratora `@property_name.setter` (gdzie `property_name` to nazwa symulowanego atrybutu), przy czym nazwa tak udekorowanej metody musi być identyczna z nazwą metody udekorowanej za pomocą `@property` (zob. przykł. 2.9)

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). W języku Python obowiązuje zatem konwencja domyślnego definiowania atrybutów jako publicznych. Atrybuty powinny być definiowane jako prywatne wyłącznie w sytuacji, gdy stanowią wewnętrzną kwestię klasy (ang. class internals) – szczegóły komplementacyjne klasy, do których użytkownik klasy nie powinien mieć dostępu (ani

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

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

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.height += 1  # Jane rośnie o 1 cm
jane.set_height(jane.height + 1)  # Jane rośnie ponownie...
```

Listing 2.8. Dekorator `@property`.

```
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!
```

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 "{self.name} {self.surname}".format(self=self)

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

jane = Person("Jane", "Smith")
print(jane.fullname)

jane.fullname = "Jane Doe"
print(jane.fullname)
print(jane.name)
print(jane.surname)
```

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 (służących wyłącznie „przepychaniu” danych), które tylko zaciemniają rzeczywistą funkcjonalność klasy!

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

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 z klasą, ale nie korzystającą ani z obiektu tej klasy, ani z pozostałej funkcjonalności klasy.

Warto zadać sobie jednak pytanie – jaki miałby być cel takiego „dopinania” luźnej funkcji do danej klasy? Python udostępnia przecież inne metody grupowania funkcjonalności – moduły (zob. rozdz. 3.4)! W praktyce nie zachodzi nigdy potrzeba tworzenia metod statycznych[9].

2.6 Metoda `__del__`

Każda klasa zawiera domyślnie zdefiniowaną metodę `__del__`, wykonywaną podczas niszczenia obiektu. Ponieważ zamiast mechanizmu RAII Python posiada mechanizm tzw. odśmiecania (zob. rozdz. 6.1), w praktyce nie zachodzi konieczność *własnoręcznego* definiowania destruktora. Oznacza to jednak również, że nie mamy gwarancji odnośnie tego kiedy[10] destruktor zostanie wykonany. 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 błędów (zob. rozdz. 4.6).

2.7 Testy jednostkowe

Python posiada wbudowane wsparcie dla testów jednostkowych – w postaci modułu `unittest`. Przykład 2.10 pokazuje sposób definiowania prostego zbioru testów zawierającego dwa przypadki testowe.

Listing 2.10. Definiowanie testów jednostkowych.

```
# import modułu testów jednostkowych
import unittest

# zbiór testów dziedziczy po klasie 'unittest.TestCase'
class TestStringMethods(unittest.TestCase):

    def test_upper(self):
      # metody testowe wywoływane są dla obiektu 'self'
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

# uruchom testy, jeśli moduł wykonywany jest jako osobny program
if __name__ == '__main__':
    unittest.main()
```

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

[10] 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).

Testy zwykło definiować się w osobnych plikach, zwykle w odpowiednio nazwanym katalogu (np. `test/`).

Materiały źródłowe[11]:

- `unittest` – Unit testing framework (PyDocs)
- Python 3 Object Oriented Programming (2010) [Phillips], Chp. 11
- Expert Python Programming (2008) [Ziade], Chp. 11
- Pro Python (2010) [Alchin]
- Python 3 Tutorial: Python Tests

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

Rozdział 3

Programowanie zorientowane na obiekty

W niniejszym rozdziale poznasz zasady i mechanizmy służące realizacji koncepcji programowania zorientowanego na obiekty.

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 pomiędzy klasami

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

3.1.1 Kompozycja

Dla przypomnienia: Relacja kompozycji to najprostszy rodzaj powiązania między dwiema klasami – polega po prostu na dodaniu do klasy `B` składowej będącej typu klasowego `A` (zob. przykł. 3.1). Oznacza to, że obiekt typu `B` nie może istnieć w oderwaniu od obiektu typu `A`, 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
```

3.1.2 Dziedziczenie

Dla przypomnienia: 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).

3.1.2.1 Przesłanianie metod

Mechanizm przesłaniania metod sprawia, że możemy w klasie potomnej zmienić implementację metody zdefiniowanej uprzednio w klasie macierzystej. Wciąż mamy możliwość odwołania się do metody zdefiniowanej w klasie macierzystej z użyciem notacji `ClassName.method_name(obj)` (zob. przykł. 3.3).

Listing 3.2. Przykład relacji dziedziczenia. Użycie funkcji `super()` oraz parametru `**kwargs` zostało opisane w rozdziałach 3.2.1.2 i 3.2.1.3.

