Python - notatki do wykładu

z przykładami w archiwum ***python_examples***

Tomasz R. Werner

Warszawa, 28 maja 2024 22:24

Spis treści

Rozdział 1

Wprowadzenie

1.1 Historia

Python został stworzone przez niderlandzkiego informatyka Guido van Rossum[1] w późnych latach 80-tych w Niderlandach. W tym czasie pracował on nad projektem zwanym Amoeba w CWI (Centrum voor Wiskunde en Informatica) w Amsterdamie. Celem była budowa rozproszonego systemu operacyjnego, a częścią projektu było opracowanie języka programowanie ogólnego przeznaczenia ABC przez zespół do którego van Rossum należał. Sam van Rossum nie był zadowolony z tego języka i w czasie urlopu noworocznego w grudniu 1989 roku rozpoczął pracę nad nowym językiem, opartym koncepcyjnie na ABC, ale pozbawionym jego wad.

Guido van Rossum

Pierwsza wersja Pythona (wersja 0.9.0) została opublikowana w lutym 1991 roku. Implementowała już klasy z dziedziczeniem, wyjątki, funkcje i podstawowe typy danych (liczbowe, listy, słowniki, napisy). Wiele elementów, na przykład podział na moduły, zostało zainspirowanych językiem Modula-3.[2]

W 1994 roku powstała grupa dyskusyjna ***comp.lang.python*** i znacznie przyspieszyła rozwój języka i jego popularność.

Python 2 został opublikowany 16 października 2000 i jego ostatnią wersją jest 2.7. Nie będzie ona kontynuowana gdyż została zastąpiona przez wersję trzecią, Python 3, opublikowaną 3 grudnia 2008. Python 3 zawiera nowe elementy, które czynią go *nie* w pełni kompatybilnym z Pythonem 2 (dystrybucja Pythona zawiera program ***2to3*** automatyzujący przetwarzanie kodu Pythona 2 na kod Pythona 3).

Sam Guido van Rossum pełnił rolę lidera rozwoju Pythona jako „dożywotni dyktator" („benevolent dictator for life") aż do 12 lipca 2018, kiedy to ogłosił „permanentne wakacje" od swoich obowiązków „dyktatora". Kierowanie całym projektem przejęła pięcioosobowa Rada Kierująca (Steering Council) wybierana przez fundację non-profit Python Software Foundation.

Sama nazwa języka Python pochodzi z komediowego serialu telewizji BBC *Monty Python's Flying Circus* (Latający Cyrk Monty Pythona).

1.2 Ogólne uwagi o języku Python

Python jest językiem obiektowo-zorientowanym: wszystko jest obiektem, również liczby całkowite i zmiennoprzecinkowe. Nawet klasy i moduły są zaimplementowane jako obiekty!

Python jest też językiem **silnie typowanym** (*strongly typed*) co oznacza, że każdy obiekt ma ściśle określony typ. Jednocześnie jest językiem **dynamicznie typowanym** (*dynamically typed*), czyli sprawdzanie typów zachodzi dopiero w czasie wykonania (Java czy C/C++ są *statycznie* typowane (*statically typed*)— typy wszystkich obiektów muszą być znane w czasie kompilacji i nie mogą się zmieniać).

[1] Prawidłową wymowę imienia i nazwiska można usłyszeć tu https://www.youtube.com/watch?v=-pz5vhTdkpA.

[2] Język opublikowany w 1988, który nie był stosowany komercyjnie, ale mocno wpłynął na takie współczesne języki jak Java, Python, C#.

Python jest językiem interpretowanym. W zasadzie kod jest kompilowany do kodu bajtowego przed wykonaniem, ale ten proces nie jest widoczny dla użytkownika, który „widzi" tylko wykonanie programu.

Python jest językiem ogólnego zastosowania, nie nakierowanym na jakieś szczególne zastosowania czy środowiska. Oczywiście, w niektórych obszarach jest mniej przydatny niż inne języki, głównie ze względu na stosunkowo duże wymagania czasowe i pamięciowe (choć można temu, przynajmniej częściowo, zaradzić poprzez użycie modułów napisanych w C).

Język jest obiektowo-zorientowany, ale wspiera też programowanie strukturalne i elementy programowania funkcjonalnego (inspirowane takimi językami jak Lisp, Haskell i ML).

Python jest wyposażony w odśmiecacz (*garbage collector*), jak Java, oraz, również jak Java, w olbrzymią bibliotekę standardową w formie pakietów i modułów, które mogą być zaimportowane do programów użytkownika. Dodatkowo, istnieje oficjalne repozytorium pakietów zewnętrznych (*third-party*) zwane ***PyPI***.[1] Za pomocą specjalnej aplikacji (***pip***) można te pakiety łatwo zainstalować i następnie używać w swoich programach. Obecnie (luty 2023), ***PyPI*** zawiera ok. 440 000 pakietów! Mogą one być używane w zasadzie do wszystkiego: automatyzacji, analizy danych, obsługi baz danych, tworzenia dokumentacji, budowania interfejsów graficznych, przetwarzania obrazów i dźwięku, uczenia maszynowego, multimediów, programowania sieciowego, programowania naukowego, administrowania systemem, budowania środowisk testowych czy do tworzenia aplikacji webowych.

1.3 Powłoka Pythona

Najlepszym sposobem na rozpoczęcie nauki Pythona jest użycie powłoki Pythona (*Python shell*) nazywana REPL — **R**ead (co wpisał użytkownik), **E**valuate (co zostało wprowadzone), **P**rint (wartość wyrażenia), **L**oop (wróć do wczytywania następnych danych). Powłoka może być uruchomiona poprzez wywołanie w linii poleceń ***python*** (czasami ***python3***) i następnie wpisywanie kolejnych poleceń po znaku zachęty, którym jest `>>>`. Jeśli wprowadzona instrukcja jest wyrażeniem, czyli gdy zwraca wartość (różną od **None**), ta wartość będzie wypisana po wciśnięciu `ENTER`:

```
$ python
Python 3.8.10 (default, Jun 22 2022, 20:18:18)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 7
>>> y = 12
>>> x+y
19
>>> [a**2 for a in range(1,11)]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> exit()
```

Jak pokazuje ten przykład, instrukcja przypisania (tu `x=7` i `y=12`) *nie jest* w Pythonie wyrażeniem[2] — dlatego po wprowadzeniu `x = 7` nie jest wypisywany żaden rezultat (patrz rozdz. 4.1, str. 40 oraz 4.2, str. 41).

[1] Python Package Index: https://pypi.org/

[2] W C/C++/Java przypisanie ma typ i wartość (referencji do) zmiennej po lewej stronie po przypisaniu.

Aby wyjść z powłoki Pythona można użyć **exit()** lub, prościej, wcisnąć Ctrl-D (Linux/Mac) lub Ctrl-Z-Enter (Windows).

W powłoce REPL (ale *nie* w skryptach Pythona!) możemy używać specjalnej „zmiennej" o nazwie _ (znak podkreślnika) — oznacza ona ostatnią wartość obliczoną i wypisaną:

```
>>> a = 6
>>> a
6
>>> b = _ + 3  # _ to 6
>>> b
9
>>> _ * 8      # _ to 9
72
```

Aby wczytać daną z klawiatury, można użyć – zarówno w REPL jak i skryptach – funkcji **input**,opcjonalnie podając jako argument napis zachęty dla użytkownika:

```
>>> whatever = input("Enter whatever: ")
Enter whatever: Hello
>>> whatever
'Hello'
>>> noprompt = input()
there was no prompt!
>>> noprompt
'there was no prompt!'
```

Funkcja zwraca to co zostało wprowadzone przez użytkownika jako napis (typu **str**); jeśli chodziło o liczbę, to należy przekonwertować ten napis na liczbę typu **int** lub **float** (ten ostatni odpowiada **double** w C/C++/Java):

```
1  >>> i = input("Enter an int: ")
2  Enter an int: 123
3  >>> i, type(i)
4  ('123', <class 'str'>)
5  >>> i = int(i)
6  >>> i, type(i)
7  (123, <class 'int'>)
8  >>>
9  >>> f = input("Enter a float: ")
10 Enter a float: 123.45
11 >>> f, type(f)
12 ('123.45', <class 'str'>)
13 >>> f = float(f)
14 >>> f, type(f)
15 (123.45, <class 'float'>)
16 >>> print(i, f)
17 123 123.45
18 >>> print("Numbers are " + str(i) + " and " + str(f))
19 Numbers are 123 and 123.45
```

Trochę wyjaśnień:

- Linia 3: dwie (lub więcej) oddzielone przecinkami wartości tworzą krotkę (patrz rozdz. 7.1, str. 118), która jest wypisywana (linia 4) jako sekwencja wartości w nawiasach okrągłych.
- Linia 3: funkcja **type** (w zasadzie to wywołanie konstruktora klasy **type**) zwraca obiekt reprezentujący typ (klasę) obiektu użytego jako argument — to, co zostało wypisane to reprezentacja napisowa tego typu.
- Linie 5 i 13: funkcje **int** i **float** (w zasadzie konstruktory odpowiednich klas) zwracają liczby reprezentowane podanymi jako argument napisami.
- Linie 5 i 13: zauważmy, że przypisujemy rezultat wywołań funkcji do tych samych zmiennych (i oraz f). Te zmienne (referencje) wskazują teraz na obiekty innych typów: wskazywały na **str**ingi, a teraz wskazują na obiekty typu **int** oraz **float** (porównaj linie 4 z 7 i 12 z 15).

W liniach 16 i 18 użyliśmy funkcji **print**. Przyjmuje ona dowolną liczbę oddzielonych przecinkami argumentów i wypisuje ich wartości przekonwertowane na napisy i oddzielone odstępami jako separatorami. Można to zmienić podając argument sep:

```
>>> a = 7
>>> b = 1.5
>>> c = 'Hello'
>>> print(a, b, c)
7 1.5 Hello
>>> print(a, b, c, sep=' - ')
7 - 1.5 - Hello
```

Domyślnie funkcja **print** (jak **println** w Javie) dodaje znak nowej linii na końcu. To też może być zmienione poprzez podanie argumentu end (który może być napisem pustym, wtedy nic nie będzie dodawane):

```
a = 123
b = 123.45
c = 'Hello'
print(a, b, sep=' and ', end=' and then ')
print(c)
```

drukuje

```
123 and 123.45 and then Hello
```

Zauważmy, że a i b są liczbami, nie napisami, ale zostały automatycznie przekonwertowane na napisy.

W Pythonie, jak w Javie, można składać napisy za pomocą operatora +, ale tylko wtedy, gdy oba operandy są napisami. W Javie na przykład, jeśli k jest **int**em, coś takiego "k = " + k zadziała: k będzie przekonwertowane do napisu i „dodane” do napisu "k = ". Aby ten sam efekt uzyskać w Pythonie, trzeba przekonwertować k do napisu „ręcznie”:

```
>>> k = 3
>>> print("k = " + str(k))
k = 3
>>> print("k = " + k)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

W Pythonie używana (i wymuszana przez interpreter) jest rzadko spotykana konwencja pisania kodu: bloki (pętli, **if**ów, definicji klas i funkcji, ...) są wyróżnione przez wcięcia (*indentację*) fragmentów kodu i *tylko* przez wcięcia (żadnych klamerek, żadnych begin...end i tym podobnych)! Wielkość wcięć jest w zasadzie dowolna, ale przyjęło się stosować zawsze 4 spacje — nie należy za to nigdy używać TABów![1] Dotyczy to również REPL. Na przykład następująca pętla ma ciało złożone z dwóch linii. Zauważmy, że po rozpoczęciu pętli[2] powłoka oczekuje jej ciała i zmienia się znak zachęty z >>> na .... Obie linie ciała muszą być wcięte tak samo (zalecane 4 spacje); po ostatniej linii wprowadzamy pustą linię aby wrócić do poziomu wcięć sprzed pętli

```
>>> for n in [2, 6, 8, 10]:
...     m = 2*n + 1
...     print(m)
...
5
13
17
21
```

1.4 IDLE IDE

Standardowa instalacja Pythona zawiera zwykle ***IDLE*** (Python's Integrated Development and Learning Environment) — bardzo proste i lekkie IDE, którego można używać do pisania małych programów, eksperymentowania albo testowania fragmentów kodu w Pythonie. Wywołuje je się z linii komend przez ***idle***. Najważniejsze cechy IDLE to:

- zaimplementowane w 100% w czystym Pythonie, przy użyciu standardowego pakietu ***tkinter***, za pomocą którego można tworzyć niezależne od platformy interfejsy graficzne;
- wieloplatformowość: działa podobnie w systemach Windows i Linux/Mac;
- wbudowana powłoka Pythona (interaktywny interpreter) z kolorowaniem składni, wejścia, wyjścia i komunikatów o błędach;
- wielookienkowy edytor tekstu z kolorowaniem składni, automatycznymi wcięciami, podpowiedziami, automatycznym uzupełnianiem, itd;
- przeszukiwanie i zastępowanie tekstu w wielu oknach i w wielu plikach;
- debugger z punktami przerwania, pracą krokową, widoczną globalną i lokalną przestrzenią nazw;
- dialogi do podglądu i zmian konfiguracji.

Szczegóły można znaleźć w podręczniku[3].

1.5 Zen Pythona

Każdy Pythonista[4] zna tak zwany *Zen of Python* — kolekcję 20 „zasad" pisania kodu w Pythonie opublikowaną przez Tima Petersa w roku 1999, i których tylko 19 jest

[1] Wszystkie edytory tekstu „rozumiejące" Pythona dodają 4 spacje po wciśnięciu klawisza TAB na początku linii.

[2] Instrukcje wymagające ciała — jak pętle czy **if**y — kończą się dwukropkiem.

[3] https://docs.python.org/3/library/idle.html

[4] Tak nazywają siebie użytkownicy i fani języka Python.

znanych. *Zen of Python* można znaleźć jako PEP 20[1] albo pisząc w powłoce Pythona (REPL) `import this`:

```
>>> 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!
```

Na marginesie, PEPs[2] (Python Enhancement Proposals) to dokumenty przedstawiające propozycje nowych elementów języka bądź podstawowych bibliotek, a także różne inne aspekty Pythona.

1.6 Kilka uwag na temat stylu

Python jest językiem wysokiego poziomu, więc program w Pythonie jest zwykłym plikiem tekstowym. Tekst może mieć różną formę i dopóki jest składniowo poprawny, interpreter nie będzie się skarżył. Jest jednak ważne, zarówno dla samego autora programu jak i dla tych, którzy w przyszłości będą go czytać, aby trzymać się pewnych zasad dotyczących stylu pisanego kodu tak, aby był jak najłatwiejszy do czytania i zrozumienia. Wspólne dla wszystkich zasady pisania programów ułatwiają też współpracę programistów pracujących nad jednym projektem. Słynny PEP 8[3] opisuje szczegółowo zalecane zasady pisania kodu i jest stale uzupełniany w miarę rozwoju języka. Każdy programista Pythona powinien przeczytać cały ten dokument; poniżej wymienimy tylko jego skrót:

- Linie kodu nie powinny być dłuższe niż 79 znaków.
- Linie kontynuacji powinny być wcięte o dodatkowe 4 spacje.
- Definicje funkcji i klas powinny być rozdzielone dwoma pustymi liniami.
- Definicje metod w klasie powinny być oddzielone jedną pustą linią.
- Nie należy otaczać spacjami indeksów, wywołań funkcji, znaku równości w przypisaniach argumentów nazwanych.

[1] http://peps.python.org/pep-0020
[2] https://peps.python.org
[3] http://peps.python.org/pep-0008

- Należy otaczać spacjami operatory i stosować jedną spację za przecinkiem.
- Komentarze (od znaku # do końca linii) powinny być umieszczane w osobnych liniach.
- Komentarze dokumentacyjne (o których będziemy jeszcze mówić) powinny być stosowane dla wszystkich nietrywialnych funkcji, klas i modułów.
- Zaleca się stosowanie CamelCase (z pierwszą literą dużą) w nazwach klas i wyjątków.
- Zaleca się używanie lower_case_with_underscores w nazwach funkcji, zmiennych i atrybutów; dopuszczalny jest też camelCase ale z pierwszą literą małą.
- Nazwy zmiennych interpretowanych jako stałe powinny być w ALL_CAPITALS z podkreślnikami.
- Pierwszy parametr niestatycznych metod (odpowiadający referencji do obiektu na rzecz którego nastąpiło wywołanie) powinien nazywać się self.[1]
- Pierwszy parametr metod klasowych powinien się nazywać cls.
- Importy powinny być podzielone na sekcje: najpierw moduły z biblioteki standardowej, potem z bibliotek zewnętrznych (*third-party*), na końcu własne. W ramach każdej sekcji kolejność powinna być alfabetyczna.

1.7 Typy wbudowane

Typy danych Pythona omówimy bardziej szczegółowo w osobnym rozdziale (rozdz. 3, str. 31); tu wymienimy tylko niektóre z nich nie wchodząc w szczegóły.

1.7.1 Typy liczbowe

Są trzy wbudowane typy liczbowe: int, float i complex. Mogą one być dowolnie mieszane w wyrażeniach arytmetycznych — nawet inty mają dobrze zdefiniowane części rzeczywiste *i urojone*! Obiekty tych typów są niemodyfikowalne.

Typ int reprezentuje liczby całkowite o praktycznie nieograniczonej wielkości, dodatniej lub ujemnej.

Typ float opisuje liczby rzeczywiste, a tak naprawdę „nieskończenie mały” podzbiór liczb wymiernych, ze względu na ograniczoną 64 bitami reprezentację binarną. Są implementowane jako liczby typu double (*nie* float!) znanych z C/C++/Javy.

Typ complex opisuje liczby zespolone reprezentowane jako pary liczb typu float — ich części rzeczywiste i urojone.

1.7.2 Typy sekwencyjne

Wbudowane typy obiektów reprezentujących sekwencje obiektów (ściśle rzecz biorąc referencji do obiektów) to list, tuple i range. Sekwencje i kolekcje są w Pythonie, tak jak kolekcje w Javie, sekwencjami i kolekcjami *referencji* do obiektów. Referencje jednak nie są, jak w Javie, „gołymi” wskaźnikami (więcej o tym później).

Typ list odpowiada ArrayList z Javy lub vector z C++. Są to listy referencji do obiektów dowolnego typu; elementami listy mogą być referencje do innych list czy krotek. Oznaczane są jako oddzielone przecinkami listy ujęte w nawiasy kwadratowe.

[1] W C++/Javie ten parametr w ogóle nie występuje na liście parametrów, więc nie ma jak nadać mu nazwy. Dlatego w tych językach jego nazwa jest ustalona raz na zawsze – this.

Listy są modyfikowalne (mutowalne) — można do nich dodawać i usuwać z nich elementy, można zamieniać ich elementy referencjami do innych obiektów innych typów nie zmieniając przy tym referencji do samego obiektu listy i jego identyfikatora:

```
mylist = []                # pusta lista
print(type(mylist))
print(id(mylist))
mylist.append(1)
mylist.append([11, 22])
mylist.extend([33, 44])
print(mylist)
print(id(mylist))
del mylist[3]
print(mylist)
print(id(mylist))
```

drukuje

```
<class 'list'>
140144996134208
[1, [11, 22], 33, 44]
140144996134208
[1, [11, 22], 33]
140144996134208
```

Type **tuple** (krotka) reprezentuje niemodyfikowalną sekwencję referencji do obiektów. Niezmienność oznacza, że nie możemy do krotki dodawać czy usuwać elementów, ani zastępować referencji, które są jej elementami, referencjami do innych obiektów. Jednakże, jeśli taka referencja odnosi się do obiektu modyfikowalnego (jak lista), to ten obiekt *może* być modyfikowany. Krotki są oznaczane jako oddzielona przecinkami lista referencji ujęta w nawiasy okrągłe (nie zawsze są one wymagane) — jeśli jest to krotka jednoelementowa, za jej jedynym elementem musi pojawić się przecinek:

```
mytuple = (5, 'Hello', [1, 2, 3], (4,))
print(type(mytuple))
print(type(mytuple[3]))
print(mytuple)
mytuple[2].pop(1)
print(mytuple)
mytuple[1] = 'World'
```

drukuje

```
<class 'tuple'>
<class 'tuple'>
(5, 'Hello', [1, 2, 3], (4,))
(5, 'Hello', [1, 3], (4,))
Traceback (most recent call last):
File "/usr/lib/python3.10/idlelib/run.py",
        line 578, in runcode
  exec(code, self.locals)
File "/home/werner/p.py", line 7, in <module>
  mytuple[1] = 'World'
TypeError: 'tuple' object does not support item assignment
```

Typ **range** reprezentuje niemodyfikowalną sekwencję liczb całkowitych tworzących ciąg arytmetyczny. Jest często stosowany w pętlach, ale ma też inne zastosowania w najrozmaitszych sytuacjach. Konstruktor **range** przyjmuje wartość pierwszego elementu, górną granicę generowanych liczb taką, że wszystkie liczby są mniejsze od niej (lub większe, jeśli różnica jest ujemna) oraz różnicę ciągu, czyli różnicę między kolejnymi wyrazami (krok). Konstruktor jednoargumentowy z argumentem n jest równoważny konstruktorowi z argumentami 0, n i 1. Zauważmy, że obiekty tego typu *nie* są kolekcjami: tworzą kolejne elementy po jednym, „na żądanie".

```
myrange = range(5, 10, 2)
print(myrange)
print(list(myrange))
print(list(range(24, -2, -5)))
print(list(range(5)))
```

drukuje

```
range(5, 10, 2)
[5, 7, 9]
[24, 19, 14, 9, 4, -1]
[0, 1, 2, 3, 4]
```

1.7.3 Zbiory i słowniki

Typ **set** reprezentuje zbiory, jak **set** w Javie czy C++. Jest to modyfikowalna kolekcja referencji do obiektów. Implementacja jest oparta na haszowaniu,[1] i elementy muszą być niemodyfikowalne, a ściślej *haszowalne*. Istnieje też typ **frozenset**: reprezentujący zbiory niemodyfikowalne (i haszowalne), które wobec tego mogą być elementami innych zbiorów. Na przykład:

```
myset = {1, 2, frozenset(['Hello', 'World']), 'Paris'}
print(type(myset))
print(2 in myset)
anotherset = {3, 4}
sum  = myset | anotherset          # suma
prod = myset & set([1, 2, 3, 4])   # przecięcie
print(myset-anotherset)            # różnica
print(anotherset)
print(sum)
print(prod)
```

drukuje

```
<class 'set'>
True
{frozenset({'World', 'Hello'}), 1, 2, 'Paris'}
{3, 4}
{1, 2, 3, 'Paris', 4, frozenset({'World', 'Hello'})}
{1, 2}
```

Typ **dict** odpowiada klasie **LinkedHashMap** z Javy. Reprezentuje kolekcję (słownik, mapę) par klucz/wartość, gdzie klucze muszą być unikalne, niemodyfikowalne i haszowalne. Na przykład:

[1] Jak **HashSet** w Javie czy **unordered_set** w C++.

```
mymap = {}      # pusty słownik, nie zbiór (set)
mymap['France'] = 'Paris'
mymap[2] = [1, 2, 3]
mymap[2].append(4)
print(2 in mymap)
anothermap = {'Spain' : 'Madrid', 2 : complex(1, 2)}
mergedmap = mymap | anothermap
print(mymap)
print(anothermap)
print(mergedmap)
```

prints

```
True
{'France': 'Paris', 2: [1, 2, 3, 4]}
{'Spain': 'Madrid', 2: (1+2j)}
{'France': 'Paris', 2: (1+2j), 'Spain': 'Madrid'}
```

Implementacja samego języka Python oparta jest w dużej mierze właśnie na słownikach.

1.7.4 Inne typy

Jest też kilka innych typów wbudowanych, które omówimy w swoim czasie. Najważniejszym z nich jest oczywiście typ **str** (string) — omówimy ten typ bardziej szczegółowo w następnych rozdziałach.
Warto tu podkreślić, że *nie ma* typu odpowiadającego typowi **char** z innych języków — znak jest po prostu jednoliterowym napisem typu **str**.

1.8 Obiekty i referencje

Nazwy zmiennych w Pythonie to tak naprawdę nazwy *referencji* do obiektów. Te referencje są w Pythonie implementowane jako C-struktury[1] w pamięci, których składowymi są wskaźniki na obiekty reprezentujące typ danej, na obiekty reprezentujące wartości danych, tak zwany licznik odwołań i dodatkowe dwa wskaźniki używane przez odśmiecacz.
Struktury reprezentujące wartości posiadają identyfikatory (*ID number*, liczby całkowite, zwykle po prostu adresy obiektów) — które są unikalne: **żadne dwa obiekty dostępne w tym samym czasie nie mogą mieć takich samych identyfikatorów.** W czasie działania programu, ta sama referencja (a więc i nazwa) może zmienić interpretację i wskazywać na inny typ i wartość, o innym ID. Innymi słowy, obiekty w Pythonie są silnie typowane, ale nie nazwy referencji.

Istnieje wbudowana funkcja, **id**, która dla danej referencji zwraca ID obiektu wskazywanego jako wartość. W przykładzie poniżej używamy też innej wbudowanej funkcji, **type**, która zwraca obiekt reprezentujący typ wartości wskazywanej przez podaną jako argument referencji:

```
>>> x = 3
>>> y = 7
>>> id(x), id(y)
(9353376, 9353504)          # 1
```

[1] A nie „gołe” wskaźniki, jak w Javie.

```
>>> x = y                    # 2
>>> id(x), id(y)
(9353504, 9353504)           # 3
>>> y = 'Python'             # 4
>>> id(x), id(y)
(9353504, 139733635191952) # 5
>>> type(x), type(y)         # 6
(<class 'int'>, <class 'str'>)
```

Jak widzimy (linia 1), identyfikatory obiektów wskazywanych przez referencje x i y są inne: odpowiednio 9353376 i 9353504. Po przypisaniu[1] x = y (linia 2), zarówno x jak i y odnoszą się do dokładnie tego samego obiektu (linia 3) typu **int** i wartości 7. Co się stało z obiektem o ID równym 9353376? Jest już niedostępny i zostanie usunięty przez odśmiecacz.
W linii 4 przypisujemy do y inną wartość, innego typu — teraz jest to **str**ing. Jak widzimy z wydruku, identyfikatory, typy i wartości x i y są teraz inne.

Nawet literały wartości są traktowane jako referencje do obiektów, a zatem można na ich rzecz wywoływać metody. Na przykład typ (klasa) **int** ma metodę **bit_length**, która zwraca liczbę bitów potrzebnych do zapisania danej liczby w postaci binarnej, natomiast napisy (typu **str**) mają metodę **upper**:

```
>>> (1).bit_length()
1
>>> (255).bit_length()
8
>>> (256).bit_length()
9
>>> 'paris'.upper()
'PARIS'
```

(Nawiasy wokół literałów typu **int** są potrzebne, w przeciwnym razie kropka byłaby zinterpretowana jako kropka dziesiętna).

Programy w Pythonie, tak jak w innych językach, mogą być napisane w dowolnym edytorze tekstu. Aby je uruchomić, wystarczy z linii poleceń wpisać

```
python3 python_program.py
```

lub

```
python python_program.py
```

zależnie od konfiguracji.
Można też uczynić plik z programem bezpośrednio wykonywalnym poprzez, w systemach Linux/Mac, umieszczenie w pierwszej linii tak zwanego *shebang* i zmianę uprawnień tak, aby plik był przez system rozpoznawany jako wykonywalny ('x'). Było również przyjętą praktyką specyfikowanie jawnie (w drugiej linii) kodowania pliku źródłowego z programem. Teraz nie jest to konieczne, bo w Pythonie 3 domyślnym kodowaniem zawsze jest UTF-8:

[1] Przypisujemy tu referencje, nie wartości.

Listing 1 AAB-First/First.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  # a comment
5  v = 1.25                                  # 1
6  print(id(v))
7  if 1 <= v <= 2:
8      print('v^2 =', v**2, end = '; ')
9      print('v^3 =', v**3, end = '; ')
10     print('v^4 =', v**4)
11     v = "now I'm a string!"               # 2
12     print(id(v))
13 print('This is v:', v)
```

Program drukuje

```
20164608
v^2 = 1.5625; v^3 = 1.953125; v^4 = 2.44140625
140529498173656
This is v: now I'm a string!
```

Widać, że obiekt wskazywany przez v początkowo (linia 1) reprezentuje liczbę **float** o identyfikatorze 20164608, ale po modyfikacji (linia 2) ta sama referencja v odnosi się do obiektu o innym ID (140529498173656) i innym typie (**str**). Obiekt o ID równym 20164608 przestał być dostępny.
Zauważmy też, że w Pythonie istnieje operator potęgowania, oznaczany, jak w Fortranie, przez podwójną gwiazdkę (**) — nigdy przez ^!
W tym przykładzie nie tylko wartość, ale i typ referencji został zmieniony. Jednakże obiekty wszystkim typów liczbowych są tak naprawdę **niemodyfikowalne** (*immutable*), więc pozorna zmiana ich wartości w rzeczywistości odpowiada utworzeniu nowego obiektu i zmianie wskaźnika w referencji:

```
>>> x = 1
>>> id(x)
9353312
>>> x = 2
>>> id(x)
9353344
```

Jest wiele niemodyfikowalnych typów w Pythonie: oprócz typów liczbowych najważniejszym przykładem są tu napisy (**str**):

```
>>> x = 'John'
>>> y = x
>>> id(x), id(y)
(139733656059312, 139733656059312)
>>> x = x + ' and Mary'
>>> x, y
('John and Mary', 'John')
>>> id(x), id(y)
(139733654847152, 139733656059312)
```

Jak widzimy, po modyfikacji zmienna x reprezentuje teraz inny obiekt, o innym identyfikatorze.

Z drugiej strony, obiekty wielu innych typów *są* modyfikowalne (*mutable*). Ważnym przykładem są listy (**list**). Lista jest listą *referencji* do obiektów (dowolnego typu). Można do listy dodawać nowe elementy czy modyfikować istniejące, ale referencja do samej listy jako całości pozostaje wtedy niezmieniona i jej ID nie ulega zmianie. Zilustrujmy to przykładem:

Listing 2 AAC-Refs/Refs.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  from copy import deepcopy
5
6    # a and be are two names of the same objects
7  a = [1, 2, [3, 5]]
8  b = a
9  print(b is a)        # True
10 a[2][1] = 4
11 print(b, '\n')       # [1, 2, [3, 4]]
12
13   # shallow copy: lists distinct, elements are copied
14   # but these are references referring to the same objects!
15 c = list(a)
16 c.append(5)
17 print(c is a)        # False
18 print(c[2] is a[2]) # True
19 a[2][1] = 6
20 print(a)             # [1, 2, [3, 6]]
21 print(c, '\n')       # [1, 2, [3, 6], 5]
22
23   # deep copy: lists distinct, elements too (recursively)
24 d = deepcopy(a)
25 d.append(5)
26 print(d is a)        # False
27 print(d[2] is a[2]) # False
28 a[2][1] = 7
29 print(a)             # [1, 2, [3, 7]]
30 print(d)             # [1, 2, [3, 6], 5]
```

W linii 7 tworzymy listę a: jej dwa pierwsze elementy to referencje do (niemodyfikowalnych) **int**ów a trzeci jest referencją do innej listy. Po b = a referencje b i a odnoszą się do tego samego obiektu listy. Dokonujemy zmiany używając nazwy a (linia 10): a[2] jest trzecim elementem listy (indeksowanie jest od zera), który sam jest referencją do (modyfikowalnej) listy [3, 5] a zatem a[2][1] jest jej drugim elementem (referencją do liczby 5). Kiedy drukujemy b, element b[2][1] jest również zmieniony, bo zarówno a jak i b oznaczają ten sam obiekt.

Możemy się o tym przekonać w linii 9: operator **is** porównuje identyfikatory (adresy) obiektów i zwraca **True** tylko gdy dwa obiekty są tak naprawdę tym samym obiektem

(jest też operator ==, który porównuje *wartości* — patrz rozdz. 4, str. 40).

W linii 15 tworzymy nową listę posyłając do konstruktora (o listach i konstruktorach powiemy później — patrz rozdz. 7.1, str. 118 i rozdz. 11.1, str. 213). Teraz `c` i `a` są różnymi listami (linia 17). Jednakże elementy `c` są kopiami elementów `a`, które są referencjami, a nie wartościami, więc wskazują na te same obiekty. Dlatego `print(c[2] is a[2])` drukuje **True**. Ten element jest modyfikowalny, bo jest listą, więc zmiana jednego z jej elementów (linia 19) wpływa i na `a` i na `c`. Takie „sprzężenie" list jest często niewskazane — aby utworzyć rzeczywiście niezależne listy możemy zatem użyć funkcji **deepcopy** z modułu ***deepcopy*** (linia 24).

1.9 Funkcje

Funkcje omówimy bardziej szczegółowo później (patrz rozdz. 5, str. 63). Tu podamy tylko przykład pokazujący jak funkcja może być zdefiniowana i użyta.

Jak wszystko inne, funkcje w Pythonie są reprezentowane obiektami. Definicja funkcji wygląda jak w przykładzie poniżej. Po słowie kluczowym **def** piszemy nazwę referencji do której chcemy przypisać funkcję, następnie w nawiasach nazwy parametrów (bez wskazywania typów!) i stawiamy dwukropek wskazujący, że dalej nastąpi blok instrukcji stanowiący ciało funkcji. Sam blok musi być zatem wcięty bo, jak wiemy, nie ma innego sposobu na zaznaczenie gdzie się blok instrukcji złożonej zaczyna a gdzie kończy. Jak w innych językach, jeśli funkcja zwraca nietrywialny wynik, ostatnią wykonywaną instrukcją musi być instrukcja **return**. W Pytonie każda funkcja coś zwraca — jeśli nie wskazaliśmy wartości zwracanej poprzez **return**, to zwracane jest **None**.

Zauważmy, że w linii 3 definiujemy funkcję (obiekt ją reprezentujący) i związujemy ją z nazwą, w tym przypadku **adder**. W Pythonie jest to instrukcja *wykonywalna,* jak przypisanie. Nazwa **adder** nie jest nazwą, poprzez którą będziemy musieli tę funkcję wywoływać. Jest to raczej nazwa *referencji* do samej funkcji. Jeśli chcemy, możemy tę referencję przekopiować do innej, co robimy w linii 17

Listing 3 AAY-Funcs/funcs.py

```
1  #!/usr/bin/env python
2
3  def adder(a, b):          # definition + assignment
4      return a + b
5
6  print(type(adder))        # type
7
8  rn = adder(7,4)           # adding numbers
9  print(rn, type(rn))
10
11 rs = adder('Monte ', 'Carlo')  # 'adding' strings
12 print(rs, type(rs))
13
14 print(adder.__str__())  # invoking method on a function!
15
16 print(id(adder))
17 a = adder                # copying reference, now a is adder
18 print (id(a))
19 print(a is adder)        # a and adder are the same thing
```

```
20
21 adder = a(1.25, 3.75)    # now adder becomes a float
22 print('Now adder is', adder, type(adder))
23
24 print("But 'a' is still our function!", a(2.5, 3.5))
25
26 print('='*30 + '\nAnd now list of attributes:')
27 print(str(dir(a)) + '\n' + '='*30)
28
29 print('But attribute a.__name__ is still ' + a.__name__)
30 a.__name__ = 'whatever'
31 print('No problem, now it is ' + a.__name__)
```

i od tej pory **a** i **adder** odnoszą się do dokładnie tego samego obiektu i są równoważne. Możemy nawet zmienić typ referencji **adder**, tak, żeby teraz wskazywała na **float** (linia 21), a do funkcji mamy wciąż dostęp pod nazwą **a** (linia 24).

Program drukuje

```
<class 'function'>
11 <class 'int'>
Monte Carlo <class 'str'>
<function adder at 0x7f078af0bd90>
139670372531600
139670372531600
True
Now adder is 5.0 <class 'float'>
But 'a' is still our function! 6.0
==============================
And now list of attributes:
['__annotations__', '__builtins__', '__call__', '__class__',
'__closure__', '__code__', '__defaults__', '__delattr__',
'__dict__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__get__', '__getattribute__', '__globals__',
'__gt__', '__hash__', '__init__', '__init_subclass__',
'__kwdefaults__', '__le__', '__lt__', '__module__',
'__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
==============================
But attribute a.__name__ is still adder
No problem, now it is whatever
```

Dla programistów wychowanych na C/C++/Javie może się wydawać dziwne, że nigdzie w definicji funkcji nie określiliśmy typów parametrów ani typu zwracanego. W Pythonie jest to nie tylko możliwe, ale nawet nie byłoby jak tego zrobić![1] To znaczy, że możemy do funkcji **adder** wysłać jakiekolwiek argumenty dowolnych typów, jeśli tylko będzie dla nich miał sens operator + użyty w definicji funkcji![2]

[1] Są sposoby na wskazywanie tych typów, ale tylko dla „ludzkich” czytelników, ewentualnie dla narzędzi zewnętrznych; interpreter i tak te informacje zignoruje.

[2] Jest to przykład tak zwanego *kaczego typowania* (ang. *duck typing*.)

Jako obiekty, funkcje mają atrybuty, i nawet sporo (mają nawet metody – linia 14). Możemy zobaczyć ich listę wywołując funkcję **dir**, jak w linii 27. Jest wśród nich `__name__`, które wciąż jest `adder` (linia 29). Ale to jest tylko napis i, jeśli nam się nie podoba, możemy go zastąpić jakimkolwiek innym napisem (co robimy w linii 30).

1.10 Moduły

Aby używać obiektów ze standardowej biblioteki, musimy, podobnie jak w Javie czy C++, „zaimportować" jej odpowiednie części do naszego programu.
W Javie używamy do tego celu instrukcji **import**. W zasadzie **import** w Javie nie jest w ogóle potrzebne, czyni tylko pisanie programu wygodniejszym. Instrukcja

```
import java.util.List;
```

mówi tylko kompilatorowi, że nazwa `List` powinna być tak naprawdę zamieniona na pełną, kwalifikowaną nazwę `java.util.List`. Jeśli nam nie przeszkadza pisanie `java.util.List` zamiast `List`, żadnych importów nie potrzebujemy.
Z kolei w C/C++ mamy dyrektywę **#include**. To jest co innego:

```
#include <map>
```

fizycznie włącza zawartość pliku ***map*** i wysyła ją do kompilacji razem z naszym kodem[1]
Python jest językiem interpretowanym i z punktu widzenia użytkownika nie ma osobnej fazy kompilacji — wszystko dzieje się podczas wykonywania. Używamy tu instrukcji **import**, która ma więcej wspólnego z **#include** z C++ niż z **import** w Javie.
Na przykład

```
import math
```

importuje (i *wykonuje*) moduł (plik) ***math***. Moduły to normalne pliki pliki źródłowe z rozszerzeniem ***.py***. Są poszukiwane w katalogach z listy katalogów, którą możemy zobaczyć tak[2]

```
>>> import sys
>>> sys.path
['', '/usr/lib/python38.zip', '/usr/lib/python3.8',
 '/usr/lib/python3.8/lib-dynload',
 '/home/werner/.local/lib/python3.8/site-packages',
 '/usr/local/lib/python3.8/dist-packages',
 '/usr/lib/python3/dist-packages']
```

(pusty napis na początku oznacza *aktualny katalog*). Kiedy plik zostaje znaleziony, jest przez interpreter Pythona wczytywany *i wykonywany*. Zazwyczaj moduły zawierają tyko definicje klas, funkcji czy zmiennych, ale jeśli są tam na przykład instrukcje **print**, zobaczymy ich rezultat.
Zaimportowane nazwy obiektów zdefiniowanych w module znajdą się w przestrzeni nazw mającej tę samą nazwę co moduł i odnosimy się do nich poprzez ich nazwy kwalifikowane nazwą przestrzeni nazw. Na przykład moduł ***math*** zawiera definicję funkcji **sqrt**. Aby jej użyć, piszemy zatem **math.sqrt**

[1] Kompilatory używają różnych sprytnych sztuczek aby uczynić ten proces efektywniejszym i uniknąć kompilowania tego samego kodu wielokrotnie; są to jednak szczegóły techniczne.
[2] Można do tej listy dodać inne katalogi, patrz rozdz. **??**, str. **??**.

```
>>> import math
>>> math.sqrt(234.567)
15.315580302424065
```

Czasem wygodnie jest (lub wymaga tego wykształcona w danej dziedzinie tradycja) nadać modułowi inną nazwę, pod którą będziemy się do niego odnosić

```
>>> import math as m
>>> m.sqrt(234.567)
15.315580302424065
```

Możemy zaimportować z modułu tylko wybrane nazwy i skopiować je do naszej przestrzeni nazw — wtedy nie będziemy musieli ich kwalifikować:

```
>>> from math import sqrt, factorial
>>> sqrt(999.99)
31.6226184874055
>>> factorial(45)
119622220865480194561963161495657715064383733760000000000
```

Jeśli chcemy zaoszczędzić na pisaniu długich nazw lub aby uniknąć konfliktu nazw, możemy zaimportowane z modułu nazwy zmienić:

```
>>> from math import sqrt as root, factorial as fac
>>> root(87)
9.327379053088816
>>> fac(12)
479001600
```

W końcu można też zaimportować wszystkie nazwy z danego modułu, choć nie jest to zalecane, bo możemy doprowadzić do konfliktu nazw i trudno później, patrząc na kod, stwierdzić która nazwa pochodzi z którego modułu:

```
>>> from math import *
>>> from time import *
>>> gcd(3219, 1508)
29
>>> factorial(23)
25852016738884976640000
>>> log(999, pi)
6.033523597161682
>>> 4*atan(1.0)
3.141592653589793
>>> strftime("%a, %d %b %Y %H:%M:%S +0000", gmtime())
'Mon, 19 Sep 2022 17:21:38 +0000'
```

1.11 Help

Przy pracy w REPL mamy dostęp do systemu pomocy Pythona. Można uzyskać pomoc dla modułów, klas, funkcji itd.

```
>>> help()

Welcome to Python 3.10's help utility!
```

```
If this is your first time using Python, you should definitely check out
the tutorial on the internet at https://docs.python.org/3.10/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".
```

Spróbujmy `help(math)`:

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.
    ...
```

albo dla funkcji **sin** z tego modułu `help(math.sin)`

```
Help on built-in function sin in math:

math.sin = sin(x, /)
    Return the sine of x (measured in radians).
```

(wprowadź 'q' aby opuścić pomoc i wrócić do REPL).

Do badania własności modułów, klas, funkcji itd. przydatna jest też bardzo wbudowana funkcja **dir**[1]. Wywołana bez argumentów zwraca ona listę nazw widocznych w aktualnym zasięgu. Jeśli argumentem jest obiekt, to zwracana jest lista atrybutów tego obiektu. Tym obiektem może być moduł – zwracana lista zawiera wtedy atrybuty tego modułu. Jeśli jest to obiekt reprezentujący klasę, lista zawiera jej atrybuty i rekursywnie atrybuty jej klas bazowych.
Na przykład

[1] https://docs.python.org/3/library/functions.html?highlight=dir#dir

```
>>> dir()                # atrybuty w zakresie local
['__annotations__', '__builtins__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__']
>>>
>>> import math
>>> dir()                # 'math' dodane do listy
['__annotations__', '__builtins__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', 'math']
>>> __name__             # nazwa bieżącego modułu
>>>
'__main__'
>>>
>>> dir(math)            # atrybuty modułu 'math'
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm',
'pi', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
>>> type(math.gamma)     # co to gamma?
<class 'builtin_function_or_method'>
>>> dir(math.gamma)      # atrybuty funkcji gamma
['__call__', '__class__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__',
'__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__self__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__text_signature__']
>>> math.gamma.__name__
'gamma'
>>> math.gamma.__doc__   # docstring dla funkcji gamma
'Gamma function at x.'
>>> type(math.pi)        # co to pi?
<class 'float'>
>>> math.pi              # wartość pi
3.141592653589793
```

Nazwy wielu atrybutów mają na początku i końcu podwójny znak podkreślnika. Są to nazwy „specjalne”, zwane też „magicznymi” — ich znaczenie poznamy później (patrz rozdz. 11.8, str. 236).