```
class DerivedClassName(BaseClassName):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)    # dobra praktyka: inicjalizacja
                                      #   klasy macierzystej (BaseClassName)
```

Listing 3.3. Przykład przesłaniania metody.

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

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

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

Z mechanizmu przesłaniania korzystaliśmy m.in. podczas definiowania metody `__init__()` (zob. przykład 3.2) – ten przykład dotyczy każdej klasy, gdyż każda klasa niejawnie dziedziczy po klasie `object`[1].

Niektórzy błędnie stosują termin „*overwriting*" zamiast poprawnego „*overriding*" – choć chodzi im o ten sam koncept 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. 3.4).

Listing 3.4. Przykład relacji dziedziczenia wielokrotnego.

```
class A:
    pass

class B:
    pass

# klasa C dziedziczy po A oraz 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 – 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.

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

Przykłady zaczerpnięto z Python 3 Tutorial: Multiple Inheritance

3.2.1.1 Problem diamentu #1: „Przypadkowa" kolejność wywołań metod

W przykładzie 3.5 klasa `B` nadpisuje metodę `m()` z klasy `A`. W przypadku wywołania metody `m` dla instancji `x` klasy `D` (`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 3.5. 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 (zob. rozdz. 6.4).

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'
```

3.2.1.2 Problem diamentu #2: Wielokrotne wywoływanie metod z klasy macierzystej

Twórca kodu z przykładu 3.6 chciał uzyskać następujący efekt: metoda `m()` z klasy `D` powinna wywoływać metody `m()` z *wszystkich* klas macierzystych.

Na pierwszy rzut oka implementacja wygląda rozsądnie, lecz problem pojawia się w sytuacji, gdy klasy `B` i `C` nadpisują metodę `m()` z klasy `A`, jednak każda z metod `m()` w klasach `B` i `C` korzysta z metody `m()` z klasy macierzystej (zob. przykł. 3.7). W przypadku wywołania metody `m()` dla instancji klasy `D` metoda `m()` z klasy `A` zostanie wywołana aż trzykrotnie!

Optymalne rozwiązanie polega na stosowaniu wbudowanej funkcji `super()`[2] (zob. przykł. 3.8). W przykładzie 3.8 zapewni ona wywołanie wszystkich metod `m()`, we właściwej kolejności, zdefiniowanej przez reguły MRO (zob. rozdz. 6.4), a także wywołanie metody ze wspólnej klasy macierzystej tylko raz.

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

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):
    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)
```

Listing 3.7. 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)

# ------

>>> x = D()
>>> x.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
```

3.2.1.3 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

Listing 3.8. 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()
```

MRO sprawia, że powinniśmy przekazać dalej nie tylko argumenty oczekiwane przez metodę bezpośredniego rodzica, ale *wszystkie* niewykorzystane argumenty.

Aby zapewnić takie przekazywanie, powszechną konwencją jest dodawanie parametrów właściwych dla danej podklasy na początku listy parametrów, a zdefiniowanie wszystkich pozostałych za pomocą argumentów pozycyjnych (`*args`) oraz słownikowych (`**kwargs`) – dzięki temu klasa potomna nie musi znać żadnych szczegółów związanych z parametrami metody `__init__()` klasy macierzystej (zob. przykł. 3.9).