1.12 Słowa kluczowe

Tak jak w innych językach, w Pytonie niektóre słowa (nazwy) są zarezerwowane dla samego języka (jak for czy while) i nie mogą być używane jako nazwy zmiennych, funkcji, klas, itd.
Są to

Tablica 1: Nazwy zarezerwowane w Pythonie

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Rozdział 2

Instrukcje przepływu sterowania

2.1 Instrukcja pass

Instrukcja **pass** pełni rolę instrukcji pustej, jak średnik w C/C++/Javie. Nie wykonuje absolutnie niczego, ale jest użyteczna, gdy składnia wymaga jakiejś instrukcji, ale niczego w tym miejscu nie chcemy i nie potrzebujemy.
Na przykład, możemy zdefiniować klasy opisujące pewne rodzaje wyjątków. To, czego potrzebujemy, to tylko sam typ wyjątku, nie będą nam przydatne żadne pola czy metody. Możemy zatem zostawić ciało definicji tych klas puste, zastępując je właśnie pustą instrukcją, jak w przykładzie poniżej

Listing 4 AAM-Pass/Pass.py

```
#!/usr/bin/env python

from math import log10

class ZeroArg(Exception):
    pass

class NegArg(Exception):
    pass

class ZeroDenom(Exception):
    pass

def f(num, denom):
    if denom == 0: raise ZeroDenom
    elif num == 0: raise ZeroArg
    elif num < 0:  raise NegArg
    return log10(num)/denom

for n, d in ( (-10, 2), (1000, 2), (100, 0), (0, 2) ):
    try:
        res = f(n, d)
        print("({:4d}, {}) -> {}".format(n, d, res))
    except Exception as e:
        print("({:4d}, {}) -> exception {}"
              .format(n, d, e.__class__.__name__))
```

który drukuje

```
( -10, 2) -> exception NegArg
(1000, 2) -> 1.5
( 100, 0) -> exception ZeroDenom
(   0, 2) -> exception ZeroArg
```

2.2 Instrukcja if

Instrukcja **if** jest podobna do analogicznych instrukcji w innych językach. W Pythonie warunki logiczne po **if**, **elif** (które odpowiada **else if** z innych języków) oraz **else** nie wymagają nawiasów. Trzeba pamiętać jednak o dwukropku na końcu, wskazującym, że co następuje dalej będzie blokiem kodu:

Listing 5 AAG-Iffs/Iffs.py

```
#!/usr/bin/env python3

number = int(input("Enter an int "))
if number < 0:
    print("negative numbers do not count")
elif number == 0:
    print("Nothing")
elif number == 1:
    print("One")
elif number == 2:
    print("Two")
else:
    print("Many")
```

Oczywiście, klauzule **elif** i **else** są opcjonalne i **else**, jeśli zostało użyte, musi być ostatnią klauzulą.

2.3 Wyrażenie warunkowe (if-else)

Wyrażenie warunkowe (**if-else**) nie jest instrukcją przepływu sterowania, ale ponieważ jest jakoś związane z instrukcją **if**, wspomnimy o nim również tutaj (bardziej szczegółowy opis, patrz rozdz. 4.14, str. 60).

Odpowiada ono wyrażeniom warunkowym oznaczanym przez trójargumentowy operator **?:** w C/C++/Javie, ale składnia jest nieco inna. Jest to *wyrażenie*, a więc ma wartość, która może być przypisana do zmiennej albo użyta jako część innego wyrażenia. Składnia jest następująca

```
expr_if_true if condition else expr_if_false
```

i wartością całego wyrażenia jest wartość expr_if_true jeśli condition daje wartość **True**, a wartość expr_if_false w przeciwnym przypadku. Tak jak w innych językach, operator trójargumentowy, również w Pythonie wyrażenie warunkowe ma własność skrótowości: w zależności od wartości logicznej warunku condition, albo tylko wartość expr_if_true, albo tylko wartość expr_if_false będzie obliczana

```
>>> a, b = 5, 0
>>> c = a//b if a > 6 else 3*a
>>> c
15
>>> d = a//b if a < 6 else 3*a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

(w pierwszym przypadku warunek nie był spełniony, więc wyrażenie a//b w ogóle nie było obliczane).

Wyrażenia warunkowe mogą być zagnieżdżane, ale jest to zwykle zły pomysł, bo mogą się stać mętne i trudne do zrozumienia

```
>>> for t in (20, 50, 90):
...     print("green" if t < 40 else "yellow" if t < 80 else "red")
...
green
yellow
red
```

2.4 Pętla while

Pętla **while** ma semantykę taką jak w innych językach. Różnicą jest, że ma opcjonalną klauzulę **else**. Będzie ona wykonana, gdy pętla zakończy się „normalnie", to znaczy *nie* na skutek wykonania instrukcji **break** lub **return** — jeśli **break** się wykona, ciało klauzuli **else** się *nie* wykona. Tak jak w innych językach, **break** znaczy „zakończ wykonywanie pętli". Podobnie, **continue** znaczy „przejdź do następnej iteracji, nie kończąc iteracji bieżącej".

Wszystkie te elementy zilustrowane są programem poniżej. Zauważmy, że ostatnia linia nigdy nie będzie wykonana, bo aby wyjść z głównej pętli, będziemy musieli użyć **break**:

Listing 6 AAJ-While/While.py

```
#!/usr/bin/env python

while True:
    number = int(input("Enter an int "))
    if   number == 0:
        break
    if number <= 1:
        print("Your number must be >= 2.")
        continue
    if number in (2, 3, 5, 7):
        print(str(number) + ' is prime.')
    elif number % 2 == 0:
        print(str(number) + ' is not prime.')
    else:
        i = 3
        while i*i <= number:
            if number % i == 0:
                print(str(number) + ' is not prime.')
                break
            i += 2
        else:
            print(str(number) + ' is prime.')
else:
    print('This will never be printed')
```

2.5 for loop

Pętla for przypomina tzw. pętlę *for-each* albo *ranged-for* z innych języków. Składnia jest

```
for target_list in expression_list:
    body_of_the_loop
[else:
    block_of_code]
```

gdzie klauzula else jest opcjonalna (i rzadko używana) i jest, jak dla pętli while, wykonywana tylko jeśli działanie pętli *nie* zakończyło się na skutek wykonania instrukcji break lub return. Wartością wyrażenia expression_list, która jest obliczana na początku i tylko raz, musi być obiekt iterowalny (**iterable**).

Obiekty iterowalne to takie, które posiadają metodę __iter__ zwracającą **iterator**.[1] Zamiast wywołania __iter__ na obiekcie, można przekazać ten obiekt jako argument wbudowanej funkcji iter. W niektórych sytuacjach funkcja ta potrafi „wyprodukować" iterator bez wywoływania __iter__, jeśli obiekt posiada metodę __getitem__.
Z kolei iterator to obiekt posiadający metodę __next__, która albo zwraca następny element, albo generuje wyjątek StopIteration gdy następnego elementu już nie ma. Zamiast wywołania __next__ na rzecz obiektu, można też posłać go jako argument wbudowanej funkcji next. Iterator powinien również mieć metodę __iter__, która po prostu zwraca self — tak więc iterator sam jest obiektem iterowalnym.

Pętla for zajmuje się tym wszystkim automatycznie. Mając obiekt iterowalny dostarczony przez expression_list, wywoływana jest funkcja iter, zwracająca iterator. Następnie ciało funkcji wywoływane jest z każdym elementem zwróconym przez next przypisanym do target_list; nie musi to być pojedyncza zmienna, jeśli next zwraca na przykład pary klucz/wartość słownika albo krotki). W pewnym momencie next zgłosi wyjątek StopIteration, który będzie obsłużony przez pętlę for, tak, że go nawet nie zobaczymy.
W poniższym przykładzie używamy pętli for a potem wykonujemy to samo zadanie „ręcznie" wywołując iter i później next aż do wystąpienia wyjątku:

Listing 7 AAK-For/For.py

```
1  #!/usr/bin/env python
2
3  persons = [("Ann", 23), ("Bea", 34), ("Sue", 29)]
4  for name, age in persons:
5      print('{} is {} years old'.format(name, age))
6
7  print('\n' + '*'*5 + " now 'by hand' " + '*'*5 + '\n')
8
9  it = persons.__iter__()
10             # or it = iter(persons)
11
12 a = it.__next__()
```

[1] Jak w Javie obiekty klas implementujących interfejs Iterable, który deklaruje metodę abstrakcyjną iterator zwracającą obiekt klasy implementującej interfejs Iterator.

```
13 print('{} is {} years old'.format(a[0], a[1]))
14
15 a = next(it) # equivalently like this
16 print('{} is {} years old'.format(a[0], a[1]))
17
18 a = next(it)
19 print('{} is {} years old'.format(a[0], a[1]))
20
21 a = it.__next__()
22 print('{} is {} years old'.format(a[0], a[1]))
```

Program drukuje

```
Ann is 23 years old
Bea is 34 years old
Sue is 29 years old

***** now 'by hand' *****

Ann is 23 years old
Bea is 34 years old
Sue is 29 years old
Traceback (most recent call last):
File "/home/werner/python/py_progs/sources/AAK-For/For.py",
        line 21, in <module>
    a = it.__next__()
StopIteration
```

Inny przykład, tym razem ze słownikiem

Listing 8 AAN-DictLoop/DictLoop.py

```
1 #!/usr/bin/env python
2
3 fourSeasons = {'Spring' : 'E major', 'Summer' : 'G minor',
4                'Autumn' : 'F major', 'Winter' : 'F minor'}
5
6 for key, value in fourSeasons.items():
7     print(key + ' is in the key of ' + value)
```

Program drukuje

```
Spring is in the key of E major
Summer is in the key of G minor
Autumn is in the key of F major
Winter is in the key of F minor
```

Zauważmy, że nie ma tu żadnej zmiennej oznaczającej kolejne indeksy elementów dostarczanych przez iterator, jak w klasycznej pętli **for** z C/C++/Javy. Często jednak taki index jest przydatny; dlatego istnieje wbudowana funkcja **enumerate** , która

dla danego obiektu iterowalnego zwraca obiekt typu **enumerate**. Obiekt ten sam jest iterowalny i przy kolejnych wywołaniach **next** zwraca krotki, których pierwszym elementem jest kolejny indeks, a drugim to co zwrócone by było przez „pierwotny" obiekt iterowalny. Na przykład

Listing 9 AAL-ForEnum/Enum.py

```
#!/usr/bin/env python

lst = [1, 7, 4, 9, 2, 6, 3]
indsOdd = []
for i, e in enumerate(lst):
    if e % 2 != 0:
        indsOdd += [i]
print('Odd elements at positions',indsOdd)
```

drukuje indeksy nieparzystych elementów listy:

```
Odd elements at positions [0, 1, 3, 6]
```

Indeksy zwracane przez enumerator nie muszą zaczynać się od zera; wartość startową można przesłać jako drugi argument funkcji **enumerate**:

Listing 10 AAO-EnumStart/EnumStart.py

```
#!/usr/bin/env python

seasons = [ "Spring", "Summer", "Autumn", "Winter" ]
for ind, s in enumerate(seasons, 1):
    print('Season', ind, '->', s)
```

co drukuje

```
Season 1 -> Spring
Season 2 -> Summer
Season 3 -> Autumn
Season 4 -> Winter
```

Indeksy można też otrzymać używając wbudowanej funkcji **range**, która zwraca obiekt iterowalny reprezentujący ciąg arytmetyczny liczb całkowitych. Obiekt taki może być uzyskany przez wywołanie funkcji `range(start, stop, step)`: ciąg rozpocznie się od start (domyślnie 0), różnica ciągu będzie wynosiła step (domyślnie 1), a ostatnim elementem zwróconym będzie ostatni wciąż mniejszy od sop (lub ostatnim większym od stop jeśli step jest ujemny). Specjalna wersja, `range(n)`, jest równoważna **range(0, n, 1)**. Poniżej pokazujemy „anatomię" użycia **range**, a potem równoważną pętlę **for**

```
>>> r = range(1, 6, 2)
>>> type(r)
<class 'range'>
>>> len(r)
3
```

```
>>> hasattr(r,'__iter__')
True
>>> it = iter(r)
>>> hasattr(it,'__next__')
True
>>> it.__next__()
1
>>> next(it)
3
>>> next(it)
5
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
>>> for i in r:    # everything taken care of by 'for'
...     print(i)
1
3
5
```

(wbudowana funkcja **hasattr** sprawdza, czy obiekt ma podany atrybut).

2.6 Instrukcja match

Instrukcja **match** odpowiada instrukcji **switch** w Javie lub C++, ale ma więcej możliwości bardziej przypominając podobne konstrukcje w językach Haskell, Rust czy Scala.

W swej najprostszej formie jest łatwa do zinterpretowania i zrozumienia, jeśli pamiętamy **switch** z C/C++/Javy:

Listing 11 AAQ-MatchBasic/MatchBasic.py

```
1  #!/usr/bin/env python
2
3  k = 3
4  match k:
5      case 0:
6          print('zero')
7      case 1 | 2 | 3 | 4:          ❶
8          print('less than 5')
9      case 5:
10         print('five')
11     case _:                     ❷
12         raise ValueError()
```

Zauważmy, że warunki w poszczególnych klauzach **case** można ORować (❶) za pomocą kreski pionowej |. Zamiast **default** z innych języków można użyć klauzuli „łapiącej wszystko” (❷), tradycyjnie oznaczanej przez symbol _ (podkreślnik).

Zwróćmy też uwagę, że nie ma tu własności „fall through”: tylko pasująca klauzula case jest wykonywana (lub żadna, jeśli żadna nie pasuje).

Dopasowywać można nie tylko wartości, ale i strukturę danej; co więcej, jest też możliwe przypisywanie znalezionych elementów struktury do zmiennych, które mogą być potem użyte w klauzuli case.
W przykładzie poniżej, kolejne klauzule odpowiadają danym o różnej długości (na przykład listom czy krotkom). Zauważmy notację użytą w linii 11: symbol `*d` pozwala użyć zmiennej `d` jako sekwencji zawierającej wszystkie pozostałe elementy dopasowanej listy czy krotki (z wyjątkiem pierwszego, który został przypisany do `c`)

Listing 12 AAT-MatchLen/MatchLen.py

```
1  #!/usr/bin/env python
2
3  tuples = [ (1, 2, 3), ['a', 'b'], (4, 5, 6, 7, 8), "One" ]
4
5  for item in tuples:
6      match item:
7          case [a1, a2]:
8              print('Two elements', a1, a2)
9          case [b1, b2, b3]:
10             print('Three elements', b1, b2, b3)
11         case [c, *d]:
12             print('Many elements', c, end=' ')
13             for x in d:
14                 print(x, end=' ')
15             print()
16         case e:
17             print('One element', e)
```

Program drukuje

```
Three elements 1 2 3
Two elements a b
Many elements 4 5 6 7 8
One element One
```

W klauzulach case użyliśmy tu notacji listowej (z nawiasami kwadratowymi), ale równie dobrze mogliśmy użyć, zamiast, na przykład, `case [a1, a2]:`, notacji „krotkowej” `case (a1, a2):` lub `case a1, a2:` bez żadnych nawiasów.

Możemy też dopasowywać strukturę i wartości jednocześnie. W przykładzie poniżej pierwszy element `cmd` jest przyrównywany do podanego dosłownie napisu, podczas gdy pozostałe elementy są przypisywane do zmiennych, których potem używamy w odpowiedniej klauzuli case

Listing 13 AAV-MatchVS/MatchValStr.py

```
1  #!/usr/bin/env python
2
```

```
commands = (
              ('store', 'MyTable', 'John', 'Doe', 123),
              ('send',  'Hello John', 'john@somewhere'),
              ('print', 'Hello, World!'),
              'Error no 42'
            )

def printMessage(message):
    print('Printing message:', message)

def sendMessage(message, address):
    print("Sending '" + message + "' to", address)

def storeInDB(database, *args):
    print('Storing in DB:', end=' ')
    for a in args:
        print(a, end=' ')
    print()

def printError(error):
    print("ERROR: couldn't process '" + error + "'")


for cmd in commands:
    match cmd:
        case "print", message:
            printMessage(message)
        case ("send", message, address):
            sendMessage(message, address)
        case ["store", database, *data]:
            storeInDB(database, *data)
        case someError:
            printError(someError)
```

Program drukuje

```
Storing in DB: John Doe 123
Sending 'Hello John' to john@somewhere
Printing message: Hello, World!
ERROR: couldn't process 'Error no 42'
```

W ostatnim przykładzie jedna klauzula **case** „łapie” wszystkie dwuelementowe sekwencje przypisując oba jej elementy do zmiennych, których następnie używamy aby dodać dane do odpowiednich kolekcji; cokolwiek, co nie jest dwuelementową sekwencją jest traktowane inaczej:

Listing 14 AAX-MatchSN/MatchSeqNam.py

```
#!/usr/bin/env python

```

```
fruit   = {}       # empty dictionary
veggies = dict()   # also empty dictionary
others  = dict()
noprice = set()

frunames = ["Apple", "Pear"]
vegnames = ["Carrot","Beetroot" ]

shopping = [
             ("Carrot", 4), ("Pear", 7),   "Hammer",
             ("Tulip", 6),  ("Carrot", 5), ("Pear", 3),
             "Chocolate",   ("Apple", 8),  ("Carrot", 4),
             ("Apple", 2),  "Laptop",      ("Beetroot", 1)
           ]

for item in shopping:
    match item:
        case [fruveg , price]:
            aux = (fruit if fruveg in frunames
                         else veggies if fruveg in vegnames
                                      else others)
            if fruveg not in aux: aux[fruveg] = 0
            aux[fruveg] += price
        case justName:
            noprice.add(justName)

print('Fruit:   ', fruit)
print('Veggies: ', veggies)
print('Others:  ', others)
print('No price:', noprice)
```

Program drukuje

```
Fruit:    {'Pear': 10, 'Apple': 10}
Veggies:  {'Carrot': 13, 'Beetroot': 1}
Others:   {'Tulip': 6}
No price: {'Laptop', 'Hammer', 'Chocolate'}
```

Więcej szczegółów można znaleźć w PEP 636.[1]

[1] http://peps.python.org/pep-0636

Rozdział 3

Podstawowe typy danych

Podstawowe typy danych wbudowane w sam język Python to

- boolean
- liczby (całkowite, zmiennoprzecinkowe i zespolone). Są też dostępne inne przydatne typy numeryczne (ale już nie wbudowane): **fraction.Fraction** opisująca liczby wymierne i **decimal.Decimal** do operacji na liczbach zmiennoprzecinkowych z dowolną precyzją;
- typy sekwencyjne (listy, krotki, napisy, typy **bytes**, **bytearray**, **memoryview**);
- zbiory;
- słowniki (mapy).

Zauważmy, że nie ma tu klasycznych tablic, za to są listy, zbiory i mapy (zwane w Pythonie słownikami), które w innych językach należą raczej do bibliotek standardowych, a nie do samego języka.

Jako że wszystko w Pythonie jest obiektem (z konieczności jakiegoś typu), są też typy opisujące

- moduły,
- klasy i instancje klas (obiekty),
- funkcje,
- metody klas,
- opisujące kod funkcji,
- opisujące obiekty typów,
- obiekt **None**,
- obiekt **Ellipsis**,
- obiekt **NotImplemented**,
- obiekty używane wewnętrznie przez interpreter.

Wiele wbudowanych typów (**int**, **float**, **str**) jest wyposażonych w konstruktory konwertujące, na przykład `int('123')` zwraca liczbę 123 (**int**) a `str(123)` daje napis `'123'` (**str**).

3.1 Typ NoneType

Jest to specjalny typ opisujący „nic” — istnieje tylko jeden obiekt (singleton) tego typu o nazwie **None**. Nie ma on żadnych atrybutów i jego wartość w kontekście logicznym jest **False**. Referencja do **None** jest zwracana przez funkcje, które niczego jawnie nie zwracają, a więc odpowiadają funkcjom typu **void** w takich językach jak C/C++/Java.

```
>>> type(None)
<class 'NoneType'>
>>> a = print()

>>> b = print()

>>> type(a), type(b)
(<class 'NoneType'>, <class 'NoneType'>)
```

```
>>> id(a), id(b)
(94793807807456, 94793807807456)
>>> if a:
...     print('true')
... else:
...     print('false')
...
false
```

Wartość None jest często używana jako wartość domyślna argumentów opcjonalnych, dzięki czemu funkcja może łatwo rozpoznać, czy argument ten został podany czy nie.

3.2 Typ ellipsis

Również typ specjalny z jednym singletonowym obiektem o nazwie Ellipsis lub, równoważnie, ... (trzy kropki). Obiekt ten jest używany prawie wyłącznie w pakietach numerycznych przy definiowaniu wycinków wielowymiarowych macierzy. W kontekście logicznym jego wartością jest True.

```
>>> type(Ellipsis)
<class 'ellipsis'>
>>> type(...)
<class 'ellipsis'>
>>> id(Ellipsis), id(...)
(9422880, 9422880)
>>> if Ellipsis: print('T')
... else:        print('F')
...
T
```

3.3 Typ NotImplementedType

Typ specjalny z jednym singletonowym obiektem o nazwie NotImplemented. Obiekt ten używany jako wartość zwracana przez metody specjalne, które powinny istnieć, ale nie zostały jeszcze zaimplementowane. Jest to jedyny obiekt, którego nie zaleca się (i jest to traktowane jako przestarzałe – *obsolete*) używać w kontekście logicznym.

3.4 Typy numeryczne

Typy numeryczne opisują liczby całkowite i zmiennoprzecinkowe. Zalicza się do nich też typ **bool**, ponieważ dziedziczy z **typ** i (niestety) *może* być używany w kontekście liczbowym.

Generalnie Python raczej niechętnie dokonuje niejawnych konwersji między typami. Jednakże, tak jak w większości innych języków, wartości numeryczne różnych typów mogą „mieszane” w operacjach arytmetycznych. Dodając wartość całkowitą do zmiennoprzecinkowej dostaniemy wartość zmiennoprzecinkową, (jako „szerszą” od **int**)

```
>>> a = 2; b = 2.5
>>> a+b, type(a+b)
(4.5, <class 'float'>)
```

Podobnie, **complex** jest „szerszy” od **float**, więc w operacjach z liczbami zespolonymi wartości zmiennoprzecinkowe (**float**) te ostatnie są promowane do zespolonych (**complex**):

```
>>> a = 1; b = 1.5; c = (1 + 1j)
>>> d = a + b + c;
>>> d, type(d)
((3.5+1j), <class 'complex'>)
```

Aby ułatwić takie mieszane operacje z wartościami różnych typów (**int**, **float**, **complex**), wartości całkowite i zmiennoprzecinkowe mają, jako swoje atrybuty, części rzeczywiste (real) i urojone (imag) — ich wartości są oczywiście zerowe:

```
>>> a = 7
>>> a, type(a), a.real, a.imag
(7, <class 'int'>, 7, 0)
>>> b = 3.75
>>> b, type(b), b.real, b.imag
(3.75, <class 'float'>, 3.75, 0.0)
```

Konstruktory typów numerycznych mogą być użyte do jawnych konwersji z innych typów liczbowych i napisów.
Konwersja **float**→**int** zaokrągla w kierunku zera:

```
>>> a = 3.9; b = -3.9
>>> int(a), int(b)
(3, -3)
```

Można też konwertować napisy do liczb:

```
>>> int('123'), float('-1.2e+2'), complex('2-3j')
(123, -120.0, (2-3j))
```

Zauważmy, że w napisie dla liczb zespolonych nie może być spacji:

```
>>> complex('1 + 1j')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: complex() arg is a malformed string
```

3.4.1 Liczby całkowite

Istnieje tylko jeden typ całkowity — **int**. Jednakże, wartości reprezentowalnych licz całkowitych nie są w zasadzie ograniczone: Python w sposób niewidoczny dla nas znajdzie odpowiednią wewnętrzną reprezentację dla dowolnie dużych wartości; na przykład 33^{33} będzie policzone prawidłowo, dając

$$129110040087761027839616029934664535539337183380513$$

(operowanie na takich dużych liczbach może jednak być nieefektywne).
Literały liczb całkowitych mogą być poprzedzone digrafem wskazującym na użytą podstawę systemu liczbowego: `0b` lub `0B` dla binarnego, `0o` lub `0O` dla ósemkowego, `0x` lub `0X` dla szesnastkowego. For example,

$$47 \equiv \mathrm{0b101111} \equiv \mathrm{0o56} \equiv \mathrm{0x2F}.$$

Jak w Javie, podkreślniki mogą być używane do oddzielenia grup cyfr dla zwiększenia czytelności

```
>>> 1_234_567
1234567
```

Klasa **int** ma konstruktory pozwalające na konwersje z napisów (ale też z obiektów typu **bytes** czy **bytes**). Drugi, opcjonalny argument oznacza podstawę systemu liczbowego (wartość domyślna to oczywiście 10):

```
>>> int('1D', 16)
29
>>> int('1D', 16), int('0x1D', 16)
(29, 29)
>>> int('57', 8), int('0o57', 8)
(47, 47)
>>> int('11001',2), int('0b11001', 2)
(25, 25)
>>> int('123', 10), int('123')
(123, 123)
>>> int('2Z', 36)
107                  # 2*36 + 35
```

Są też wbudowane funkcje **bin**, **oct** i **hex** do konwersji **int** → **str** w których wynikowy napis reprezentuje daną liczbę przy podstawach 2, 8 lub 16 (reprezentacja dziesiętna może być uzyskana przez użycie konstruktora **str**):

```
>>> a = bin(0x8E)
>>> a, type(a)
('0b10001110', <class 'str'>)
>>>
>>> b = oct(0xF)
>>> b, type(b)
('0o17', <class 'str'>)
>>>
>>> c = hex(255)
>>> c, type(c)
('0xff', <class 'str'>)
>>>
>>> d = str(123)
>>> d, type(d)
('123', <class 'str'>)
```

Implementacja wartości typu **int** jest w Pythonie bardzo szczególna. Wszystkie liczby całkowite, małe i duże, dodatnie i ujemne, są tego samego typu **int**; nie ma typów **long**, **short**, **unsigned**, itd.[1]

Obiekty reprezentujące małe liczby całkowite, przynajmniej w zakresie $[-5, 256]$, są tworzone raz i zapamiętywane (*cached*). Referencje do nich wskazują zatem na dokładnie te same obiekty:

```
>>> a = -2; b = 0b1011 - 0xD; c = -4000//2000
>>> a, b, c
(-2, -2, -2)
>>> id(a), id(b), id(c)
(9801152, 9801152, 9801152)
```

[1] Patrz jednak rozdz. ??, str. ??.

Wewnętrznie wartości całkowite są reprezentowane jako C-struktury **PyLongObject** zawierające, prócz innych informacji,[1]

- wskaźnik do tablicy ob_digit „cyfr", które są liczbami typu **uint32_t**;
- wymiar tej tablicy ob_size.

Dla liczb mniejszych od 2^{30} tablica ma jeden element, który jest interpretowany jako bezznakowa 32-bitowa liczba. Dla większych liczb, elementy tablicy („cyfry") traktowane są jako współczynniki przy kolejnych potęgach 2^{30} i same mają wartości z przedziału $[0, 2^{30} - 1]$. Na przykład

$$39854065135908438582144781 5 = 1234567 + 23456789 * 2^{30} + 345678912 * 2^{30*2}$$

więc dla tej liczby ob_digit jest $[1234567, 23456789, 345678912]$ a ob_size wynosi 3.

Liczby ujemne są reprezentowane tak samo, a ich znak sygnalizowany jest przez ujemny znak ob_size. Jeśli ob_size jest 0, to liczba też jest 0.

Z powodu tej skomplikowanej implementacji obiekty typu **int** są stosunkowo duże i liczba zajmowanych bajtów zależy od wielkości liczby. Małe liczby zajmują w sumie 28 bajtów (3,5 razy więcej niż **long** w C/C++/Javie). Tak jest dla liczb dla których ob_size jest 1 (czyli mniejszych od 2^{30}). Dla większych liczb wymiar tablicy ob_digit rośnie i obiekty stają się coraz większe:

```
>>> a = 1
>>> a, type(a), a.__sizeof__()
(1, <class 'int'>, 28)
>>>
>>> a = 2**30-1
>>> a, type(a), a.__sizeof__()
(1073741823, <class 'int'>, 28)  # still one element
>>>
>>> a = a + 1
>>> a, type(a), a.__sizeof__()
(1073741824, <class 'int'>, 32)  # two elements needed
>>>
>>> a = (2**30-1)*2**30 + (2**30 - 1)
>>> a, type(a), a.__sizeof__()
(1152921504606846975, <class 'int'>, 32) # still two elements
>>>
>>> a = a + 1
>>> a, type(a), a.__sizeof__()
(1152921504606846976, <class 'int'>, 36) # now three needed
```

Oczywiście są algorytmy dzięki którym operacje na **int**ach są w miarę szybkie.

3.4.2 Typ logiczny

Istnieje w Pythonie osobny typ logiczny **bool**, który dziedziczy z **int**. Istnieją dwie wartości tego typu reprezentowane przez literały **True** i **False**.

It has only two values: **True** and **False**. Jednakże, jak w wielu innych językach (ale nie w Javie!), wartości innych typów mogą być używane w kontekstach wymagających wartości i wtedy następujące obiekty

- wartości numeryczne równe 0,

[1] Dla platform 64 bitowych.

- puste sekwencje i kolekcje (listy, zbiory, słowniki, napisy, krotki),
- obiekt **None**,
- stała **False**

są interpretowane jak **False** („*falsy*”) w kontekstach logicznych. Wszystkie inne obiekty są traktowane jak **True** (*truthy*). Jedynym obiektem, którego nie zaleca się (choć wciąż jest to dozwolone) używać w kontekście logicznym jest singleton **NotImplemented** (który odpowiada **True**).

Jeśli rzeczywiście chcemy mieć zmienną typu **bool**, możemy użyć konstruktora konwertującego tej klasy:

```
>>> a = bool("Hello")
>>> a, type(a)
(True, <class 'bool'>)
>>> bool({})
False
>>> bool(0)
False
>>> bool(78)
True
>>> bool(NotImplemented)
<stdin>:1: DeprecationWarning: NotImplemented should
           not be used in a boolean context
True
>>> bool(Ellipsis)
True
>>> bool([[]]) # non-empty list: contains one element!
True
```

Niestety, obiekty typu **bool** są reprezentowane przez liczby całkowite 0 i 1, więc formalnie *są* liczbami (choć nie powinny...). Na przykład,

```
>>> True/3
0.3333333333333333
>>> (7-False)/(True+1)
3.5
```

Fakt, że klasa **bool** dziedziczy z **int** ma czasem niespodziewane konsekwencje; na przykład

```
>>> True == 2-1
True
>>> True == 3-1
False
```

i nie jest możliwe użycie jednocześnie **True** i 1 w zbiorze lub jako klucza w słowniku.

Logiczna alternatywa i koniunkcja są oznaczane przez słowa kluczowe **or** oraz **and**. Tak jak w innych językach wartościowanie jest „skrócone” — jeśli wynik jest znany po obliczeniu lewego operandu, prawy nie jest w ogóle obliczany; będzie to miało miejsce dla alternatywy gdy lewy operand jest **True**, a dla koniunkcji gdy jest on **False**.
Negacja logiczna jest oznaczana przez słowo kluczowe **not**.

Zauważmy tu, że znane z innych języków operatory **||**, **&&** i **!** w Pythonie *nie* będą działać!

3.4.3 Liczby zmiennoprzecinkowe

Jest tylko jeden typ zmiennoprzecinkowy – **float**. Odpowiada on typowi **double** w C/C++/Javie. Wartość tego typu zajmuje 8 bajtów (64 bity): jeden bit znaku, 11 bitów cechy (eksponenty) i 52 bity mantysy. Daje to precyzję 53 cyfr binarnych, bo w systemie binarnym pierwszą (najbardziej znaczącą) cyfrą jest zawsze 1, więc nie ma po co fizycznie tej jedynki zapisywać (szczegóły można znaleźć na przykład tutaj[1]).

Pamiętać należy, że liczby zmiennoprzecinkowe są prawie zawsze *przybliżeniami*; ponieważ są reprezentowane jako skończona sekwencja bitów, nawet tak „niewinna" liczba jak 0,1 nie ma dokładnej reprezentacji. Tylko ułamki wymierne z pełną potęgą dwójki w mianowniku mogą mieć skończoną reprezentację w układzie dwójkowym:

```
>>> v0100 = 0
>>> v0125 = 0
>>> for i in range(1_000_000):
...     v0100 += 0.100
...     v0125 += 0.125
>>> v0100 - 100_000
1.3328826753422618e-06
>>> v0125 - 125_000
0.0
```

W drugim przypadku dodajemy 0,125, czyli $1/2^3$, a ta liczba akurat jest reprezentowana dokładnie, więc suma miliona takich liczb jest dokładnie 125 000. Jednakże 0,1 jest reprezentowana tylko w przybliżeniu, więc suma miliona tych liczb nie jest dokładnie równa 100 000. Możemy to zobaczyć drukując 0,1 jako obiekt typu **Decimal**:

```
>>> import decimal
>>> decimal.Decimal(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

Ten wynik jest *dokładny* — jest to, w układzie dziesiętnym, dokładnie liczba reprezentowana przez *przybliżenie* wartości 0,1 w układzie binarnym z 53 dwójkowymi cyframi znaczącymi; liczba 0,1 nie ma bowiem skończonej reprezentacji dwójkowej, tak jak nie ma jej $1/3$ w układzie dziesiętnym.

Czasami, szczególnie w obliczeniach numerycznych, mamy do czynienia z nieskończonościami (Inf) lub wartościami not-a-number (NaN). Są one również zdefiniowane w Pythonie:

```
>>> a = float('nan')
>>> b = float('nan')
>>> c = a + b
>>> a == b, a, b, c
(False, nan, nan, nan)
```

Jak widzimy, NaN porównywany z czymkolwiek, nawet z samym sobą, daje **False**. Każda operacja arytmetyczna z udziałem NaN daje też NaN.

Popatrzmy teraz na nieskończoności:

```
>>> x = float('inf')
>>> y = float('-inf')
>>> x, y, x + 100, x - y, x + y
(inf, -inf, inf, inf, nan)
```

[1] https://en.wikipedia.org/wiki/IEEE_floating_point

Dodawanie/odejmowanie skończonych liczb do/od ±Inf daje ±Inf; również nieskończoności tego samego znaku dodają się do nieskończoności; jednakże odejmowanie nieskończoności (czyli dodawanie nieskończoności różnych znaków) nie jest matematycznie określone i daje NaN.

3.4.4 Liczby zespolone

Literały typu **complex** zapisuje się z literą 'j' (albo 'J', ale nie 'i' jak w matematyce) dołączoną do części urojonej.
Na przykład `0.5*(1+1j)**2` daje `1j`. Części rzeczywista i urojona są reprezentowane jako 64-bitowe liczby typu **float** i są niemodyfikowalnymi atrybutami real i imag liczby zespolonej:

```
>>> z = 3+4j
>>> z.real
3.0
>>> z.imag
4.0
>>> z.imag = 8
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
  AttributeError: readonly attribute
```

Funkcja **abs** (wbudowana, nie potrzebujemy żadnych importów) działa też dla liczb zespolonych; jest też moduł ***cmath***, który zawiera wiele funkcji operujących na liczbach zespolonych (jak **phase**, **exp**, **log**, **sqrt**, funkcje trygonometryczne i hiperboliczne, itd.). Można mieszać liczby zespolone z **float**ami i **int**ami w operacjach matematycznych: jeśli jest choć jedna liczba zespolona, to pozostałe też będą promowane do tego typu i wynik również będzie zespolony (być może z częścią urojoną 0):

```
>>> z = 1 + 1j
>>> import cmath
>>> 2 + cmath.log(z)/1.5
(2.2310490601866486+0.5235987755982988j)
>>> from cmath import exp as cexp
>>> z - cexp(cmath.log(z))
2.220446049250313e-16j       # essentially 0
```

3.5 Napisy (wprowadzenie)

Napisy są w Pythonie reprezentowane obiektami typu **str**. Literały tego typu mogą być zapisywane na kilka sposobów. Można je umieszczać w pojedynczych apostrofach lub znakach cudzysłowu. Jeśli użyliśmy apostrofów, to wewnątrz cudzysłowy są po prostu cudzysłowami i nie muszą być poprzedzane wstecznym ukośnikiem (i na odwrót, jeśli użyliśmy zewnętrznych cudzysłowów). Tak jak w C/C++, sąsiadujące literały napisowe są konkatenowane i są traktowane jako jeden napis:

```
>>> "That's OK, " 'double quote doesn\'t have to be "escaped" here'
'That\'s OK, double quote doesn\'t have to be "escaped" here'
```

Literał napisowy może zawierać dowolne znaki Unicodu jeśli tylko znamy ich kod: notacja `\uxxxx` jest używana gdy podajemy 16-bitowy kod (cztery cyfry szesnastkowe) a `\Uxxxxxxxx` gdy podajemy 32-bitowy kod (osiem cyfr szesnastkowych) — każdy 'x' oznacza tu jedną cyfrę szesnastkową.

```
>>> '\u00a9 \u017b\u00f3\u0142w'
'© Żółw'
```

Tak jak w innych językach, niektóre specjalne znaki można wprowadzić poprzedzają jednoznakowy symbol wstecznym ukośnikiem:

- `\a` znak alarmu (bell - BEL);
- `\b` znak cofania (backspace - BS);
- `\f` znak nowej strony (formfeed - FF);
- `\n` znak nowej linii (new line, line feed - LF);
- `\r` znak powrotu karetki (carriage return - CR);
- `\t` znak tabulacji poziomej (horizontal tab - HT);
- `\v` znak tabulacji pionowej (vertical tab - VT);
- `\'` znak apostrofu (single quote, apostrophe);
- `\"` znak cudzysłowu (double quote).

Można też ująć literał napisowy *potrójnymi* apostrofami lub cudzysłowami W takich literałach apostrofy i cudzysłowy są normalnymi znakami i nie wymagają wstecznych ukośników; również znaki nowej linii są zachowywane

```
>>> """That's OK, quotes don't
... have to be "escaped" here"""
'That\'s OK, quotes don\'t\nhave to be "escaped" here'
```

Dodając literę 'r' (lub 'R') przed otwierającym delimiterem (dowolnym: apostrofem, cudzysłowem, pojedynczym lub potrójnym) otrzymujemy tak zwany *surowy* napis (*raw string*). W takich napisach ukośniki wsteczne oznaczają same siebie, czyli na przykład `r"\t\v\b"` zawiera 6 znaków, trzy z nich to wsteczne ukośniki, a pozostałe trzy to litery (bez tego 'r' byłby to napis zawierający trzy znaki: tabulator, tabulator pionowy i znak cofania).
Surowe napisy są szczególnie użyteczne przy definiowaniu wyrażeń regularnych, które zwykle zawierają zatrzęsienie wstecznych ukośników.

Ważne: Tak jak w Javie, napisy (obiekty typu `str` są niemodyfikowalne. Nie jest możliwa modyfikacja takiego obiektu; funkcje, które wydają się to robić zawsze tworzą nowe obiekty na podstawie istniejących:

```
>>> s = "Alice"
>>> id(s)
139733635023176
>>> s = s.lower()
>>> s, id(s)
('alice', 139733635017168)
```

Rozdział 4

Operatory

Jak wiemy z innych języków, takich jak C/C++/Java, operatory są w istocie funkcjami w przebraniu; wywołujemy je nie w „normalny” sposób, po nazwie i z argumentami w nawiasach, ale poprzez symbole (*, &, +, itd.) umieszczane przed albo pomiędzy operandami (argumentami). Wiemy również, że istnieją różne typy operatorów: unarne (jednoargumentowe, np. ! czy ~), binarne (dwuargumentowe, np. +, %), a nawet ternarne (trójargumentowe, np. ?:).

W tym rozdziale omówimy operatory występujące w Pythonie, ich priorytety i cechy. Najpierw jednak parę słów o przypisaniu, które w Pythonie operatorem *nie* jest.

4.1 (Nie)operator przypisania

Operatory, jedno- czy dwuargumentowe, po zaaplikowaniu do argumentów zawsze, z definicji, zwracają wartość. Jednakże w Pythonie operator ten jest instrukcją, i przypisanie nie jest wyrażeniem

```
>>> a = (b = 2)              # OK in C/C++/Java
File "<stdin>", line 1
    a = (b = 2)
        ^
SyntaxError: invalid syntax
```

Jak z tego wynika, przypisanie, =, jak również jego wersje złożone, jak +=, *= itd, *nie* są w Pythonie operatorami, jak w C/C++/Javie, w których przypisanie jest „normalnym”, dwuargumentowym, wiązanym od prawej operatorem.

Co może jednak być zaskakujące, `a = b = 2` *jest* prawidłowe. Przypisanie łańcuchowe jest w Pythonie traktowane specjalnie: najpierw wyrażenie po prawej stronie jest obliczane (musi to być wyrażenie!), a następnie obliczona wartość jest przypisywana do zmiennych występujących przed tym wyrażeniem *od lewej do prawej* (sic!):

```
>>> b = [3, 3]
>>> a = b[a] = b[a-1] = len(b)-1
>>> b
[1, 1]
```

Jak widzimy, gdy wartość `len(b)-1`, czyli 1, jest przypisywana do `b[a]`, zmienna `a` już istnieje i ma wartość 1. Tak więc, koncepcyjnie

$$\text{target}_1 = \text{target}_2 = \text{target}_3 = \text{expr}$$

jest równoważne obliczeniu expr i serii przypisań

$$\begin{aligned} \text{target}_1 &= \text{expr} \\ \text{target}_2 &= \text{expr} \\ \text{target}_3 &= \text{expr} \end{aligned}$$

w tej kolejności.

Jednak,

```
>>> x = y = z += 2
SyntaxError: invalid syntax
```

nie zadziała, bo `z += 2` *nie jest* wyrażeniem — nie ma wartości.

4.2 „Prawdziwy” operator przypisania

W Python 3.8 został dodany do języka „prawdziwy” operator przypisania (wyrażenie przypisania). Znaczy to, że przypisanie takie ma typ i wartość lewej strony po przypisaniu, jak w C/C++/Javie.
Operator oznaczany jest symbolem := symbol[1] i ma formę `name := expr`, gdzie `name` jest identyfikatorem, a `expr` dowolnym wyrażeniem (ale nie krotką nieujętą w nawiasy).

Taka forma przypisania upraszcza często kod i czyni go czytelniejszym.

Na przykład, załóżmy, że wczytujemy od użytkownika dane aż do wpisania przez niego `"quit"` (i dodajemy wczytane liczby do listy). Mogłoby to wyglądać tak:

```
numbers = []
inp = input("Enter number (or 'quit'): ")
while inp != "quit":
    numbers.append(int(inp))
    inp = input("Enter number (or 'quit'): ")
```

To będzie działać, ale jest brzydkie i funkcja **input** pojawia się dwa razy. Możemy to samo zrobić lepiej, używając operatora := (zwanego *walrus*, czyli po polsku *mors*). To odpowiada przypisaniu jakie znamy z C/C++/Javy; jest to wyrażenie, które ma wartość:

```
numbers = []
while (inp := input("Enter number (or 'quit'): ")) != "quit":
    numbers.append(int(inp))
```

Wyrażenie przypisania używane jest do nadania nazwy pośrednim wynikom, aby można ich było później użyć bez potrzeby liczenia ich wartości jeszcze raz, nawet w tej samej instrukcji:

```
s = 'rotator'
if (r := s[::-1]) == s:
    print('Palindrom')
else:
    print('Not palindrom -> ' + r)
print('r is ' + r)
```

Program drukuje

```
Palindrom
r is rotator
```

co pokazuje inną ważną własność Pythona: chociaż zmienna `r` utworzona została wewnątrz instrukcji **if**, jest widoczna za całą tą instrukcją (patrz rozdz. 5.1, str. 63).

Również w składaniu sekwencji (rozdz. 6.1, str. 93) takie przypisania mogą być użyteczne; na przykład

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

print([(n, r) for n in range(24,31) if (r := fib(n)) % 2 != 0])
```

[1] Zwanym „walrus” (czyli *mors*).

drukuje nieparzyste liczby Fibonacciego F_n dla $n \in [24, 30]$

```
[(25, 75025), (26, 121393), (28, 317811), (29, 514229)]
```

W dalszym ciągu wykładu napotkamy inne zastosowania tego operatora.

4.3 Priorytety operatorów

Priorytety operatorów w Pythonie są zaprezentowane w poniższej tabeli:

Tablica 2: Priorytety operatorów

Operator(y)	Opis
(wyrażenia...), [wyrażenia...], {klucz: wartość...}, {expressions...}	Wyrażenia nawiasowe: literały krotek, list, słowników, zbiorów
x[i], x[i:j], x[i:j:ks], x(argumenty...), x.atrybut	Indeksowanie, wycinki, operator wywołania, odwołanie do atrybutu
await x	Wyrażenie await
**	Potęgowanie
+x, −x, ~x	jednoargumentowe +/-, bitowe NOT
*, @, /, //, %	Mnożenie, mnożenie macierzowe, dzielenie, dzielenie całkowitoliczbowe, modulo
+, −	Dodawanie, odejmowanie
<<, >>	Bitowe przesunięcia
&	Bitowe AND
^	Bitowe XOR
|	Bitowe OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Porównania, testy przynależności i identyczności
not	Logiczne NOT
and	Logiczne AND
or	Logiczne OR
if-else	Wyrażenie warunkowe
lambda	Wyrażenie lambda
:=	Operator przypisania

4.4 Parenthesized expressions

Wyrażenia nawiasowe tworzą listy (`[ expressions... ]`) (rozdz. **??**, str. **??**), zbiory (`{ expressions... }`) (rozdz. 7.3, str. 130), słowniki (`{ key:value... }`) (rozdz. 7.4, str. 137):

```
>>> a = [1, 2, 'Hello']
>>> a, type(a)
([1, 2, 'Hello'], <class 'list'>)
>>> b = {1, 2, 'Hello'}
```

```
>>> b, type(b)
({1, 2, 'Hello'}, <class 'set'>)
>>> c = {1 : 'un', 3 : 'trois', 'un' : 1, 'trois' : 3}
>>> c, type(c)
({1: 'un', 3: 'trois', 'un': 1, 'trois': 3}, <class 'dict'>)
```

Zamiast definiowania sekwencji elementów, możemy też użyć tzw. składania (*comprehension*) dla list (rozdz. 6.1.1, str. 94), zbiorów (rozdz. 6.1.2, str. 97), słowników (rozdz. 6.1.3, str. 98) i generatorów (rozdz. 6.2.1, str. 102):

```
>>> a = [x*x for x in range(1, 11) if x%2 != 0]
>>> a, type(a)
([1, 9, 25, 49, 81], <class 'list'>)
>>>
>>> b = {x for x in dir(a) if x[:1] == 'c'}
>>> b, type(b)
({'count', 'copy', 'clear'}, <class 'set'>)
>>>
>>> c = {x : len(x) for x in b}
>>> c, type(c)
({'count': 5, 'copy': 4, 'clear': 5}, <class 'dict'>)
>>>
>>> d = (x for x in range(1,101) if x**2 % 10 == 6)
>>> d, type(d)
(<generator object <genexpr> at 0x7f52a900>, <class 'generator'>)
>>> for x in d:
    print(x, end=' ')

4 6 14 16 24 26 34 36 44 46 54 56 64 66 74 76 84 86 94 96
```

4.5 Indeksowanie i wycinki

4.5.1 Indeksowanie

Indeksowanie w Pythonie jest takie jak w C/C++/Javie — indeksy podajemy w nawiasach kwadratowych i zaczynamy od zera. Indeksować można cokolwiek, co ma metodę **__getitem__** — w szczególności obiekty typów sekwencyjnych jak **list**, **tuple**, **range** a także **str**.