Listing 3.9. 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)
```

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

W metodzie `__init__()` należy *zawsze* wywoływać `super().__init__()` (nawet, gdy klasa dziedziczy tylko po `object`) – tylko wtedy mamy gwarancję, że wszystkie fragmenty klasy zostaną poprawnie zainicjalizowane (zob. też rozdz. 3.2.1.3).

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)

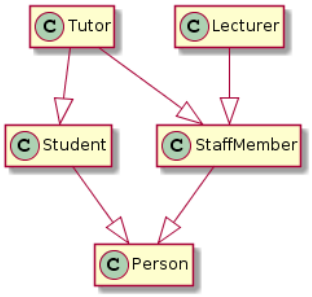
Materiały źródłowe:

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

3.2.2 Mix-iny

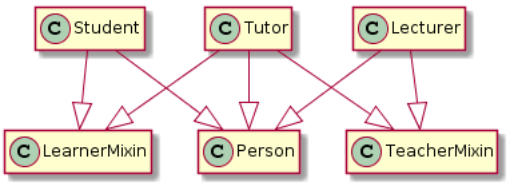
Rozważ następujący problem: Tworzysz system do zarządzania uczelnią – w szczególności studentami i prowadzącymi zajęcia. Zarówno studenci, jak i prowadzący, to osoby. Co więcej, student może być jednocześnie prowadzącym (np. student studiów doktoranckich).

W takim przypadku zamiast tworzyć mało przejrzystą pionową hierarchię dziedziczenia (Rys. 3.1), lepiej podzielić funkcjonalność na mniejsze „bloczki" – 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 3.10 – 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 problem diamentu.

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[3] 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. 3.11).

Jak widzisz, pisanie mix-inów jest proste, gdyż Python udostępnia mechanizm sprawdzania istnienia składowych obiektu niezależnie od jego typu oraz jest dynamicznie typowany (zob. rozdz. **??**, 3.2.2.1).

Po utworzeniu innego mix-inu, `ToDictMixin`, udostępniającego metodę `to_dict()`, możemy efektywnie tworzyć klasy wymagające (de)serializacji – dziedzicząc po obu mix-inach (zob. przykł. 3.12).

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

Listing 3.10. Mix-iny `LearnerMixin` oraz `TeacherMixin`.

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


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

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


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

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

Listing 3.11. 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"
```

Listing 3.12. Tworzenie klas korzystających z wielu mix-inów.

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

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

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

- Czy wydzielona funkcjonalność zostanie *od razu* zastosowana w różnych niepowiązanych ze sobą klasach? – Być może warto zastosować min-in.
  Jeśli wydzielona funkcjonalność ma być użyta tylko w jednej klasie – nie wydzielaj jej.
- Czy potrafisz streścić wydzieloną funkcjonalność za pomocą przymiotnika (a nie rzeczownika czy czasownika), np. *Movable*? – Jeśli nie, to zapewne nie chodzi Ci o utworzenie mix-ina.
- Czy wydzielona funkcjonalność stanowi udoskonalenie bądź specjalizację istniejącego typu? – Być może lepiej zastosować „zwykłe” dziedziczenie.

- Masz problem ze skutecznym wydzieleniem mix-ina? – 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 – w przypadku kompozycji ryzyko wystąpienia takich problemów jest znacznie mniejsze.

Unikaj stosowania 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, że często istnieją lepsze alternatywy dla mix-inów:

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

Przykładami mix-inów z biblioteki standardowej są klasy `ForkingMixIn` oraz `ThreadingMixIn` w module `socketserver` (zob. `socketserver` – A framework for network servers).

3.2.2.1 `getattr`, `setattr` i `hasattr`

Podczas pisania mix-inów przydatne są funkcje wbudowane `hasattr()`, `getattr()` oraz `setattr()` – zob. `getattr`, `setattr` and `hasattr` (OOPinP).

3.2.3 Klasy abstrakcyjne i interfejsy

Język Python nie wspiera bezpośrednio[4] tworzenia klas abstrakcyjnych – możemy jednak w tym celu skorzystać z funkcjonalności udostępnianych przez standardowy moduł `abc`[5]. W tym celu definiowana klasa abstrakcyjna powinna dziedziczyć po klasie `ABC` oraz posiadać choć jedna metodę udekorowaną jako `@abstractmethod`.

Przykład 3.13 zawiera definicję klasy abstrakcyjnej `Shape` będącej szablonem dla kształtów (dwuwymiarowych figur). Próba utworzenia obiektu takiej klasy skutkuje błędem:

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

Listing 3.13. Abstrakcyjna klasa `Shape`.

```
from abc import ABC, abstractmethod

class Shape(ABC):

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

Przykład 3.14 zawiera definicję klasy `Square` dziedziczącej po `Shape` i implementującej wszystkie metody abstrakcyjne, czyli **klasy konkretnej** (ang. concrete class). Próba utworzenia obiektu klasy `Square` kończy się powodzeniem:

```
>>> s = Square(2)
>>>
```

Listing 3.14. 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 dodać odpowiedni własnoręcznie zdefiniowany dekorator do metod „nadpisywanych" w klasach potomnych (zob. Stack Overflow).

3.2.4 Zastępowanie dziedziczenia kompozycją

Ponieważ Python jest językiem typowanym dynamicznie (a nie statycznie, jak np. C++), polimorfizm można osiągnąć bez konieczności stosowania dziedziczenia – wystarczy, że dany obiekt posiada wszystkie niezbędne atrybuty i/lub metody (zob. duck typing, rozdz. 3.3).

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

W Pythonie kompozycja jest bardziej elegancka i naturalna niż dziedziczenie.

Przykład 3.15 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)
```

Możemy uniknąć wspomnianych problemów nadpisując w klasie `SecurityDoor` metodę specjalną `__getattr__()`, wywoływaną gdy żądana składowa nie została znaleziona w obiekcie (zob. przykł. 3.16).