```
>>> ls = [1, 2, '3', 'four', ('Hello', 'World')]
>>> ls[0]
1
>>> ls[3]
'four'
>>> ls[4]
('Hello', 'World')
>>> ls[4][1]
'World'
```

To wszystko jest bez niespodzianek, ale Python ma więcej do zaoferowania. Czasem jest znacznie wygodniej mówić o pozycji elementu licząc od końca, a nie od początku; na przykład *daj mi element trzeci od końca*. Zamiast liczyć jakiemu indeksowi to odpowiada, możemy użyć indeksów *ujemnych*.

Indeks -1 odpowiada *ostatniemu* elementowi, czyli temu o dodatnim indeksie $-1+$leng, gdzie leng jest długością (liczbą elementów) sekwencji. Index -2 odpowiada elementowi przedostatniemu, i tak dalej. Ogólnie zatem, ujemny indeks $-n$ odpowiada dodatniemu indeksowi $-n+$leng. Na przykład napis `'PYTHON'` ma długość 6 i jego elementy (znaki) mogą być indeksowane tak:

```
 0   1   2   3   4   5
 P   Y   T   H   O   N
-6  -5  -4  -3  -2  -1
```

Tak więc legalne indeksy to

$$-\text{leng}, -\text{leng}+1, \ldots, -1, 0, 1, \ldots, \text{leng}-1$$

co możemy zilustrować następującym fragmentem kodu:

```
>>> s = 'PYTHON'
>>> s, len(s)
('PYTHON', 6)
>>> s[0]
'P'
>>> s[5]
'N'
>>> s[-1]
'N'
>>> s[-6]
'P'
>>> s[6]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
>>> s[-7]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Częstym zadaniem jest znalezienie indeksu elementu sekwencji (listy, krotki, napisu, ...). Istnieje zatem przydatna metoda **index**, która pozwala to zrobić; znajduje ona indeks pierwszego wystąpienia danego elementu):

```
>>> a = (1, 4, 5, 8)     # tuple
>>> b = [1, 4, 5, 8]     # list
>>> c = '1458'           # string
>>> a.index(5), b.index(5), c.index('5')
(2, 2, 2)
```

Można dodać dodatkowy argument mówiący od którego indeksu rozpoczynamy poszukiwanie

```
>>> a = [5, 6, 7, 5, 6, 7]
>>> a.index(5, 2)
>>> 3
```

Rezultatem jest 3, bo rozpoczęliśmy poszukiwanie od indeksu 2 (elementu o wartości 7). Opcjonalnie, można dodać jeszcze trzeci argument: *przed* którym elementem zakończyć poszukiwanie, a więc element o tym indeksie już nie należy do zakresu przeszukiwania:

```
>>> a = [5, 6, 7, 8, 6, 4, 6]
>>> a.index(6, 2, 5)
4
```

Tu element o wartości 6 był poszukiwany dla przedziału indeksów $[2, 6)$, czyli $2, 3, 4, 5$ (bez 6). Jeśli element o podanej wartości nie występuje, podnoszony jest wyjątek **ValueError**:

```
>>> a = [1, 2, 4, 5, 3]
>>> a.index(3)
4
>>> a.index(3, 0, 4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 3 is not in list
```

Wystąpił tu wyjątek, bo wartości 3 nie ma w zakresie indeksów $[0, 4)$, który nie zawiera indeksu 4. Aby uniknąć wyjątków, można sprawdzić czy dany element w ogóle występuje w sekwencji za pomocą operatorów **in** i **not in**:

```
>>> a = [1, 2, 3, 4]
>>> b = 'Hello'
>>> 2 in a
True
>>> 5 in a
False
>>> 5 not in a
True
>>> 'el' in b
True
```

Analogiczna metoda **find**, działa tylko dla napisów i obiektów typu **bytes** (see rozdz. **??**, str. **??**); nie zgłasza ona wyjątku tylko zwraca -1 w przypadku nieznalezienia elementu:

```
>>> s = 'abcdefg'
>>> s.find('d')
3
>>> s.find('h')
-1
>>> s.find('b', 2)     # start from index 2
-1
```

4.5.2 Wycinki

Wycinki (*slices*) są *kopiami* podciągów sekwencji (pobranie wycinka nie zmienia samej sekwencji).

Powiedzmy, że seq jest sekwencją o długości (liczbie elementów) leng. Wtedy po

```
sl = seq[beg:end:step]
```

sl będzie sekwencją *kopii* elementów (które, przypomnijmy, są referencjami!) sekwencji seq które odpowiadają indeksom beg, beg + step, beg + 2 ∗ step, *ldots* aż do, ale *bez*, end.

Na przykład, dla `[2:7:1]` będą to elementy o indeksach $2, 3, 4, 5, 6$, a dla `[2:7:2]` — $2, 4, 6$.
Jeśli step nie jest podana, jej wartością domyślną jest 1, więc `[2:5]` jest równoważne `[2:5:1]`. Można też opuścić wartość beg (ale nie następujący po niej dwukropek!) i/lub end. Wtedy, dla dodatnich wartości step:

- wartością beg jest 0, czyli `[:5]` jest równoważne `[0:5:1]`,
- wartością end jest długość sekwencji.

Na przykład:

```
>>> ls = ['a', 'b', 'c', 'd', 'e', 'f']
>>> ls[:3]                      # = ls[0:3:1]
['a', 'b', 'c']
>>> ls[1::2]                    # = ls[1:6:2]
['b', 'd', 'f']
>>> ls[:]                       # = ls[0:6:1]
['a', 'b', 'c', 'd', 'e', 'f']
>>> ls[2:6:2]
['c', 'e']
>>> ls[5:1:-1]
['f', 'e', 'd', 'c']
```

W ostatnim przykładzie step jest ujemne, ale zarówno beg jak i end są jawnie podane.

Ujemne wartości kroku step przy brakujących wartościach beg i/lub end nie są już tak intuicyjnie oczywiste. Wtedy

- beg wynosi domyślnie -1,
- end wynosi domyślnie `-leng-1`.

Na przykład dla napisu o długości 6 i ujemnym kroku, wartość domyślna end wynosi -7:

```
>>> s = 'abcdef'
>>> s[:-2:-1]     # -1
'f'
>>> s[-3:-6:-1]   # -3, -4, -5
'dcb'
>>> s[:-4:-1]     # -1, -2, -3
'fed'
>>> s[-2::-1]     # -2, -3, -4, -5, -6
'edcba'
>>> s[:-5:-2]     # -1, -3
'fd'
>>> s[::-2]       # -1, -3, -5
'fdb'
>>> s[::-1]       # -1, -2, -3, -4, -5, -6
'fedcba'
```

W komentarzach po symbolu # podane są indeksy jakie będą uwzględnione w wycinku; są one ujemne, ale pamiętamy, że odpowiadają dodatnim indeksom po dodaniu długości sekwencji (w naszym przypadku 6). Pamiętać też trzeba, że indeks end nigdy do wycinka *nie* wchodzi.

Ostatni przykład jest interesujący: jak widzimy, `[::-1]` daje kopię całej sekwencji, ale w odwróconym porządku![1]

Aby zrozumieć lepiej, jak to wszystko działa, można użyć obiektów typu **slice**. W zasadzie, jeśli napiszemy `seq[beg:end:step]`, to „pod spodem" obiekt typu **slice** jest tworzony i zastosowany do znalezienia „prawdziwych" (nieujemnych) indeksów. Możemy to zrobić „ręcznie" (argumenty **None** konstruktora znaczą „zastosuj wartość domyślną"):

```
>>> ls = [1, 2, 3, 4, 5]
>>> sl = slice(None, 3, 2)
>>> ls[sl]
[1, 3]
```

Jak zauważyliśmy, `seq[::-1]` zwraca kopię `seq` w odwrotnej kolejności. Zobaczmy jak to zostało policzone. Klasa **slice** ma metodę **indices**, której posyłamy długość sekwencji, a zwraca krotkę określającą „wyliczone" indeksy:

```
>>> sl = slice(None, None, -1) # # corresponds to [::-1]
>>> sl.indices(6)
(5, -1, -1)
>>> 'abcdef'[sl]
'fedcba'
```

Rezultat, $(5, -1, -1)$, mówi nam, że dla sekwencji o długości 6 indeksy biegną od 5 (ostatni element), co 1 w dół do -1, ale *bez* -1; innymi słowy będą 5, 4, 3, 2, 1, 0.

Podobnie:

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sl = slice(-2, 1, -2)
>>> sl.indices(9)
(7, 1, -2)            # 7 5 3
>>> a[sl]
[8, 6, 4]
>>> a[-2:1:-2]
[8, 6, 4]
```

Wyrażenia takie jak `[beg:end:step]` dają nam kopie (pod)sekwencji (w odwrotnym porządku, jeśli krok jest ujemny). Jest to często używane, ale są tutaj pułapki, których trzeba być świadomym. Rozważmy:

```
>>> ls = [1, ['a', 'b'], 'c']
>>> copy = ls[::]
>>> reve = ls[::-1]
>>> reve[1][1] = 'd'
>>> copy
[1, ['a', 'd'], 'c']
>>> reve
['c', ['a', 'd'], 1]
>>> ls
[1, ['a', 'd'], 'c']
```

[1] Zatem `if ls==ls[::-1]:...` sprawdza czy `ls` jest palindromem.

Listy copy i reve zawierają w szczególności *kopie* drugiego elementu oryginalnej listy (o indeksie 1). Ten element to jednak *referencja* do listy, a listy są modyfikowalne. Modyfikujemy tę listę (`reve[1][1] = 'd'`) używając reve, ale ls i copy wciąż zawierają tę samą referencję do zmienionej już listy. Dlatego wszystkie listy wydają się zmienione.

Możemy temu zaradzić, stosując *głęboką* kopię listy, która kopiuje referencje do obiektów niemodyfikowalnych, ale dla modyfikowalnych tworzy (rekurencyjnie, jeśli trzeba) nowe obiekty. W pakiecie ***copy*** są funkcje do obu tych operacji: program

```
import copy
a = [1, ['a', 'b'], 2]
b = copy.copy(a)
c = copy.deepcopy(a)
print(id(a[0]), id(a[1]), id(a[2]))
print(id(b[0]), id(b[1]), id(b[2]))
print(id(c[0]), id(c[1]), id(c[2]))
```

wydrukował

```
139948167725296 139948166858368 139948167725328
139948167725296 139948166858368 139948167725328
139948167725296 139948166871680 139948167725328
```

Widzimy, że referencje do liczb zostały w obu przypadkach przekopiowane, bo liczby są niemodyfikowalne. Ale dla drugiego elementu listy, którym jest referencja do listy, funkcja **copy** też przekopiowała referencję, ale **deepcopy** utworzyła *nową* listę niezależną od listy wskazywanej przez tę samą referencję a[1] i b[1].

Wycinki mogą też być używane do modyfikacji modyfikowalnych sekwencji (jak listy). Możemy usuwać podsekwencje albo zastępować je innymi

```
>>> a = [1, 4, '5', [6, '7'], 2.25]
>>> a[2:4] = 'hello'                    # 1
>>> a
[1, 4, 'h', 'e', 'l', 'l', 'o', 2.25]
>>> a[1:3] = [8, 9, 'X']                # 2
>>> a
[1, 8, 9, 'X', 'e', 'l', 'l', 'o', 2.25]
>>> a[2:-2] = ''                        # 3
>>> a
[1, 8, 'o', 2.25]
>>> a[1:3] = ['hello']                  # 4
>>> a
[1, 'hello', 2.25]
```

Zauważmy, że wyrażenia, które same są sekwencjami, są "wypłaszczane" i dodawane element po elemencie (#1 and #2). Jeśli chcemy dodać sekwencję jako pojedynczy element, możemy użyć jednoelementowej listy zawierającej tę sekwencję (#4). Zastąpienie podsekwencji pustą sekwencją po prostu usuwa tę podsekwencję (#3).

4.6 Operator wywołania funkcji

Operator wywołania oznaczany jest nawiasami okrągłymi i jest używany przede wszystkim do wywoływania funkcji. Ale, tak jak w C++, można zdefiniować klasy, obiekty

której zachowują się jak funkcje: w C++ wystarczy w klasie zdefiniować metodę **operator()**, w Pythonie natomiast specjalną [1] metodę **__call__** — patrz rozdz. 11.8, str. 236. Również obiekt reprezentujący samą klasę jest wywoływalny — „wywołując" klasę tworzymy jej obiekt (który sam może być wywoływalny, jeśli klasa definiuje metodę **__call__**.

Istnieje wbudowana funkcja **callable** zwracająca **True** wtedy i tylko wtedy gdy dany obiekt jest wywoływalny:

```
>>> class Hello:
...     def __call__(self):
...         return 'Hello'
...
>>> h = Hello()
>>> print(h(), callable(Hello), callable(h))
Hello True True
```

4.7 Wyrażenie await

Wyrażenie **await** jest używane w programowaniu współbieżnym korzystającym z pakietu ***asyncio*** — patrz rozdz. ??, str. ??.

4.8 Potęgowanie

Potęgowanie (podnoszenie do potęgi, jak x^y w matematyce) jest oznaczane podwójną gwiazdką (**), jak w Fortranie, a więc x^y w matematyce to `x**y` w Pythonie. Priorytet potęgowania jest wyższy od jednoargumentowych operatorów po lewej stronie, ale niższy od tych po prawej stronie (w szczególności od −): `a**-2` jest interpretowane jako `a**(-2)`, podczas gdy `-a**n` znaczy `-(a**n)`.

Operator ** może być używany wszędzie tam, gdzie jest używana funkcja **pow** w C/C++/Javie:

```
>>> 2**8
256
>>> 2**-8
0.00390625
>>> 2**(-8)
0.00390625
>>> 8**(-1/3)
0.5
>>> 8**(1/3)
2.0
>>> 8**1/3    # this is just 8/3
2.6666666666666665
>>> 26**10    # number of ten-letter strings
141167095653376
>>> 33**33
129110040087761027839616029934664535539337183380513
```

Pamiętajmy, że ^ *nie* oznacza potęgowania, jak w niektórych językach: ten symbol, jak w C/C++/Java, oznacza bitowy operator XOR!

[1] Takie metody nazywane są „"magicznymi", albo metodami „dunder".

Zresztą, funkcja **pow** istnieje również w Pythonie i jest nawet funkcją wbudowaną, której można użyć bez żadnych importów. Składnia jest taka sama jak w C/C++/Java, ale w Pythonie istnieje też wersja trójargumentowa: wszystkie argumenty muszą być całkowite, a rezultatem jest reszta z dzielenia modulo trzeci argument pierwszego argumentu podniesionego do potęgi wskazywanej przez argument drugi: `pow(x, y, n)` is therefore $x^y \mod n$. Na przykład

```
print(pow(2, 3), pow(2, 3, 5))
```

prints `8 3`

Możemy zdefiniować operator `**` dla własnych klas; wystarczy w klasie zdefiniować magiczną metodę `__pow__` (patrz rozdz. 11.8, str. 236). Na przykład

```
class bizzarePow:
    def __init__(self, s):
        self.s = s
    def __pow__(self, p):
        return (self.s*p + '\n')*p*len(self.s)

print(bizzarePow('ab')**3)
print(bizzarePow('X')**5)
```

drukuje

```
ababab
ababab
ababab
ababab
ababab
ababab

XXXXX
XXXXX
XXXXX
XXXXX
XXXXX
```

4.9 Unary operators

W Pythonie są trzy operatory jednoargumentowe: Dwa z nich

- numeryczna negacja (odwrócenie znaku): `-`; oraz
- jednoargumentowy plus, `+`, który zwraca niezmienioną wartość argumentu

nie wymagają objaśnień. Trzeci, negacji bitowej, `~`, zostanie przedstawiony w rozdziale o operatorach bitowych (patrz rozdz. 4.11, str. 53).

4.10 Arithmetic operators

Dwuargumentowe operatory arytmetyczne są ogólnie rzecz biorąc takie, jak w C/C++/Javie, choć są też dwa specyficzne dla Pythona, a interpretacja niektórych innych jest nieco różna.

Pięć z nich ma taki sam priorytet; są to

- * — mnożenie (bez problemów);
- @ — mnożenie macierzowe; żadne standardowe typy Pythona nie wspierają tego operatora, ale jest używany przez niestandardowe pakiety, szczególnie naukowe i statystyczne (jak *NumPy*[1]);
- / — dzielenie (patrz niżej);
- // — dzielenie całkowitoliczbowe (patrz niżej);
- % — operator modulo (patrz niżej).

Dwa pozostałe operatory mają niższy priorytet, ale są raczej trywialne

- + — dodawanie;
- − — odejmowanie;

i tu bez niespodzianek.

Powiedzmy tylko o nietrywialnych operatorach, których interpretacja może nie być oczywista.

4.10.1 Operator dzielenia

Operator / oznacza dzielenie. Jeśli choć jeden argument jest typu **float** (co odpowiada **double** w C/C++/Javie), rezultat jest też tego typu i tu nie ma niespodzianek.
Jednakże, ten sam operator zastosowany do liczb całkowitych daje w C/C++/Javie również liczbę całkowitą, przy czym zachodzi „obcięcie” w kierunku zera. W Pythonie, nawet dla obu argumentów całkowitych, wynikiem jest zawsze **float**!
Tak więc w Javie

```
jshell> 9/3      // JAVA
$1 ==> 3

jshell> 9/4
$2 ==> 2

jshell> 1/3
$3 ==> 0

jshell> -4/3
$4 ==> -1
```

ale w Pythonie

```
>>> 9/3          #  PYTHON
3.0
>>> 9/4
2.25
>>> 1/3
0.3333333333333333
>>> -4/3
-1.3333333333333333
```

Dla dzielenia całkowitoliczbowego mamy w Pythonie specjalny operator //. Dla argumentów całkowitych wynik jest też całkowity, ale „obcinanie” nie jest w kierunku zera, jak w C/C++/Javie, ale w kierunku $-\infty$, w dół (i dlatego takie dzielenie jest nazywane „podłogowym”, *floor division*). W Javie mamy zatem

[1] https://numpy.org

```
jshell> 5/4      // JAVA
$1 ==> 1

jshell> -5/4
$2 ==> -1        // -1.25 truncated towards zero

jshell> Math.floor(-5./4)
$3 ==> -2.0
```

podczas gdy w Pythonie

```
>>> import math #  PYTHON
>>> 5//4
1
>>> -5//4       #  -1.25 truncated towards minus infinity
-2
>>> math.floor(-5/4)
-2
```

czyli // jest jak „zwykłe” dzielenie, po którym następuje wzięcie „podłogi”.

4.10.2 Operator modulo

Tak jak w C/C++/Java, operator modulo, , lub reszty, % nie odpowiada operatorowi modulo w matematyce dla dla liczb ujemnych. W dodatku jego działanie w Pythonie różni się od tego w C/C++/Java.

We wszystkich tych językach, następujący wzór dla całkowitych a i b (gdzie oczywiście $b \neq 0$) musi obowiązywać

$$(a/b) * b + a\%b = a$$

gdzie / jest dzieleniem całkowitym.

Dla C++ możemy to sprawdzić prostym programem

```
#include <iostream>
#include <cstdio>
void pr(int a, int b) {
    static int fst = 0;
    if (++fst == 1)
        std::printf("C++\n  a   b a/b a%%b (a/b)*b+a%%b\n");
    std::printf("%3d%4d%3d%4d%9d\n",a,b,a/b,a%b,(a/b)*b+a%b);
}
int main() {
    pr( 17,  10);
    pr( 17, -10);
    pr(-17,  10);
    pr(-17, -10);
}
```

który drukuje

```
C++
  a   b a/b a%b (a/b)*b+a%b
 17  10  1   7        17
 17 -10 -1   7        17
```

```
-17  10 -1  -7      -17
-17 -10  1  -7      -17
```

Dla ujemnych argumentów zasada tu jest prosta: oblicz modulo dla wartości bezwzględnych argumentów i weź rezultat ze znakiem a (ignorując znak b).

Aby jednak nasz wzór działał również w Pythonie, definicja operatora modulo musi być inna, bo inna jest definicja dzielenia całkowitoliczbowego!

Podobny program w Pythonie

```
def pr(a, b, fst=[]):
    if not fst:
        print("PYTHON\n  a   b a//b a%b (a//b)*b+a%b")
        fst.append(1)
    print(f"{a:3}{b:4}{a//b:4}{a%b:4}{(a//b)*b+a%b:9}");

pr( 17,  10);
pr( 17, -10);
pr(-17,  10);
pr(-17, -10);
```

drukuje

```
PYTHON
  a   b a//b a%b (a//b)*b+a%b
 17  10   1   7       17
 17 -10  -2  -3       17
-17  10  -2   3      -17
-17 -10   1  -7      -17
```

i najłatwiej to ogarnąć pamiętając, że zawsze musi być $a\%b = a - (a/b) * b$ gdzie / to dzielenie całkowitoliczbowe właściwe dla danego języka (// dla Pytona).

Najlepiej zatem w ogóle unikać operatorów dzielenia całkowitoliczbowego i modulo, jeśli podejrzewamy, że argumenty mogą być ujemne.

4.11 Bitwise operators

Sześć operatorów bitowych w Pytonie, w kolejności malejących priorytetów, to:

- bitowe NOT (~),
- lewe (<<) i prawe (>>) przesunięcia,
- bitowe AND (&),
- bitowe XOR (^),
- bitowe OR (|).

Operacje bitowe są w Pythonie trochę „tricky”. Problem wynika z tego, że reprezentacja liczb całkowitych w tym języku jest skomplikowana (patrz rozdz. 3.4.1, str. 33) i ma mało wspólnego z implementacją liczb całkowitych w C/C++/Javie. W tych językach reprezentacja liczb całkowitych ma ustaloną długość z najstarszym bitem traktowanym inaczej niż pozostałe (dla typów **signed**). Ponieważ długość (liczba bitów) jest znana, wiadomo, który bit jest najstarszy. Jednak w Pythonie reprezentacja jest inna: z samej konstrukcji pozwala ona na zapis liczb dowolnie dużych, których reprezentacja w „normalnym” kodzie uzupełnień do 2 wymagałaby dowolnie długich ciągów bitów.

Ponieważ jednak pierwszym niezerowym bitem (licząc od lewej) jest w układzie binarnym zawsze 1, nie wiadomo by było, czy wkład od tej jedynki ma być wzięty z minusem, czy z plusem.

Operatory przesunięcia działają zgodnie z przewidywaniem, ale pewne szczegóły są inne. Przesuwanie w lewo jest *zawsze* równoważne mnożeniu przez potęgę dwóch

```
m << n   ≡   m * pow(2, n)
```

Dla dodatnich m możemy łatwo sprawdzić, że odpowiada to przesuwaniu w „normalnej” reprezentacji bitowej m o n bitów, z zerami wchodzącymi od prawej (poniżej używamy funkcji **bin** aby zobaczyć bitową reprezentację *dodatniej* liczby):

```
>>> a = 153
>>> bin(a)
'0b10011001'
>>> b = a << 3
>>> bin(b)
'0b10011001000'
>>> bin(a*2**3)
'0b10011001000'
```

Zauważmy, że w reprezentacji bitowej o ustalonej długości, jak w C/C++/Javie, skrajne lewe bity mogą „wypaść” poza lewą „krawędź” — wtedy przesuwanie w lewo może *nie* być równoważne mnożeniu przez potęgę 2, a rezultat może nawet zmienić znak (dla typów ze znakiem); następujący fragment w C++

```
int a = 100'000'000;
cout << (a << 10) << endl;
```

drukuje

```
-679215104
```

i podobnie w Javie

```
jshell> int a = 100_000_000
a ==> 100000000

jshell> System.out.println(a << 10)
-679215104
```

Jednak w Pythonie to nie jest problem, bo reprezentacja liczb całkowitych może być dowolnie długa:

```
>>> a = 1_000_000
>>> b = a << 10
>>> c = a * 2**10
>>> bin(a)
'0b11110100001001000000'
>>> bin(b)
'0b11110100001001000000000000000000'
>>> bin(c)
'0b11110100001001000000000000000000'
>>> b, c
(1024000000, 1024000000)
>>> b == c
True
```

Trochę gorzej jest z liczbami ujemnymi. Równoważność

```
m << n    ≡    m * pow(2, n)
```

też obowiązuje. Nie jest jednak tak prosto zobaczyć wzorzec bitów, bo dla ujemnych liczb funkcja **bin** zwraca reprezentację wartości bezwzględnej z dodanym znakiem minus na początku, a nie reprezentację samej liczby w kodzie uzupełnień do dwóch:

```
>>> a = 913
>>> b = -a
>>> bin(a)
'0b1110010001'
>>> bin(b)
'-0b1110010001'
```

Możemy jednak użyć metody **to_bytes** aby zobaczyć poszczególne bajty:[1]

```
>>> a = 913
>>> b = -a
>>> c = b << 2
>>> d = b * 2**2
>>> c == d
True
>>> c, d
(-3652, -3652)
>>> e = d.to_bytes(4, 'big', signed=True) # 4 bytes as int in Java
>>> e
b'\xff\xff\xf1\xbc'
>>> int.from_bytes(e, 'big', signed=True)
-3652
```

\xff\xff\xf1\xbc to 11111111111111111111000110111100 binarnie, istotnie, możemy sprawdzić w Javie, że jest to -3652:

```
jshell> a = -3652
a ==> -3652

jshell> Integer.toBinaryString(a)
$1 ==> "11111111111111111111000110111100"
```

Bitowe operatory AND (&), OR (|) i XOR (^) działają zgodnie z przewidywaniem (ale też lepiej unikać liczb ujemnych):

```
>>> m = 0b11001010
>>> n = 0b11010
>>> bin(m & n)
'0b1010'
>>> bin(m | n)
'0b11011010'
>>> bin(m ^ n)
'0b11010000'
```

[1] Wywołując tę metodę podajemy, że rezultat ma być 4-bajtowy (jak dla **int** w C/C++/Java) i że porządek bajtów powinien być big — od bajtu najbardziej do najmniej znaczącego. [Zamiast little lub big możemy też użyć sys.byteorder, czyli porządek właściwy dla danej platformy; zwykle jest to little.]

Bitowy operator negacji NOT (*bit-flip*, ~) Nie jest taki trywialny. Ponieważ liczby całkowite są dowolnie duże, możemy sobie wyobrażać, dla liczby dodatniej, że za ostatnią (z lewej strony) jedynką występuje nieskończona liczba zer. Odwrócenie bitów spowodowałoby otrzymanie nieskończonej liczby jedynek. I tak właśnie jest: oczywiście nie są one nigdzie fizycznie przechowywane, ale są tam koncepcyjnie.

Zobaczmy, co się dzieje gdy negujemy bitowo liczbę. Rozpatrzmy na przykład liczbę 321_{10} (=101000001_2). Jej reprezentacja bitowa to

```
>>> bin(321)
'0b101000001'
>>> (321).bit_length()
9
```

Python dodaje jedno zero na początku (bo liczba jest dodatnia), ten ciąg dziesięciu bitów neguje:

```
0101000001          # 321 plus leading 0
1010111110          # flipped
```

i traktuje powstały ciąg bitów jako zapis binarny 10-bitowej liczby w kodzie uzupełnień do 2; pierwszy bit jest traktowany jako bit znaku, więc wkład od niego brany jest z minusem i dostajemy wartość $-2^9 + 2^7 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1$ czyli $-512 + 128 + 32 + 16 + 8 + 4 + 2 = -322$.

```
>>> a = 321
>>> nega = ~a
>>> bin(a)
'0b101000001'
>>> bin(nega)
'-0b101000010'
>>> a, nega
(321, -322)
>>> negabytes = nega.to_bytes(2, 'big', signed=True)
>>> negabytes
b'\xfe\xbe'
>>> int.from_bytes(negabytes, 'big', signed=True)
-322
```

`\xfe\xbe` is 1111111010111110: there is "infinite" sequence of ones at the beginning, so the number is negative and only the last one is significant (giving, as we showed above, -322).

Jak pamiętamy, w kodzie uzupełnień do 2 odwrócenie znaku jest równoważne odwróceniu wszystkich bitów i dodaniu jedynki. Możemy sprawdzić, że ta własność jest zachowana:

```
>>> ~321+1
-321
>>> ~-9987654321+1
9987654321
>>> ~123123123123123123+1
-123123123123123123
>>> ~99999999999999999999999999999999999999999999999+1
-99999999999999999999999999999999999999999999999
```

4.12 Porównania

4.12.1 Porównania arytmetyczne

Sześć standardowych operatorów porównań arytmetycznych do porównań *wartości,* obiektów <, < =, ==, !=, > = i >, działa tak samo jak w innych językach, więc nie ma sensu ich omawiać. Mają one jednak pewną cechę specyficzną dla Pythona — można stosować porównania „łańcuchwe", których nie da się zinterpretować z zasad łączności (lewej lub prawej). Takie łańcuchy porównań byłyby nielegalne w Javie, ale skompilowałyby się w C/C++, dając bezsensowne rezultaty; na przykład:

```
int a = 1, c = 3;
cout << boolalpha << (c < a < a) << endl;
```

wydrukowałoby **true**.

W Pythonie jednak takie łańcuchy porównań są legalne mają dobrze określony sens. Na przykład

```
a < b < c
```

to ani

```
(a < b) < c
```

(jak byłoby to w C/C++), ani

```
a < (b < c)
```

ale

```
a < b and b < c
```

(z tym, że b będzie obliczane tylko raz). Łańcuchy porównań są równoważne koniunkcji oddzielnych porównań od lewej do prawej:

```
>>> 5 < 7 > 2
True
>>> 5 < 7 and 7 > 2
True
>>>
>>> 5 < 7 > 7
False
>>> 5 < 7 and 7 > 7
False
>>>
>>> a = 5; b = 7
>>> a < b == b - a + 5
True
>>> a < b and b == b - a + 5
True
>>>
>>> a == b - 2 < b >= a
True
>>> a == b - 2 and b - 2  < b and b >= a
True
```

4.12.2 Porównanie identyczności

Są dwa operatory porównania identyczności: **is** i **is not**. Używa się ich (raczej rzadko) do stwierdzenia, czy dane dwa obiekty są w istocie tym samym obiektem czy dwoma

obiektami (o, być może, tej samej wartości). Pamiętajmy, że == porównuje *wartości,* a nie identyfikatory czy adresy (co na jedno wychodzi). Taki operator jest zdefiniowany dla większości typów wbudowanych. Jeśli nie jest zdefiniowany, na przykład dla naszych własnych klas, to będzie użyte zamiast niego **is** porównujące adresy (analogicznie do metody **equals** Javy).

Na przykład dla klasy **str** (string), operator == jest oczywiście dobrze zdefiniowany (jak dla **String::equals** w Javie):

```
>>> a = 'Hello, '; b = 'World'
>>> s1 = a + b
>>> s2 = 'Hello, World'
>>> s1 is s2
False
>>> s1 == s2
True
>>> s1 is not s2
True
```

Jak pamiętamy, małe liczby całkowite są tworzone raz i reference do danej wartości są tak naprawdę referencjami do dokładnie tego samego obiektu (patrz rozdz. 3.4.1, str. 33): możemy to sprawdzić za pomocą operatora **is**

```
>>> a = 17; b = 17
>>> a == b
True
>>> a is b
True
>>> id(a), id(b)
(9801760, 9801760)
>>>
>>> # but:
>>>
>>> a = 1200; b = -1000; c = a - b - 1000
>>> a == c
True
>>> a is c
False
>>> id(a), id(c)
(140120910778288, 140120910340336)
```

4.12.3 Testowanie przynależności

Test przynależności ma ten sam priorytet, co operatory porównania. Operator **in** i jego negacja **not in** sprawdza, czy dana wartość znajduje się w kolekcji: liście, krotce, słowniku, zbiorze czy też napisie, który jest sekwencją znaków. Dla słowników porównywany jest tylko klucz pomijając związaną z nim wartość. Użycie tych operatorów jest oczywiste:

```
ls = [(1, 2), (3, 4, 5), (6, 7, 8, 9)]
di = {'one': 1, 'two':2, 'three':3}
se = {1, 2, 3, 4}
st = 'abcde'
```

```
print((1, 2) in ls)    # True
print((2, 1) in ls)    # False
print(4 in ls[1])      # True
print(4 in ls[2])      # False
print('one' in di)     # True
print(2 in di)         # False
print(4 in se)         # True
print(5 in se)         # False
print('cd' in st)      # True
print('f' in st)       # False
```

4.13 Logiczne operatory not, and i or

Operatory logiczne: **not**, **and** oraz **or** odpowiadają znanym z C/C++/Javy operatorom **!**, **&&** i **||**.

Tak jak w tych językach, operatory **and** i **or** są skrótowe (*short-circuited*) — drugi argument nie jest w ogóle wyliczany, jeśli po wyliczeniu pierwszego rezultat jest już znany.

Jak pamiętamy, (patrz rozdz. 3.4.2, str. 35) wartości dowolnego typu (z wyjątkiem **NotImplementedType**) mają wartość logiczną w kontekstach wymagających takiej wartości: numeryczna wartość 0, puste sekwencje i kolekcje, **None** i **False** są „falsy", wszystko inne jest „truthy".

Wynika z tego, że argumenty (operandy) operatorów logicznych mogą być dowolnego typu, niekoniecznie **bool** — będą i tak traktowane jak **True** lub **False**. Ale jaki jest wartości zwracanej?

Dla operatora **not** jest to **bool**: **True** albo **False**:

```
>>> not {}
True
>>> not {7}
False
>>> not 3
False
>>> not 0
True
>>> not [1, 2, 3]
False
>>> not []
True
>>> not 'a'
False
>>> not ''
True
>>> not False
True
>>> not True
False
```

Jednak dla operatorów **and** i **or** zwracana wartość *nie* będzie **True** albo **False** tylko po prostu wartość jednego z argumentów (która, jak wiemy, może być zawsze zinterpretowana jako wartość logiczna w kontekście tego wymagającym). Zasady są następujące:

- koniunkcja: `expr1 and expr2` zwraca
  - `expr1` (i `expr2` nie jest w ogóle obliczane) jeśli `expr1` jest „falsy”;
  - `expr2` w przeciwnym przypadku.
- Alternatywa: `expr1 or expr2` zwraca
  - `expr1` (i `expr2` nie jest w ogóle obliczane) jeśli `expr1` jest „truthy”;
  - `expr2` w przeciwnym przypadku.

Na przykład:

```
>>> a = 0 and [1, 2]
>>> a, type(a), bool(a)
(0, <class 'int'>, False)
>>>
>>> a = -1 and [1, 2]
>>> a, type(a), bool(a)
([1, 2], <class 'list'>, True)
>>>
>>> a = -1 and {}
>>> a, type(a), bool(a)
({}, <class 'dict'>, False)
>>>
>>> a = [] or {}
>>> a, type(a), bool(a)
({}, <class 'dict'>, False)
>>>
>>> a = [] or {1, 2}
>>> a, type(a), bool(a)
({1, 2}, <class 'set'>, True)
```

Jak pokazuje ten przykład, funkcja **bool** — a ściślej konstruktor klasy **bool** — może być użyty do konwersji dowolnego obiektu na „prawdziwą” wartość logiczną.

4.14 Wyrażenie warunkowe

Wyrażenie warunkowe odpowiada trójargumentowemu operatorowi `?:` w C/C++/Javie. Składnia jest jednak nieco inna:

```
expr1 if condition else expr2
```

i wartością całego wyrażenia jest wartość

- `expr1` jeśli `condition` ma wartość „truthy” (i `expr2` nie jest wtedy w ogóle obliczana);
- `expr2` jeśli `condition` ma wartość „falsy” (i `expr1` nie jest wtedy obliczane).

Oba wyrażenia, `expr1` i/lub `expr2` mogą być **None** — na przykład jeśli jest to wywołanie funkcji, która niczego nie zwraca (czyli zwraca **None**):

```
a = 6
b = print('even') if a % 2 == 0 else print('odd')
print(b)
```

drukuje

```
even
None
```

Inny przykład:

```
n = p = 7
while p > 1:
    print(n, end=' ')
    p, n = n, n//2 if n%2 == 0 else 3*n + 1
```

drukuje ciąg Collatza (gradowy) rozpoczynający się of 7

```
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Poniższy program sprawdza, które z liczb Fibonacciego F_n dla $n \in [3, 15]$ są liczbami pierwszymi; użyte tu jest wyrażenie warunkowe, a również, w ostatniej linii, wyrażenie przypisania (operator *walrus*), aby uniknąć wielokrotnego obliczania tych samych liczb Fibonacciego:

Listing 15 ABB-Cond/Cond.py

```
#!/usr/bin/env python

def is_prime(n):
    if n <= 1: raise ValueError
    if n <= 3: return True
    if n % 2 == 0 or n % 3 == 0: return False
    k = 5
    while k**2 <= n:
        if n % k == 0 or n % (k+2) == 0: return False
        k += 6
    return True

def fib(n):
    f = 0; s = 1
    if n < 2: return n
    for i in range(n-1): f, s = s, f+s
    return s

for n in range(3, 16):
    print(f"{v} yes" if is_prime(v := fib(n)) else f"{v} no")
```

Program drukuje

```
2 yes
3 yes
5 yes
8 no
13 yes
21 no
```

```
34 no
55 no
89 yes
144 no
233 yes
377 no
610 no
```

Wyrażenia warunkowe mogą być zagnieżdżane

```
expr1 if condition1 else expr2 if condition2 else expr3
```

ale używanie tak mętnej składni nie jest zalecane, bo ciężko to przeczytać i zrozumieć. Wiązanie jest prawe (jak dla `?:` w C/C++/Javie), więc powyższe wyrażenie jest równoważne

```
expr1 if condition1 else (expr2 if condition2 else expr3)
```

co łatwiej zrozumieć — wartością jest tu zatem

- `expr1` jeśli `condition1` jest prawdą — pozostałe warunki i wyrażenie nie są wtedy obliczane;
- `expr2` jeśli `condition1` jest fałszem, ale `condition2` prawdą;
- `expr3` jeśli i `condition1` i `condition2` są fałszywe.

Na przykład:

```
a = 1; b = 7
x1 = 'expr1' if a < b else 'expr2' if a > b else 'expr3'
x2 = 'expr1' if a > b else 'expr2' if a < 2 else 'expr3'
x3 = 'expr1' if a > b else 'expr2' if b < a else 'expr3'
print(x1)
print(x2)
print(x3)
```

drukuje

```
expr1
expr2
expr3
```

Wyrażenia warunkowe mogą być też używane w konstruowaniu sekwencji składanych; na przykład poniżej zastępujemy liczby z zakresu $[-1, 8)$ mniejsze od 2 dwójką, a większe od 5 piątką:

```
ls = [5 if n > 5 else 2 if n < 2 else n for n in range(-1, 8)]
print(ls)  # prints [2, 2, 2, 2, 3, 4, 5, 5, 5]
```

Rozdział 5

Funkcje

5.1 Definiowanie funkcji. Zasięg zmiennych

Funkcje, jak wszystko inne, są w Pythonie reprezentowane przez obiekty. Prosta definicja funkcji może wyglądać tak

```
def mul(x, y):
    return x*y

def printData(x):
    print('Data is {}'.format(x))   # No return
```

Jak widzimy, typ zwracany w ogóle *nie* jest deklarowany. Tak naprawdę wszystkie funkcje Pythona coś zwracają, nawet jeśli ich wykonanie nie kończy się instrukcją **return** lub gdy **return** wprawdzie jest, ale bez żadnej wartości, która miałaby być zwrócona — zwracaną wartością jest wtedy **None**.

Również typ parametrów pozostaje nieokreślony: będzie on określany dynamicznie na podstawie typów argumentów za każdym razem gdy funkcja jest wywoływana. Wynika z tego, że ta sama funkcja może być wywoływana za każdym razem z parametrami całkowicie różnych i nie związanych ze sobą typów — jeśli składnia ciała funkcji okaże się prawidłowa dla danych typów, takie wywołania mogą być prawidłowe. Taka cecha zwana jest *kaczym typowaniem* (ang. *duck typing*.)

```
>>> def mul(x, y):
...     return x*y
...
>>> def printData(x):
...     print('Data:', x, type(x))
...
>>> printData(1.5)
Data: 1.5 <class 'float'>
>>>
>>> printData((1, 2, 3))
Data: (1, 2, 3) <class 'tuple'>
>>>
>>> printData([1, 'Eve'])
Data: [1, 'Eve'] <class 'list'>
>>>
>>> printData(r := mul(3, 3))
Data: 9 <class 'int'>
>>>
>>> printData(r := mul((1+2j), (2-4j)))
Data: (10+0j) <class 'complex'>
>>>
>>> printData(r := mul([1, 2], 3))
Data: [1, 2, 1, 2, 1, 2] <class 'list'>
```

W przeciwieństwie do większości innych języków, definicja funkcja jest instrukcją wykonywalną. Na przykład definicja funkcji **printData** powyżej nie różni się składniowo od

definicji zmiennej liczbowej `x = 7`. Nazwa printData jest, jak zwykle, nazwą referencji do obiektu reprezentującego nasza funkcję. Możemy ją skopiować do innej zmiennej

```
>>> pp = printData
>>> pp(1+2j)
Data: (1+2j) <class 'complex'>
>>> printData = 17
>>> printData
17
```

Jednak początkowo nadana nazwa funkcji wciąż jest jednym z atrybutów obiektu reprezentującego naszą funkcję, co jest zresztą łatwo zmienić:

```
>>> dir(pp)
['__annotations__', '__builtins__', '__call__', '__class__',
 '__closure__', '__code__', '__defaults__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__']
>>> pp.__name__
'printData'
>>> pp.__name__ = 'pp'
>>> pp.__name__
'pp'
```

Funkcje, będąc obiektami, mogą być tworzone wewnątrz innych funkcji, zwracane z funkcji czy przesyłane do nich jako argumenty. Funkcje, które zwracają lub pobierają jako argumenty inne funkcje nazywane są **funkcjami wyższego rzędu.**

W poniższym przykładzie funkcja **compos** pobiera dwie funkcje, f i g, i zwraca też funkcję, a mianowicie ich kompozycję[1]

Listing 16 AAP-FunCompos/Compos.py

```
1  #!/usr/bin/env python
2
3  def compos(f, g):
4      def _h(x):
5          return f(g(x))
6      # just for fun...
7      _h.__name__ = g.__name__ + '_and_then_' + f.__name__
8      return _h
9
10 def squa(x):
11     return x*x
12 def add2(x):
13     return x+2
```

[1]Kompozycją funkcji f i g jest funkcja $f \circ g$ taka, że $(f \circ g)(x) \equiv f(g(x))$.

```
14
15 sqad = compos(squa, add2)
16 adsq = compos(add2, squa)
17
18 print(type(squa), type(add2))
19
20 print('name of sqad:', sqad.__name__)
21 print('name of adsq:', adsq.__name__)
22
23 print(sqad(2))    # 16
24 print(adsq(2))    #  6
```

Program drukuje

```
<class 'function'> <class 'function'>
name of sqad: add2_and_then_squa
name of adsq: squa_and_then_add2
16
6
```

Spójrzmy na inny przykład:

Listing 17 AAR-FunCount/Counter.py

```
1  #!/usr/bin/env python3
2
3  def getCounter():
4      n = 0
5      def count():
6          nonlocal n
7          n += 1
8          return n
9      return count
10
11 f = getCounter()
12 for i in range(3):
13     print(f(), end = ' ')    # 1 2 3
14
15 print('\n' + str(type(f)))  # <class 'function'>
16 print(f.__name__)           # count
```

Co jest tu ważne, to użycie słowa kluczowego **nonlocal** w linii 6. Prowadzi nas to do pojęcia **zakresu** (ang. *scope*): obszar programu gdzie widoczne są powiązania obiektów z niekwalifikowanymi nazwami, pod którymi są one dostępne. W czasie wykonania programu, nazwy są wyszukiwane w następujących zakresach:

1. Najbardziej wewnętrzny zakres **L**okalny.
2. Zakres otaczających – **E**nclosing – funkcji, jeśli jest to funkcja zagnieżdżona.
3. Zakres **G**lobalny modułu.
4. Zakres nazw wbudowanych – **B**uilt-in.