3.3 Duck Typing

Mechanizm „duck typing" służy do realizacji zasady:
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 3.17 demonstruje użycie duck typingu w Pythonie[6].

3.4 Programowanie modułowe

Już na wiele lat przed pojawieniem się komputerów inżynierowie zauważyli, że łatwiej tworzyć złożone systemy z mniejszych, funkcjonalnych elementów – modułów – które można rozwijać niezależnie od siebie i z których można korzystać w różnych systemach. Wyróżniki systemu modułowego to: 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

[4] Inaczej niż w C++, gdzie taki mechanizm udostępnia nam kompilator.
[5] ABC to akronim od ang. Abstract Base Class – abstrakcyjna klasa macierzysta.
[6] W C++ przykładem duck typingu są szablony.

Listing 3.15. Relacja kompozycji.

```
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()
```

Listing 3.16. 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()
>>>
```

„wyższa warstwa" łączy je ze sobą – przykładowo w komputerze rolę takiej „wyższej warstwy" pełni płyta główna (zapewnia komunikację między podzespołami).

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

Listing 3.17. 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()

# -----

>>> 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'
```

3.4.1 Czym jest moduł w Pythonie?

W Pythonie moduł służy do grupowania elementów realizujących powiązaną (np. tematycznie) funkcjonalność, co pozwala uniknąć eksplozji nazw w globalnej przestrzeni nazw. Elementami tymi mogą być dowolne obiekty (klasy, funkcje, atrybuty globalne itd.), zatem moduł w Pythonie stanowi odpowiednik przestrzeni nazw z języka C++.

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

3.4.2 Importowanie modułów

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

```
import math  # zaimportuj moduł 'math'
```

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

Należący do biblioteki standardowej moduł `math` udostępnia różne stałe oraz funkcje matematyczne, np. π (`math.pi`) oraz sinus (`math.sin()`). Po zaimportowaniu tego modułu (w powyższy sposób) możemy korzystać z jego funkcjonalności za pomocą kwalifikowanej nazwy:

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

Gdy potrzebujemy skorzystać tylko z kilku wybranych funkcjonalności danego modułu oraz sądzimy, że nazwy takich funkcjonalności nie wejdą w konflikt z innymi istniejącymi w programie nazwami, możemy zaimportować te obiekty bezpośrednio:

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

a następnie korzystać z nich już bez kwalifikacji:

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

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

```
from math import *
```

jednak z tej opcji powinno się korzystać wyłącznie podczas pracy w interaktywnej konsoli Pythona.

Bezpośrednie importowanie wszystkich obiektów z modułu 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 zapisu `from ...import *` dla wygody, aby uniknąć pisania długich kwalifikowanych nazw. Python pozwala jednak rozwiązać ten problem w inny sposób – poprzez przemianowywanie przestrzeni nazw podczas importowania modułu. 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
```

3.4.3 Projektowanie i kodowanie modułów

Utwórzmy 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):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

def fibi(n):
    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 bądź skryptu:

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

3.4.4 Paczki

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

Paczką jest każdy znajdujący się w ścieżce Pythona katalog zawierający plik `__init__.py`, wykonywany podczas importowania paczki. Plik `__init__.py` zawiera instrukcje służące konfiguracji paczki – ustawieniu wartości, zaimportowaniu innych paczek i modułów – natomiast może też być pusty.

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

3.4.4.1 Prosty przykład

Aby utworzyć przykładową paczkę `simple_package` zawierającą moduły `a` oraz `b`, musimy utworzyć katalog `simple_package` zawierający pliki `__init__.py`, `a.py` oraz `b.py`.
Zawartość pliku `a.py`:

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

Zawartość pliku `b.py`:

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

Sprawdźmy działanie naszej paczki w interaktywnej konsoli Pythona:

```
>>> 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, sama paczka `simple_package` została zaimportowana, jednak ani moduł `a`, ani moduł `b`, nie zostały załadowane! Aby nie było konieczności ręcznego importowania każdego z modułów[7] 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
```

[7] `from simple_package import a, 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 4

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).

Materiały źródłowe:

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

4.1 Błędy składniowe

Błędy składniowe, in. **błędy parsowania** (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.

4.2 Wyjątki

Nawet wyrażenie poprawne syntaktycznie 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

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 C++ – w Pythonie wyjątki są stosowane powszechnie. Wynika to z filozofii Pythona, zgodnie z którą „lepiej prosić o przebaczenie, niż o pozwolenie".

4.3 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 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 bądź jego nadklasę (np. **except** `BaseException` wychwyci w szczególności wszystkie wbudowane wyjątki).
  - Jeśli odpowiednia klauzula **except** zostanie znaleziona, zostanie ona wykonana, a następnie wykonanie programu będzie kontynuowane po instrukcji **try** (dalsze klauzule **except** nie są 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. 4.1). 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 Pythonie 2 składnia **except** `Exception, e` służyła do powiązania wyjątku z opcjonalnym parametrem – w tym przypadku `e` – który pozwalał na późniejszą dalszą inspekcję wyjątku.

Listing 4.1. 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()
```

4.4 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 bądź klasę wyjątku (w tym drugim przypadku zostanie domyślnie wywołany konstruktor bezargumentowy):

```
raise ValueError('X')    # obiekt wyjątku
raise ValueError    # równoważne 'raise ValueError()'
```

4.5 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 (bądź pośrednio) przynajmniej po standardowej klasie `Exception` (zob. rozdz. 4.9):

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

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

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

```
class ShoeError(Exception):
    """Basic exception for errors raised by shoes"""

class UntiedShoelace(ShoeError):
    """You could fall"""

class WrongFoot(ShoeError):
    """When you try to wear your left show on your right foot"""
```

Jeśli to ma sens, twoje klasy błędów powinny również dziedziczyć po standardowych typach wyjątków (zob. rozdz. 4.9), przykładowo:

```
class CarError(Exception):
    """Basic exception for errors raised by cars"""

class InvalidColor(CarError, ValueError):
    """Raised when the color for a car is invalid"""
```

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. 4.2). W przypadku definiowania własnej metody `__init__()` należy wywołać metodę `__init__()` klasy macierzystej. W nadrzędnej klasie (w przykładzie 4.2: w klasie `CarError`) należy wywołać ją z jednym argumentem – wartością, która zostanie wypisana przez `BaseException.__str__()` jako komunikat diagnostyczny.

Listing 4.2. Definiowanie nowego wyjątku we własnym module (własnej bibliotece).

```
class CarError(Exception):
    """Basic exception for errors raised by cars"""
    def __init__(self, car, msg=None):
        if msg is None:
            # Ustaw domyślny użyteczny (!) komunikat
            msg = "An error occured with car %s" % car
        super().__init__(msg)    # wywołanie konstruktora klasy 'Exception'
        self.car = car

class CarCrashError(CarError):
    """When you drive too fast"""
    def __init__(self, car, other_car, speed):
        super().__init__(
            car, msg="Car crashed into %s at speed %d" % (other_car, speed))
        self.speed = speed
        self.other_car = other_car
```

4.6 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 4.3 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 4.3. (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 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. 4.4).

Listing 4.4. Przykład korzystania z zasobów systemowych z użyciem **`with`**.

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

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 wchodzenia do kontekstu i podczas opuszczania kontekstu – w przykładzie 4.4 menedżer kontekstu powiązany z funkcją `open()` odpowiednio otwiera i zamyka zadany plik. Istnieje możliwość definiowania własnych menedżerów kontekstu.

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)

4.7 Rzucaj wyjątki zamiast zwracać `None`

Ponieważ Python udostępnia specjalną wartość `None`, początkującym programistom Pythona 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
```

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 `None` zdecydowanie lepszym rozwiązaniem (czytelnym i pozwalającym zmniejszyć ryzyko błędów) jest rzucenie odpowiedniego wyjątku – w ten sposób użytkownik, który zapomni obsłużyć wyjątek, zostanie o tym niezwłocznie poinformowany. Przykład 4.5 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).

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

Listing 4.5. 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:
        raise ValueError('Divisor cannot be zero.') from e

# ----

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

4.8 Luźne uwagi

- Wyjątki służą do sygnalizowania sytuacji, które nie są częścią normalnego (oczekiwanego) przebiegu wykonania programu.
  Przykładowo, funkcja `str.find()` zwraca `-1`, gdy wzorzec nie zostanie znaleziony w łańcuchu znaków (gdyż to normalna sytuacja), lecz 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`...) zwykle najlepiej zostawić programiście/użytkownikowi, gdyż próba ich „obsługi” jedynie ukryje faktyczną przyczynę problemów.

4.9 Hierarchia wbudowanych wyjątków

Built-in Exceptions: Exception hierarchy (PyDocs)

Rozdział 5

Be pythonic!

Można programować w języku Python tkwiąc mentalnie w C/C++. Można przestawić się na sposób myślenia programisty Pythona, 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 Pythona.

5.1 Zen of Python

Prawdopodobnie najlepsze zestawienie zasad filozofii Pythona zostało stworzone przez Tima Petersa, wieloletniego współtwórcy 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* (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 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.

5.1.1 Beautiful Is Better Than Ugly

Piękno to dość subiektywne kryterium, jednak spoglądając na przykład 5.1, zawierający warianty metod służących do walidacji danych formularza, możemy z całą pewnością stwierdzić, że pierwsza propozycja

[1] zob. PEP 20 – The Zen of Python

jest „piękniejsza” od drugiej – nazwa funkcji `is_valid()` od razu sugeruje typ zwracanej wartości, natomiast nazwa `validate()` nawet nie sugeruje zwracania wartości.

Listing 5.1. „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)
```