Gdy użyta jest nazwa, jej znaczenie poszukiwane jest po kolei w wymienionych zakresach — jest to zasada zwana **LEGB**. Obowiązują przy tym następujące prawidła:

- w zakresie lokalnym (wewnątrz funkcji), nazwy pojawiające się po lewej stronie przypisania są traktowane jako nazwy zmiennych lokalnych (które są tworzone, jeśli nie istniały);
- w pozostałych przypadkach, nazwa odnosi się do zmiennej lokalnej, jeśli istnieje: jeśli nie istnieje, to jest poszukiwana w nieglobalnym zakresie otaczającym (jeśli taki istnieje), następnie w zakresie globalnym modułu, a w końcu w przestrzeni nazw wbudowanych;
- jeśli w zakresie lokalnym (na przykład wewnątrz funkcji) nowa wartość ma być przypisana do zmiennej globalnej, ta zmienna musi być przed użyciem zadeklarowana jako **global**.
- Załóżmy, że jesteśmy wewnątrz zakresu, który jest zawarty w innym lokalnym zakresie (na przykład w funkcji zawartej w innej funkcji). Wtedy, aby przypisać nową wartość do zmiennej z tego otaczającego zakresu (ale *nie* globalnego), zmienna ta w zakresie wewnętrznym musi być przed użyciem zadeklarowana jako **nonlocal**,

Na przykład kod poniżej nie będzie działać: do zmiennej `n` wewnątrz funkcji przypisujemy wartość, więc jest ona traktowana jako nowa zmienna lokalna, ale jej nowa wartość sama zależy od `n`:

```
>>> n = 1
>>> def fun():
...     n = n + 7
...
>>> fun()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fun
UnboundLocalError: local variable 'n' referenced before assignment
```

ale to będzie działać

```
>>> n = 1
>>> def fun():
...     global n
...     n = n + 7
...
>>> fun()
>>> print(n)
8
```

Podobnie, następujący fragment jest nieprawidłowy

```
>>> def fun():
...     n = 9
...     def inn():
...         n += n
...         return n - 8
...     return inn
...
```

```
>>> f = fun()
>>> print(f())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in inn
UnboundLocalError: local variable 'n' referenced before assignment
```

ale prawidłowy jest kod poniżej, bo tu nie próbujemy przypisać do n, więc nazwa będzie poszukiwana, i zostanie znaleziona, w zakresie otaczającym

```
>>> def fun():
...     n = 9
...     def inn():
...         return n - 8
...     return inn
...
>>> f = fun()  # f is now inn
>>> print(f())
1
```

Podobnie, kod poniżej jest prawidłowy, bo zadeklarowaliśmy n jako **nonlocal**

```
>>> def fun():
...     n = 9
...     def inn():
...         nonlocal n
...         n += n
...         return n - 8
...     return inn
>>> f = fun()
>>> print(f())
10
```

Rozpatrzmy kod poniżej. Zmienna n jest utworzona za pierwszym (i jedynym) wywołaniem funkcji **halves**. Wewnątrz zagnieżdżonej funkcji **addhalf** jest ona zadeklarowana jako **nonlocal**, a więc to ta sama zmienna utworzona w linii 7. Zauważmy, że jest ona używana (i redefiniowana) wewnątrz funkcji, która to funkcja jest zwracana z **halves**. Wydawałoby się, że wobec tego n już nie istnieje, bo była zmienną lokalną w **halves**. Ale będzie skopiowana do tak zwanego **domknięcia** (ang. *closure*) funkcji **addhalf** i dostępna w niej do odczytu i modyfikacji:

Listing 18 AAS-Scopes/Scopes.py

```
1  #!/usr/bin/env python
2
3  n = 3.0                              # global
4  print('Global n:', id(n), n, end = '\n\n')
5
6  def halves():
7      n = -0.5                         # local
8      print('Local  n:', id(n), '\n')
9      def addhalf():
10         nonlocal n                   # n from line 7
```

```
11          print('Inner1 n:', id(n), n) # same as local
12          n += 0.5
13          print('Inner2 n:', id(n), n) # changed now
14          return n
15      return addhalf
16
17  f = halves()
18  for i in range(2):
19      print(f(), end = '\n\n')
20
21  print('Global n:', id(n), n)
22  print('From closure:',f.__closure__[0].cell_contents)
```

Program drukuje

```
Global n: 139640753183728 3.0

Local  n: 139640753183472

Inner1 n: 139640753183472 -0.5
Inner2 n: 139640753183632 0.0
0.0

Inner1 n: 139640753183632 0.0
Inner2 n: 139640753183600 0.5
0.5

Global n: 139640753183728 3.0
From closure: 0.5
```

[Zob. też dyskusję w rozdziale o lambdach, rozdz. 5.3, str. 74]

W przeciwieństwie do takich języków jak C/C++/Java, bloki instrukcji złożonych (na przykład pod **if**em lub w pętli), *nie* tworzą osobnych zasięgów. Zmienne zdefiniowane w takich blokach są dynamicznie dodawane do słownika lokalnych zmiennych i tam już pozostają nawet gdy przepływ sterowania opuści blok. Rozpatrzmy następujący przykład

```
x = int(input('Enter x (negative or positive) '))

if x >= 0:
    a = 'a'
else:
    b = 'b'

print('"a" exists?', 'a' in locals())
print('"b" exists?', 'b' in locals())
```

Jeśli uruchamiając ten program podamy dodatnią liczbę, po instrukcji **if** zmienna `a` będzie istnieć, ale `b` nie

```
Enter x (negative or positive) 4
"a" exists? True
"b" exists? False
```

natomiast z wartością ujemną jako daną wejściową

```
Enter x (negative or positive) -3
"a" exists? False
"b" exists? True
```

to b będzie istnieć, ale a nie.

5.2 Przekazywanie argumentów

Spójrzmy na następujący przykład:

```
>>> def fun(a, b, c):
...     return f'{a=} {b=} {c=}'
...
>>> fun(1, 'Hello', 2.5)
"a=1 b='Hello' c=2.5"
>>>
>>> fun(c='World', a=17, b=(3, '3'))
"a=17 b=(3, '3') c='World'"
>>>
>>> fun(21, c=8.28, b='Ottawa')
"a=21 b='Ottawa' c=8.28"
>>>
>>> fun(a=21, 'Ottawa', c=8.28)
  File "<stdin>", line 1
    fun(a=21, 'Ottawa', c=8.28)
                              ^
SyntaxError: positional argument follows keyword argument
```

Parametry funkcji **fun** są *pozycyjne,* jest ich trzy i nie mają określonych wartości domyślnych. Zatem, wywołując funkcję, musimy podać dokładnie argumenty. Jednakże, jak widzimy w drugim wywołaniu, ich kolejność może być zmieniona, jeśli *nazwy* parametrów zostały określone. Mówimy wtedy, że argumenty zostały przesłane jako *nazwane* – ang. *keyword arguments.*

W trzecim wywołaniu pierwszy argument jest bez nazwy, więc musi odpowiadać pierwszemu parametrowi pozycyjnemu. Generalnie, nienazwane argumenty muszą być podane jako pierwsze i w odpowiedniej kolejności; pozostałe, nazwane, mogą być przesyłane w dowolnej kolejności. Jeśli któryś argument jest przesyłany „po nazwie", to nie może po nim wystąpić argument nienazwany — w takim przypadku dostaniemy wyjątek, jak w czwartym wywołaniu w powyższym przykładzie.

Może się zdarzyć, że funkcja oczekuje kilku argumentów, ale w naszym programie mamy je w postaci sekwencji (listy albo krotki). Nie możemy przesłać jej bezpośrednio, bo byłby to *jeden* argument (typu **list** albo **tuple**). Możemy jednak „rozpakować" tę sekwencję na pojedyncze elementy stosując operator *:

```
>>> def fun(a, b, c):
...     return f'{a=} {b=} {c=}'
...
>>> lst = ['a', 1, 'Hello']
>>> fun(*lst)
"a='a' b=1 c='Hello'"
```

```
>>> tpl = 6, 'World', 1.5
>>> fun(*tpl)
"a=6 b='World' c=1.5"
```

Podobnie, możemy „rozpakować” słownik na ciąg jego pojedynczych wpisów z nazwami parametrów jako kluczami i przesłać tak rozpakowany słownik jako ciąg argumentów nazwanych; rozpakowania słownika dokonujemy za pomocą operatora **:

```
>>> def fun(a, b, c):
...     return f'{a=} {b=} {c=}'
...
>>> dct = dict(b = 'Hello', c = 1.25)
>>> fun('First', **dct)
"a='First' b='Hello' c=1.25"
```

Tu pierwszy argument jest nienazwany, więc musi odpowiadać pierwszemu argumentowi pozycyjnemu — pozostałe dwa argumenty będą nazwane po rozpakowaniu słownika.

Inny przykład:

```
#!/usr/bin/env python

def fun(a, b, c, d, e, f):
    print(a, b, c, d, e, f)

fun(1, *(2, 3), f=6, *(4, 5))
fun(*(1, 2), e=5, *(3, 4), f=6)
fun(1, **{'b': 2, 'c': 3}, d=4, **{'e': 5, 'f': 6})
fun(c=3, *(1, 2), **{'d': 4}, e=5, **{'f': 6})
```

drukuje

```
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
```

Podobnie jak w C++ (ale nie Javie), dla wszystkich lub niektórych parametrów funkcji możemy określić wartości domyślne. Jeśli parametry z wartościami domyślnymi występują, to muszą zostać podane na liście parametrów funkcji jako *ostatnie.* Funkcje posiadające wartości domyślne mogą zatem być wywołane z mniejsza liczbą argumentów — interpreter użyje wartości domyślnych dla „brakujących” argumentów. Spójrzmy na przykład:

Listing 19 AAU-DefArgs/DefArgs.py

```
1 #!/usr/bin/env python3
2
3 def f(a, b='defaultB', c=None):
4     if c == None:
5         print("c hasn't been specified!", end = ' ')
```

```
6       print(f'{a=} {b=} {c=}')
7   f(1, 2, 3)  # a=1 b=2 c=3
8   f(1, 2)     # c hasn't been specified! a=1 b=2 c=None
9   f(1)        # c hasn't been specified! a=1 b='defaultB' c=None
10  f(c='c', a='a', b='b')  # a='a' b='b' c='c'
11
12  i = 7
13  def fundef(a, b=i): # default value of b will always be 7!
14      return a + b;
15
16  i += 10
17    # this will print 'i = 17 fundef(1) = 8'
18  print('i =', i, 'fundef(1) =', fundef(1))
19
20    # important example
21  def funlist(val, lst = []): # default list created once!
22      lst.append(val)
23      return lst;
24  funlist(1)
25  funlist(2)
26  funlist(3)
27  print(funlist(4))                # [1, 2, 3, 4]
28  print(funlist('c', ['a', 'b'])) # ['a', 'b', 'c']
29  print(funlist('d'))              # [1, 2, 3, 4, 'd'] !
```

Przykład ilustruje ważny problem, o którym musimy pamiętać, gdy używamy wartości domyślnych. Obiekt reprezentujący wartość domyślną jest tworzony tylko raz, w momencie gdy interpreter przetwarza definicję funkcji. W każdym jej wywołaniu z pominiętym argumentem, dokładnie ten sam obiekt będzie użyty jako wartość domyślna. Na przykład, w linii 13 definiujemy funkcję z argumentem domyślnym — jego wartością (niemodyfikowalną, bo to liczba) jest 7. Nawet jeśli zmodyfikujemy zmienną i w linii 16 i wywołamy naszą funkcję bez podania argumentu, użyta zostanie stara wartość, czyli 7. To jest jasne dla argumentów domyślnych niemodyfikowalnych (liczby, napisy, ...). Jednak, jak widzimy z przykładu gdzie wartością domyślną jest pusta lista (linia 21), może to prowadzić do niespodzianek jeśli wartością domyślną jest obiekt modyfikowalny (jak na przykład lista). Wtedy dla kolejnych wywołań z pominiętym argumentem, ten sam obiekt będzie użyty jako wartość domyślna i będzie on taki, jak był po poprzednim takim wywołaniu. A zatem *nie* będzie to już lista pusta!

Python wspiera funkcje ze zmienną liczbą argumentów (ang. *variadic functions*). Ciąg argumentów (być może pusty) jest oznaczany na liście parametrów poprzez *args (nazwa args jest konwencją, ważna jest gwiazdka). Gdy funkcja jest wywoływana, wszystkie argumenty za obowiązkowymi, pozycyjnymi a przed nazwanymi, jeśli takie są, będą zebrane do krotki i przesłane do funkcji — tak więc wewnątrz funkcji args będzie referencją do, być może pustej, krotki:

```
#!/usr/bin/env python

def f(a, *args):
    print("Positional:", a, end=', ')
```

```
        if len(args) == 0:
            print("No var args;")
            return
        print("Var args:", end=' ')
        for i, v in enumerate(args):
            print(f'{i+1} -> {v};', end=' ')
        print()

f(1)
f(2, 'A', 1.5, 'B')
f(3, ('X', 'Y'))
f(4, *('X', 'Y'))
```

Program drukuje

```
Positional: 1, No var args;
Positional: 2, Var args: 1 -> A; 2 -> 1.5; 3 -> B;
Positional: 3, Var args: 1 -> ('X', 'Y');
Positional: 4, Var args: 1 -> X; 2 -> Y;
```

Zwróćmy uwagę na ostatnie dwa wywołania: w trzecim przesyłamy krotkę `('X', 'Y')` jako jeden argument, natomiast w czwartym rozpakowujemy tę krotkę do dwóch pojedynczych argumentów, które będą zebrane z powrotem do krotki args po stronie funkcji.

Na końcu listy parametrów funkcji można umieścić **kwargs (znowu, nazwa kwargs jest konwencją, ważne są dwie gwiazdki). Gdy wywołujemy funkcję, podajemy dowolną liczbę argumentów nazwanych (w postaci `k1=v1, k2=v2,...`). Będą one zebrane do słownika, który zostanie przesłany do funkcji — tak więc wewnątrz funkcji kwargs będzie referencją do, być może pustego, słownika:

Listing 20 AAW-ArgsKwargs/ArgsKwargs.py

```
#!/usr/bin/env python

def f(a, *args, **kwargs):
    print('Positional:', a, end=', ')
    if len(args) == 0:
        print('No var args;', end=' ')
    else:
        print('Var args:', end=' ')
        for i, v in enumerate(args):
            print(f'{i+1} -> {v};', end=' ')
    if len(kwargs) == 0:
        print('No kwargs;')
    else:
        print('KWargs:', end=' ')
        for k, v in kwargs.items():
            print(f'{k} : {v}; ', end = ' ')
        print()

f(1)
f(2, 'x', 'y')
```

```
21 f(3, fName='Sue', lName='Lee')
22 f(4, 'x', 'y', w=24, h=30)
23
24 lst = ['a', 'b']
25 dic = dict(sue="S", joe="J")
26 f(5, *lst, **dic)
```

W ostatnim wywołaniu argumenty są zebrane do krotki i słownika, które rozpakowujemy; będą one z powrotem zebrane do krotki args i słownika kwargs po stronie funkcji. Program drukuje:

```
Positional: 1, No var args; No kwargs;
Positional: 2, Var args: 1 -> x; 2 -> y; No kwargs;
Positional: 3, No var args; KWargs: fName : Sue;  lName : Lee;
Positional: 4, Var args: 1 -> x; 2 -> y; KWargs: w : 24;  h : 30;
Positional: 5, Var args: 1 -> a; 2 -> b; KWargs: sue : S;  joe : J;
```

Możemy również zażądać, aby niektóre argumenty były przesyłane zawsze bez nazwy, albo zawsze z nazwą.
Istnieje „specjalny” parametr oznaczany przez /, który wskazuje, że wszystkie argumenty poprzedzające pozycję tego parametru *nie mogą* być nazwane:

```
>>> def fun(a, b, c, /, d):
...     print(a, b, c, d)
...
>>> fun(1, 2, 3, 4)
1 2 3 4
>>> fun(1, 2, 3, d=4)
1 2 3 4
>>> fun(1, 2, c=3, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() got some positional-only arguments
  passed as keyword arguments: 'c'
```

Można też odwrotnie, zażądać, aby niektóre argumenty były podawane *zawsze* „po nazwie”. Jeden sposób na to jest następujący:

```
>>> def fun(*args, a):
...     print(args, a)
...
>>> fun(1, 2, 3, a=4)
(1, 2, 3) 4
>>> fun(a=4)
() 4
>>> fun(1, 2, 3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() missing 1 required keyword-only argument: 'a'
```

Musimy podać nazwę argumentu a, bo w przeciwnym przypadku byłby on „połknięty” przez krotkę args.

Ale jest też drugi parametr „specjalny”, oznaczany przez *, który mówi, że argumenty występujące *za* jego pozycją *muszą* być podawane „po nazwie”:

```
>>> def fun(a, b=2, *, c, d):
...     print(a, b, c, d)
...
>>> fun(1, c=3, d=4)
1 2 3 4
>>> fun(1, 2, c=3, d=4)
1 2 3 4
>>> fun(1, 2, 3, d=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fun() takes from 1 to 2 positional arguments but
3 positional arguments (and 1 keyword-only argument) were given
```

Jako przykład możemy podać wbudowaną funkcję **sorted**, która ma nagłówek

```
sorted(iterable, /, *, key=None, reverse=False)
```

co oznacza, że pierwszy argument (obiekt iterowalny) musi być podany bez żadnej nazwy, za to użycie któregoś z argumentów opcjonalnych (key i/lub reverse) wymaga podania jego nazwy

```
>>> lst = ['Jane', 'Sophia', 'Liz']
>>> sorted(lst, reverse=True)
['Sophia', 'Liz', 'Jane']
>>> sorted(lst, key=lambda e: e[-1])
['Sophia', 'Jane', 'Liz']
>>> sorted(lst, reverse=True, key=lambda e: e[-1])
['Liz', 'Jane', 'Sophia']
```

5.3 Lambdy

Innym sposobem tworzenia funkcji jest użycie wyrażeń lambda. Mimo protestów Guido van Rossuma, wyrażenia te zostały wprowadzone do Pythona pod naciskiem społeczności użytkowników (i miłośników języka Lisp).

Składnia jest bardzo prosta:

```
lambda args_list: expr
```

gdzie args_list jest listą oddzielonych przecinkami parametrów formalnych, a expr jest *jednym* wyrażeniem (czyli czymś, co ma wartość) w którym można użyć zadeklarowanych wcześniej parametrów. Zwracaną wartością funkcji reprezentowanej przez lambdę będzie wartość tego właśnie wyrażenia (nie używamy słowa **return**).

Ponieważ expr musi być *pojedynczym* wyrażeniem, nie można tu użyć, na przykład, instrukcji **if-else**; wyrażenia warunkowe są za to dozwolone:

```
pal = lambda ls: 'palindrom' if ls == ls[::-1] else 'not palindrom'

print(pal('rotor')) # prints 'palindrom'
print(pal('rover')) # prints 'not palindrom'
```

Lambdom można, jak w przykładzie powyżej, nadawać nazwy i używać ich dokładnie tak jak nazwy funkcji

```
q = lambda x: 0.25*x**2 - x - 3
print(q(2), q(3), q(-4))          # prints -4.0 -3.75 5.0
```

Wyrażenia lambda są typu **function** i mogą być używane jako (często anonimowe) funkcje definiowane *ad hoc*, podobnie jak w Javie lub, tym bardziej, w C++. Zauważmy jednak, że nie są tak użyteczne jak w tych językach, bo ich ciało musi być ograniczone do pojedynczego wyrażenia, którego wartość jest zwracana.

Każda lambda jest w zasadzie równoważna funkcji, jak **sq** i **lam** poniżej

```
>>> def sq(x):
...     return x*x
...
>>> lam = lambda x: x*x
>>> type(sq), type(lam)
(<class 'function'>, <class 'function'>)
>>>
>>> dir(sq)
['__annotations__', '__builtins__', '__call__', '__class__',
 '__closure__', '__code__', '__defaults__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>>
>>> dir(lam)
['__annotations__', '__builtins__', '__call__', '__class__',
 '__closure__', '__code__', '__defaults__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__get__', '__getattribute__', '__globals__',
 '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__kwdefaults__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__']
>>>
>>> sq.__name__, lam.__name__
('sq', '<lambda>')
```

Jak widzimy, te dwa obiekty są zasadniczo takie same, z wyjątkiem ich atrybutu `__name__`. Jednakże lambdy mogą być czasem użyte w kontekście, w którym nie moglibyśmy użyć definicji „zwykłej” funkcji — na przykład bezpośrednio na liście argumentów wywołania, jak w przykładzie poniżej

```
>>> def apply_n_times(f, n, x):
...     for i in range(n):
```

```
...         x = f(x)
...     return x
...
>>> apply_n_times(lambda x: x*x, 3, 2)
256
```

Oczywiście liczba parametrów lambd nie jest ograniczona do jednego; w poniższym przykładzie lambda ma dwa parametry i zwraca dwuelementową krotkę z sumą kwadratów argumentów jako pierwszym elementem, i kwadratem ich sumy jako drugim:

```
>>> myfun = lambda x, y: (x**2+y**2, (x+y)**2)
>>> myfun(4, -1)
(17, 9)
```

Lambdy, będąc obiektami, mogą być przesyłane do funkcji jako argumenty i zwracane z funkcji. W przykładzie poniżej, funkcja **quad** przyjmuje trzy współczynniki funkcji kwadratowej ax^2+bx+c, która to funkcja kwadratowa zwracana jest w postaci lambdy:

```
def quad(a, b, c):
    print('id of c is:', id(c))
    return lambda x: a*x*x + b*x + c

f, g, h = quad(1, 1, 1), quad(2, -1, 3), quad(-1, 4, -2)

print('\n', f(2), g(2), h(2))
print()
print('h.__closure__ is:', h.__closure__, '\n')
print('id of h.__closure__[2].cell_contents is:',
        id(h.__closure__[2].cell_contents))
print('h.__closure__[2].cell_contents is:',
    h.__closure__[2].cell_contents)
```

Program drukuje

```
id of c is: 140437638578416
id of c is: 140437638578480
id of c is: 140437638578320

 7 9 2

h.__closure__ is:
(<cell at 0x7fba2e1fbcd0: int object at 0x7fba2f8f80b0>,
 <cell at 0x7fba2e1fbca0: int object at 0x7fba2f8f8150>,
 <cell at 0x7fba2e1fbc70: int object at 0x7fba2f8f8090>)

id of h.__closure__[2].cell_contents is: 140437638578320
h.__closure__[2].cell_contents is: -2
```

Funkcja **quad** drukuje też ID zmiennej `c`, która jest zmienną *lokalną* w tej funkcji i powinna przestać istnieć po powrocie z niej.