5.1.2 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 (nie trzeba szukać w dokumentacji) – zob. przykł. 5.2

Listing 5.2. 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'?!
```

5.1.3 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` bądź `True`) bez konieczności jawnego testowania (zob. przykł. 5.3).

Listing 5.3. Złożoność rozwiązania – na przykładzie instrukcji warunkowej.

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

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

5.1.4 Complex Is Better Than Complicated

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

- rozwiązanie złożone (complex) – składa się z wielu wzajemnie powiązanych elementów,
- rozwiązanie skomplikowane (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).

5.1.5 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 5.4 zawiera zagnieżdżoną, nieczytelną strukturę. Można ją „spłaszczyć”, jak w przykładzie 5.5, co pozwala łatwiej zorientować się w przepływie sterowania.

Listing 5.4. 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 5.5. Płaska (przejrzysta) struktura instrukcji warunkowych, równoważna strukturze z przykładu 5.4.

```
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.")
```

5.1.6 Sparse Is Better Than Dense

Należy formatować kod źródłowy w taki sposób, aby był czytelny[2] – z jednej strony nie ma sensu na siłę zagęszczać elementów (zob. przykł. 5.6), a z drugiej strony nie należy zbyt hojnie umieszczać odstępów (zob. przykł. 5.7).

Listing 5.6. Zbytnie zagęszczenie kodu może zmniejszać jego czytelność.

```
if x == 4: print('OK')  # źle - zbytnie zagęszczenie

if x == 4:
    print('OK')  # dobrze
```

Listing 5.7. Zbytnie rozstrzelenie kodu może zmniejszać jego czytelność.

```
spam(ham[1], {eggs: 2})  # OK
spam( ham[ 1 ], { eggs: 2 } )  # źle - za dużo odstępów
```

5.1.7 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).

[2] zob. PEP 8 – Style Guide for Python Code

Zagadnieniu czytelności kodu poświęcony jest PEP 8 – Style Guide for Python Code.

5.1.8 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 `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.

5.1.9 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).

5.1.10 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 Pythona chodzi również o iteracyjne podejście do rozwoju oprogramowania – Python umożliwia prototypowanie, dzięki czemu nie musimy mieć od pierwszej chwili gruntownie przemyślanego całego rozwiązania (ewentualne refaktoryzacje nie są zwykle kosztowne).

5.1.11 If the Implementation is Hard to Explain, It's a Bad Idea

Oczywiście – „moja racja jest racja najmojsza" – jednak zwykle lepiej skonsultować swój pomysł z inną osobą. Jeśli wytłumaczenie sposobu rozwiązania zajmuje dużo czasu i wysiłku intelektualnego, to jest to najprawdopodobniej złe rozwiązanie.

5.1.12 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 poprawne! Zwykle dopiero rzetelna ocena przez innych programistów pozwala na wypracowanie optymalnego rozwiązania.

5.2 Comprehensions

Konstrukcja **comprehension** to pythoniczny sposób implementacji stosowanego przez matematyków sposobu definiowania zbiorów i ciągów. Przykładowo, w matematyce zbiór kwadratów liczb naturalnych od 0 do 9, S, definiowany jest następująco:

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

Oto sposób na zdefiniowanie zbioru S w Pythonie z użyciem **set comprehension**:

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

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

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

w Pythonie użyjemy następującej instrukcji:

```
M = {x for x in S if x % 2 == 0}
```

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

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

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

Analogicznie możemy użyć **dict comprehension**, przykładowo:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

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)

5.3 Generatory

Materiały źródłowe:

- Python 3 Tutorial: Generators (Python Course)
- Tutorial – Python List Comprehension With Examples (by Aarshay Jain)

5.4 Dekoratory

W Pythonie **dekoratorem** (ang. decorator) jest każdy dający się wywołać obiekt, który służy do zmiany zachowania funkcji bądź klasy – referencja do (oryginalnej) funkcji `func` bądź 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 Pythona pozwalające zdefiniować i użyć prosty dekorator. Szczegóły dotyczące sposobu implementowania dekoratorów znajdziesz w dokumencie PEP 318 – Decorators for Functions and Methods.

Materiały źródłowe:

- Python 3 Tutorial: Decorators (Python Course)

5.4.1 Aliasing

Pamiętaj, że w Pythonie nazwy funkcji to w rzeczywistości referencje do obiektów funkcyjnych, oraz że możemy nadać obiektowi funkcyjnemu kilka nazw:

```
def succ(x):
  return x + 1

successor = succ  # 'successor' i 'succ' odnoszą się do tego
                  # samego obiektu funkcyjnego
```

5.4.2 Zagnieżdżanie funkcji

W Pythonie mamy możliwość definiowania **funkcji zagnieżdżonej** (ang. nested function), czyli funkcji zdefiniowanej wewnątrz innej funkcji (zob. przykł. 5.8, 5.9).

Listing 5.8. Funkcja `g()` jest zagnieżdżona w funkcji `f()`.

```
def f():
    def g():
        print("This is the function 'g'")

    print("This is the function 'f'")
    g()

# ---

>>> f()
This is the function 'f'
This is the function 'g'
```

Listing 5.9. Zagnieżdżenie funkcji zwracających wartość.

```
def temperature(t):
    def celsius2fahrenheit(x):
        return 9 * x / 5 + 32

    result = "It's " + str(celsius2fahrenheit(t)) + " degrees!"
    return result

# ---

>>> print(temperature(20))
It's 68.0 degrees!
```

5.4.3 Funkcje jako parametry

Ponieważ każdy parametr funkcji jest referencją na obiekt, a funkcje też są obiektami, możemy przekazywać funkcje (a w zasadzie: referencje do funkcji) jako argumenty funkcji (zob. przykł. 5.10).

Listing 5.10. Przekazywanie funkcji jako argumentu innej funkcji.

```
import math

def foo(func):
    return sum([func(x) for x in [1, 2, 2.5]])

# ---

>>> print(foo(math.sin))
2.3492405557375347
>>> print(foo(math.cos))
-0.6769881462259364
```

5.4.4 Funkcje zwracające funkcje

Ponieważ wartość zwracana przez funkcję to referencja do obiektu, funkcje mogą zwracać referencje do innych obiektów funkcyjnych (zob. przykł. 5.11).

Listing 5.11. „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

# ---

>>> 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
```

5.4.5 Prosty dekorator

Mamy obecnie opanowane wszystkie niezbędne podstawy, aby zdefiniować nasz pierwszy prosty dekorator (zob. przykł. 5.12).

Listing 5.12. Definicja prostego dekoratora.

```
def our_decorator(func):
    def function_wrapper(x):
      # obiekty funkcyjne zawierają atrybut '__name__'
      #   przechowujący identyfikator funkcji
        print("Before calling " + func.__name__)
        print(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
None
After calling foo
```

Zwróć uwagę, że po zastosowaniu dekoracji (`foo = our_decorator(foo)`), `foo` staje się referencją do obiektu funkcyjnego `function_wrapper`. Obiekt funkcyjny `foo` wciąż będzie wywołany wewnątrz `function_wrapper`, lecz przed i po tym wywołaniu zostanie wykonany dodatkowy kod (w tym przypadku: wypisane zostaną informacje diagnostyczne).

5.4.6 Typowa składnia dekoratorów w Pythonie

W Pythonie zwykle unikamy dekorowania za pomocą przypisania (por. `foo = our_decorator(foo)`), gdyż w ten sposób 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):
    ...
```

Za pomocą dekoratora `our_decorator` możemy udekorować każdą inną *jednoparametrową* funkcję:

```
@our_decorator
def succ(n):
    return n + 1

succ(10)
```

Rezultat wykonania powyższego programu:

```
Before calling succ
11
After calling succ
```

5.4.7 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 odpowiednim wyjątkiem. Aby uniknąć zaciemniania kodu takich funkcji matematycznych wywołaniem pomocniczej funkcji weryfikującej, możemy napisać i zastosować odpowiedni dekorator (zob. przykł. 5.13).

Listing 5.13. 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("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)
```

5.5 Docstrings

Materiały źródłowe:

- PEP 257 – Docstring Conventions
- Google vs NumPy style

5.6 Idiomatyczny Python

Oto znany cytat dotyczący pisania kodu, który da się utrzymywać:

```
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 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).

5.6.1 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 każdego z poniższych wyrażeń to `False`:

- `None`
- `False`
- zero dla typów numerycznych
- puste kontenery (tj. `[]`, `{}`, and `()`)
- wartość 0 bądź `False` zwrócona przez wywołanie `__len__` bądź `__nonzero__`

natomiast wszelkie inne wyrażenia są traktowane jako `True` (zatem „prawdziwość" w większości przypadków to `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 `True`).

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`).

5.6.2 Stosuj słowo kluczowe `in` do iterowania po `iterable`

Pamiętaj, że w Pythonie do iterowania po zakresie (tzw. styl `for_each`) służy słowo kluczowe `in` – a *nie* zmienna indeksująca (por. przykł. 5.14 i 5.15).

Listing 5.14. 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 5.15. Iterowanie za pomocą `in` – **idiomatyczne**.

```
my_list = ['Larry', 'Moe', 'Curly']
for element in my_list:
    print (element)
```

5.6.3 Unikaj potarzania 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. 5.16 i 5.17). Dzięki temu unikniesz zaciemniania kodu zbędnymi powtórzeniami nazwy zmiennej.

Listing 5.16. Powtarzanie nazwy zmiennej w złożonym warunku – **szkodliwe**.

```
name = 'Tom'
if name == 'Tom' or name == 'Dick' or name == 'Harry':
    pass
```

Listing 5.17. Korzystanie z `iterable` oraz z `in` – **idiomatyczne**.

```
name = 'Tom'
if name in ('Tom', 'Dick', 'Harry'):
    pass
```

5.6.4 Unikaj umieszczania kodu rozgałęzienia w linii z dwukropkiem

Aby zwiększyć czytelność kodu, umieszczaj instrukcje do wykonania w ramach danego rozgałęzienia w osobnych linijkach – z użyciem wcięcia. Symbol `:` powinien być ostatnim znakiem w danej linii (por. przykł. 5.18 i 5.19).

Listing 5.18. Kod rozgałęzienia w linii z dwukropkiem – **szkodliwe**.

```
name = 'Jeff'
address = 'New York, NY'
if name: print(name)
print(address)
```

Listing 5.19. Kod rozgałęzienia w osobnej linii, z wcięciem – **idiomatyczne**.

```
name = 'Jeff'
address = 'New York, NY'
if name:
    print(name)
print(address)
```

5.6.5 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 C++:

```
for (int i = 0; i < container.size(); ++i) {
    // Do stuff
}
```

W Pythonie służy do tego wbudowana funkcja `enumerate` (por. przykł. 5.20 i 5.21).

Listing 5.20. 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 5.21. Uzyskiwanie informacji o indeksie za pomocą `enumerate` – **idiomatyczne**.

```
my_container = ['Larry', 'Moe', 'Curly']
for index, element in enumerate(my_container):
    print ('{} {}'.format(index, element))
```

5.6.6 Stosuj klauzulę `else` do pętli

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`** bądź **`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. 5.22 i 5.23).

Listing 5.22. 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 5.23. 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!')
```

5.7 Iterowanie po strukturach danych

W przypadku intensywnej pracy ze strukturami danych warto znać techniki efektywnego iterowania po nich – poniższe linki 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 Pythonem 2 a Pythonem 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 6

Zagadneinia 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)

6.1 Garbage collector

Materiały źródłowe:

- Things you need to know about garbage collection in Python (by Artem Golubin)

6.2 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, 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 przestrzeni nazw w 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 Pythona 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 Pythonie (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:

- zasięg najgłębszy – przeszukiwany w pierwszej kolejności – zawiera nazwy lokalne

[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.

- 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” Pythona 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 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
```

[2] tj. funkcji znajdujących się w danym momencie na stosie wywołań

Z kolei wykonanie poniższego programu, również ilustrującego zachowanie się zasięgów:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    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: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

6.3 Python Search Path

Materiały źródłowe:

- How does python find packages? (by Lee Mendelowitz)

6.4 Method Resolution Order

W jaki sposób funkcja `super()` określa, 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 'su
>>> 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(object):
    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`.

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] Python wspiera wielokrotne dziedziczenie.

Podziękowania

Korekta

Michał Krzyszczuk

Dominika Przewłocka