Przeanalizujmy jednak obiekt reprezentujący lambdę **h**. Jego atrybut __closure__ (domknięcie) jest krotką zawierającą trzy „komórki" (*cells*) odpowiadające trzem lokalnym zmiennym, a, b i c, które istniały w funkcji **quad** gdy była wykonywana. Drukując zawartość cell_content trzeciej komórki (z indeksem 2), widzimy, że jej ID i wartość odpowiadają dokładnie zmiennej c, gdy była ona lokalną zmienną w funkcji **quad**. Tak więc zmienne lokalne z zakresu otaczającego potrzebne w definicji funkcji (w szczególności lambdy) są zapamiętywane w obiekcie reprezentującym tę funkcję i mogą być użyte nawet po powrocie z funkcji w której były zmiennymi lokalnymi.

Domknięcie jest tworzone jeśli funkcja (lambda) jest tworzona wewnątrz innej funkcji i „pamięta" referencje do obiektów będących zmiennymi lokalnymi. Jeśli nie jest funkcją zanurzoną, domknięcie nie jest tworzone.

Spójrzmy na następujący fragment kodu. Zauważmy, że tworzona lambda używa zmiennych a, b i c mimo że one nie istnieją! Lambda zawiera tylko ich *nazwy:* wartości będą zawsze odpowiadać *aktualnym* wartościom, gdy lambda jest wywoływana, tak więc ich modyfikacje pomiędzy wywołaniami będą widoczne, nawet jeśli odpowiadają zupełnie nowym obiektom innych typów:

```
1    quad = lambda x: a*x**2 + b*x + c # a, b, c nie istnieja!
2    a = 1; b = 1; c = 1               # int
3    print(quad(-2))                   # drukuje 3
4    a = 2.5; b = 2.5; c = -0.75       # teraz float
5    print(quad(-2))                   # drukuje 4.25
6    print(quad.__closure__)           # drukuje None
```

W linii 4 redefiniujemy zmienne a, b i c, tak, że odpowiadają one teraz obiektom o innym typie (teraz **float**) i kiedy lambda jest wykonywana, używane są *nowe* wartości. Żadne domknięcia nie są tworzone.

Jeśli jednak ten sam kod umieścimy wewnątrz funkcji, domknięcia *będą* tworzone

```
1    def d():
2        quad = lambda x: a*x**2 + b*x + c # a, b, c do not exist!
3        a = 1; b = 1; c = 1               # ints
4        print(quad(-2))                   # prints 3
5        a = 2.5; b = 2.5; c = -0.75       # now floats
6        print(quad(-2))                   # prints 4.25
7        print(quad.__closure__)           # now closure exists!
8    d()
```

jak widzimy z wyników tego programu:

```
3
4.25
(<cell at 0x7f5ef349b820: float object at 0x7f5ef3deddb0>,
 <cell at 0x7f5ef349b850: float object at 0x7f5ef3deddb0>,
 <cell at 0x7f5ef349b880: float object at 0x7f5ef3ded3f0>)
```

Lambdy, jako funkcje, mogą być przekazywane do funkcji i z nich zwracane — same te funkcje też mogą być lambdami. Zobaczmy następujący przykład

```
1    cmp = lambda f, g: lambda x: f(g(x)) # composition
2
3    fun1 = cmp(lambda x: x-1,  lambda x: x**2)
```

```
4     fun2 = cmp(lambda x: x**2, lambda x: x-1)
5
6     print(fun1(3), fun2(3))                # prints 8 4
```

W linii 1 tworzymy lambdę z dwoma parametrami, f i g. Odpowiadają one dwóm jednoargumentowym funkcjom. Sama lambda zwraca też lambdę — odpowiada ona złożeniu $f \circ g$ (złożenie funkcji jest zdefiniowane jako $(f \circ g)(x) = f(g(x))$). Wynikowej lambdzie nadajemy nazwę **cmp**.
W liniach 3 i 4 wywołujemy **cmp** przekazując dwie funkcje, jak to jest wymagane, i te funkcje też są lambdami: jedna odpowiada funkcji $x \mapsto x - 1$ a druga $x \mapsto x^2$ (przekazujemy je w obu wywołaniach w odwrotnej kolejności). Teraz **fun1** odpowiada $x \mapsto x^2 - 1$ a **fun2** funkcji $x \mapsto (x - 1)^2$ — zatem wynikiem jest '8 4'.

Lambdy są najczęściej używane gdy wołamy funkcje oczekujące funkcji jako jednego ze swoich argumentów. Na przykład metoda **sort** listy musi wiedzieć według jakiego kryterium ma porównywać elementy: ta informacja przekazywana jest jako funkcja, która zastosowana do elementów listy daje wartości według których te elementy mają być porównywane. Na przykład

```
>>> ls = [('Joe', 182, 79), ('Sue', 173, 59), ('Liz', 175, 62)]
>>> ls.sort(key=lambda person: person[2])
>>> ls
[('Sue', 173, 59), ('Liz', 175, 62), ('Joe', 182, 79)]
```

sortuje listę 3-krotek reprezentujących imiona, wzrost i wagę osób według trzeciego elementu krotki (o indeksie 2: waga):

```
[('Sue', 173, 59), ('Liz', 175, 62), ('Joe', 182, 79)]
```

Mimo, że treść lambdy musi być pojedynczym wyrażeniem, może ono w szczególności zawierać wywołanie tejże lambdy, a więc lambdy mogą być rekurencyjne, choć by to było możliwe, muszą być nazwane. Rekurencyjnych lambd używa sie rzadko, bo są zwykle mało czytelne, ale jest to możliwe. Na przykład funkcję silnia można zdefiniowac tak:

```
>>> fac = lambda n: 1 if n <= 1 else n*fac(n-1)
>>> fac(5)
120
>>> fac(45)
119622220865480194561963161495657715064383733760000000000
```

Po prawej stronie przypisania w pierwszej linii nazwa fac jest niezdefiniowana, ale *będzie* znana, gdy już po przypisaniu lambda będzie *użyta.*

5.4 Kilka funkcji „specjalnych".

W szczególności, lambdy są używane jako argumenty takich funkcji jak **filter**, **map**, **reduce**, **accumulate**[1] i również funkcji **sort**/**sorted**.
Są one używane jak funkcje, choć, ściśle biorąc, niektóre z nich, jak **map** czy **filter**, to tak naprawdę klasy (ale klasy *są* wywoływalne, więc nie jest to wcale takie dziwne):

[1] Są to podstawowe funkcje w językach funkcyjnych, jak Haskell.

```
>>> lst = [1,2,3,4,5,6]
>>>
>>> f = filter(lambda n: n%2 != 0, lst)
>>> type(f)
<class 'filter'>
>>> f
<filter object at 0x7fd978400e20>
>>> list(f)
[1, 3, 5]
>>>
>>> m = map(lambda n: n**n, lst)
>>> type(m)
<class 'map'>
>>> m
<map object at 0x7fd978403f10>
>>> list(m)
[1, 4, 27, 256, 3125, 46656]
```

5.4.1 *filter*

Funkcja **filter** pobiera obiekt iterowalny (coś, po czym można iterować) oraz funkcję (predykat), która dla dowolnego elementu obiektu zwraca **True** (lub coś, co jest „truthy”) lub **False** (lub coś, co jest „falsy”). Sama funkcja **filter** zwraca obiekt iterowalny, którego iteracja daje elementy wejściowego obiektu, ale tylko te dla których predykat jest spełniony. Jeśli predykat jest **None** (wartość domyślna), to użyta zostanie jako predykat identyczność, która wybierze tylko elementy „truthy” odrzucając te „falsy”.

Na przykład

Listing 21 AAZ-Filter/Filter.py

```
#!/usr/bin/env python

def is_prime(n):
    if n <= 1: raise ValueError
    if n <= 3: return True
    if n % 2 == 0 or n % 3 == 0: return False
    k = 5
    while k**2 <= n:
        if n % k == 0 or n % (k+2) == 0: return False
        k += 6
    return True

lst = [1, 5, 1, 0, '', {}, 2]

print(type(filter))           # <class 'type'>
a = filter(None, lst)
print(a)                      # <filter object at 0x7f569fabbc40>

for e in filter(None, lst):
    print(e, end=' ')         # only 'truthy': 1 5 1 2
```

```
print()

lst = ["Kyle", "Sue", "Mary", "Ada"]
for e in filter(lambda s: len(s) > 3, lst):
    print(e, end=' ')          # Kyle Mary
print()

tup = (3, 9, 12, 11, 27, 83, 787)
for e in filter(lambda n: is_prime(n), tup):
    print(e, end=' ')          # 3 11 83 787
print()

dic = dict(Alice = 15, Jane = 21, Kyle = 12, Mary = 28)
d = dict(filter(lambda item: item[1] >= 18, dic.items()))
print(d)                       # {'Jane': 21, 'Mary': 28}
```

Zauważmy, że w ostatnim przykładzie wynikowy obiekt iterowalny przekazaliśmy bezpośrednio do konstruktora słownika.

5.4.2 *map*

Funkcja **map** pobiera funkcję oraz jeden lub wiele obiektów iterowalnych, a zwraca również obiekt iterowalny, którego iteracja daje wyniki działania przekazaną funkcją na elementy wejściowego obiektu iterowalnego. Jeśli przekazaliśmy `n` obiektów iterowalnych, to argumentowość[1] funkcji też musi być `n`: wynikowy iterator zwraca wtedy krotki wartości, a krotek tych będzie tyle, ile jest elementów w najkrótszym przekazanym obiekcie iterowalnym.

Na przykład

```
lst = [1, 4, 7, 2, 6]
m = map(lambda x: (x-1)**2, lst)

print(map)          # <class 'map'>
print(m)            # <map object at 0x7f4c7f403a60>
print(list(m))      # [0, 9, 36, 1, 25]
```

Inny przykład, gdzie działamy funkcją formatującą (patrz rozdz. **??**, str. **??**) na elementy listy

```
lsf = [1.2345, 1.2223e2]
print(list(map(lambda x: f'{x:.1f}', lsf))) # ['1.2', '122.2']
```

Tu natomiast mamy dwa obiekty iterowalne

```
lsti = [ 1,    3,    2,      2,   2]
lsts = ['a', 'bc', 'def', 'ghij']

m = map(lambda x, y: x*y, lsti, lsts)
print(map)          # <class 'map'>
print(m)            # <map object at 0x7ffaf3583a90>
print(list(m))      # ['a', 'bcbcbc', 'defdef', 'ghijghij']
```

(otrzymaliśmy cztery elementy, bo tyle ma krótsza z przekazanych list).

[1] Czyli liczba parametrów.

5.4.3 *reduce*

Funkcja **reduce** (z modułu ***functools***) pobiera binarną funkcję, obiekt iterowalny oraz, opcjonalnie, inicjator (ziarno) — ang. *initializer.* — o domyślnej wartości None.

Jeśli inicjator został podany, staje się wartością początkową tak zwanego **akumulatora**. Następnie funkcja jest aplikowana do akumulatora i kolejnych elementów iteratora, a wynik jest za każdym razem przypisywany do akumulatora. W ten sposób obiekt iterowalny jest „redukowany” do pojedynczej wartości:

```
iterable = [1, 2, 3, 4]
init = 0
fun = lambda x, y: x + y

acc = init
for e in iterable:
    acc = fun(acc, e)
print(acc)                                    # 10

  # now with reduce
from functools import reduce
print(reduce(lambda x, y: x+y, iterable, 0))    # 10
```

Jeśli nie podamy ziarna, albo podamy None, to akumulator zostanie zainicjowany pierwszym elementem, po czym proces rozpocznie się od wywołania funkcji dla akumulatora i drugiego elementu:

```
iterable = [1, 2, 3, 4]
init = 0
fun = lambda x, y: x + y

it = iter(iterable)
acc = next(it)                                # pierwszy element
for e in it:                                  # it jest iteratorem
    acc = fun(acc, e)
print(acc)                                    # 10

  # now with reduce
from functools import reduce
print(reduce(lambda x, y: x+y, iterable))    # 10
```

W powyższym przykładzie wyniki są takie same, bo w pierwszym przypadku ziarnem był element neutralny dodawania; dla innych wartości wyniki będą oczywiście inne:

```
from functools import reduce

iterable = [1, 2, 3, 4]

print(reduce(lambda x, y: x+y, iterable))    # 10
print(reduce(lambda x, y: x+y, iterable, 5)) # 15
```

Inne przykłady:

```
from functools import reduce

lsi = [3, 6, -1, 3, 7, 2]

mn = reduce(lambda x, y: min(x, y), lsi) # min
mx = reduce(lambda x, y: max(x, y), lsi) # max
nb = reduce(lambda x, y: x+1, lsi, 0)    # number of elems

print(mn, mx, nb)           # -1 7 6


lss = ['Kate', 'Joanna', 'Sue', 'Mary']

  # longest word
ln = reduce(lambda x, s: x if len(x) > len(s) else s, lss)
  # initials
ii = reduce(lambda i, s: i + s[0], lss, '')
  # length of the longest word
lo = reduce(lambda l, s: l if l > len(s) else len(s), lss, 0)

print(ln, ii, lo)           # Joanna KJSM 6
```

5.4.4 *accumulate*

Funkcja **accumulate** (z modułu ***itertools***) jest podobna do **reduce**, ale zwraca nie pojedynczą wartość, ale obiekt iterowalny, którego iteracja zwraca wszystkie wyniki pośrednie. Pierwszym argumentem **accumulate** jest obiekt iterowalny, a drugim funkcja binarna, która domyślnie odpowiada dodawaniu (**operator.add**). Tak więc następujące dwa wywołania

```
from itertools import accumulate
lsi = [1, 2, 3]

print(list(accumulate(lsi)))                    # [1, 3, 6]
print(list(accumulate(lsi, lambda x, y: x + y))) # [1, 3, 6]
```

są równoważne.

W przykładzie poniżej liczby są wpłatami na konto oszczędnościowe. Każdego miesiąca doliczane jest 5% dotychczas zebranego kapitału oraz wpłata danego miesiąca. Zwrócony obiekt iterowalny daje stan konta po każdej wpłacie (funkcja **map** została tu użyta do ładniejszego formatowania liczb):

```
from itertools import accumulate

lsi = [150, 200, 300, 350]

a = accumulate(lsi, lambda summ, e: 1.05*summ + e)
print(type(a))
print(list(map(lambda e: round(e,1), a)))
```

Program drukuje

```
<class 'itertools.accumulate'>
[150, 357.5, 675.4, 1059.1]
```

Można również podać wartość początkową definiując trzeci, opcjonalny argument initial (argument obowiązkowo nazwany). Wtedy w wyniku otrzymamy o jedną wartość więcej:

```
from itertools import accumulate

lsi = [150, 200, 300, 350]

a = accumulate(lsi, lambda summ, e: 1.05*summ + e, initial=100)
print(list(map(lambda e: round(e,1), a)))
```

Teraz program drukuje

```
[100, 255.0, 467.8, 791.1, 1180.7]
```

5.4.5 *sorted*

Funkcja **sorted** pobiera obiekt iterowalny i zwraca jego (płytką) kopię w porządku wzrastającym.

Dla obiektów **list** jest też *metoda,* **sort**, która sortuje elementy „w miejscu” poprzez ich przestawianie i nie zwraca niczego (czyli zwraca **None**).

Ważne: Algorytm sortowania nie jest ustalony raz na zawsze[1] i może być w przyszłości zmieniony, ale *musi* być stabilny, czyli elementy równe zachowują swoje względne uporządkowanie.

Funkcja **sorting** używa operatora < do porównywania elementów, tak więc będzie działać dla liczb (prócz **complex**), napisów jak i innych typów dla których ten operator może być zastosowany (na przykład dla których metoda **__lt__** jest zdefiniowana).

```
lsiorig = [7, 2, 8, 1, 5]
lsorted = sorted(lsiorig)
print('original', lsiorig) # [7, 2, 8, 1, 5]
print('  sorted', lsorted) # [1, 2, 5, 7, 8]
```

Prócz sekwencji do posortowania, istnieją dwa dodatkowe parametry funkcji **sorted**, obydwa opcjonalne i obowiązkowo nazwane.

Jednym z nich jest reverse, o oczywistym znaczeniu i wartości domyślnej **False**:

```
lsiorig = [7, 2, 8, 1, 5]
lsorted = sorted(lsiorig, reverse=True)
print('original', lsiorig) # [7, 2, 8, 1, 5]
print('  sorted', lsorted) # [8, 7, 5, 2, 1]
```

Trzecim (też koniecznie nazwanym) parametrem funkcji **sorted** jest funkcja klucza key. Odpowiadającym mu argumentem powinna być jednoargumentowa funkcja,[2] która zastosowana do elementów sortowanej sekwencji da wartości, które będą porównywane zamiast oryginalnych elementów. Zauważmy jednak, że te wartości będą użyte tylko do porównywania, same elementy sekwencji nie będą modyfikowane tylko przestawiane.

Na przykład poniżej sortujemy krotki na dwa sposoby: według ich pierwszego elementu (imienia), a następnie według drugiego (wieku):

[1] Przez wiele lat był to Timsort – hybrydowy algorytm oparty na sortowaniu przez scalanie i wstawianie (opracowany przez by Tima Petersa w 2002 roku). Ten sam algorytm jest używany w systemie Android i Javie. W wersji 3.11 języka Python, został on zastąpiony przez algorytm Powersort (I. Munro i S. Wild).

[2] Zwana czasem *projektorem.*

```
ltup = [('Jane', 29), ('Alice', 31), ('Betty', 7)]
print(sorted(ltup, key=lambda t: t[0]))
print(sorted(ltup, key=lambda t: t[1]))
```

prints

```
[('Alice', 31), ('Betty', 7), ('Jane', 29)]
[('Betty', 7), ('Jane', 29), ('Alice', 31)]
```

W przykładzie poniżej sortujemy najpierw bez podawania funkcji klucza, czyli użyty zostanie operator < dla napisów (przypomnijmy, że wszystkie duże litery są „mniejsze" od wszystkich małych). Następnie sortujemy podając jako funkcję klucza metodę **lower**: elementy listy nie będą zmieniane, ale porównywanie będzie *case-insensitive*. Na koniec drukujemy oryginalną listę, aby pokazać, że nie uległa zmianie — **sorted** zwraca posortowaną *kopię:*

```
lstring = ['Jane', 'Hugh', 'bertha', 'henry', 'alice']
lsorted = sorted(lstring)
ltolowe = sorted(lstring, key = lambda s: s.lower())
print('   original', lstring)
print(' sort as is', lsorted)
print('case insens', ltolowe)
print('   original', lstring)
```

drukuje

```
   original ['Jane', 'Hugh', 'bertha', 'henry', 'alice']
 sort as is ['Hugh', 'Jane', 'alice', 'bertha', 'henry']
case insens ['alice', 'bertha', 'henry', 'Hugh', 'Jane']
   original ['Jane', 'Hugh', 'bertha', 'henry', 'alice']
```

Bardzo często kryteria porównywania są bardziej złożone; na przykład porównujemy za pomocą pewnego „głównego" kryterium, ale elementy, które wypadły przy tym równe chcemy posortować za pomocą „dodatkowego" kryterium. Aby to zrobić możemy wykorzystać fakt, że krotki *są* porównywalne (za pomocą operatora <) jeśli tylko kolejne ich elementy takie są. Tak więc funkcja klucza może zwracać krotki, które będą porównywane leksykograficznie:

Na przykład poniżej sortujemy słowa według ich długości. Te o równej długości będą dodatkowo posortowane alfabetycznie bez uwzględniania wielkości liter:

```
ls = ['jane', 'Bertha', 'henry', 'alice', 'Hugh']
print(sorted(ls, key=lambda s: (len(s), s.lower())))
```

drukuje

```
['Hugh', 'jane', 'alice', 'henry', 'Bertha']
```

Alternatywnie, dzięki gwarantowanej stabilności algorytmu sortowania, możemy najpierw posortować według *dodatkowego* kryterium, a potem tak posortowana listę, według kryterium „głównego".

Na przykład we fragmencie poniżej sortujemy napisy według ich długości, a te z równą długością alfabetycznie i bez uwzględniania wielkości liter:

```
ls = ['jane', 'Bertha', 'henry', 'alice', 'Hugh']
    # secondary: alphabetically, case-insensitively
ls = sorted(ls, key=lambda s: s.lower())
    # and now primary: length
ls = sorted(ls, key=lambda s: len(s))
print(ls)
```

drukuje, jak poprzednio

```
['Hugh', 'jane', 'alice', 'henry', 'Bertha']
```

W wielu językach programowania do sortowania używana jest funkcja, która dla dwóch obiektów zwraca liczbę ujemną, jeśli pierwszy z nich ma być traktowany jako „mniejszy", dodatnią jeśli to drugi ma być „mniejszy", a zero jeśli uważane mają być jako równe. Dla użytkowników przyzwyczajonych do tego rodzaju funkcji, istnieje specjalna funkcja,

cmp_to_key z modułu ***functools***, która pobiera binarną funkcję opisanego rodzaju a zwraca funkcję, która dla *jednego* argumentu daje wartość, której możemy użyć jako klucza sortującego:

```
from functools import cmp_to_key

def my_cmp(s1, s2):
      # by length
    if (d := len(s1) - len(s2)) != 0: return d
      # alphabetically, case-insensitively
    if   s1.lower() < s2.lower(): return -1
    elif s2.lower() < s1.lower(): return +1
    else                        : return  0

ls = ['jane', 'Bertha', 'henry', 'alice', 'Hugh']

ls = sorted(ls, key=cmp_to_key(my_cmp))
print(ls)
```

znów daje wynik

```
['Hugh', 'jane', 'alice', 'henry', 'Bertha']
```

5.5 Dekoratory funkcji.

Dekorator to funkcja (ogólniej – obiekt wywoływalny) pobierająca obiekt wywoływalny[1] jako argument i zwracająca też obiekt wywoływalny.

Na przykład:

```
def decorator(func):
    return func

def adder(a, b):
    """Return sum of arguments."""
```

[1]Klasy i ich metody też są wywoływalne, zatem one też mogą być dekorowane. Poniższe przykłady będą jednak dotyczyć funkcji.

```
    return a + b

add = decorator(adder)
print(add(5, 7))        # 12
print(add.__name__)     # adder
```

W tym trywialny przykładzie funkcja **add** zachowuje się zupełnie jak **adder**, choć jej własność `__name__` jest `adder`, jak w „oryginalnej" funkcji opakowanej funkcją **decorator**. Możemy zrobić coś mniej trywialnego, zwracając funkcję podobną do oryginalnej, ale z jakimiś cechami dodanymi lub zmodyfikowanymi — jest to na tyle użyteczne, że doczekało się narzędzi składniowych czyniących używanie dekoratorów prostszym i czytelniejszym: zamiast wywoływać jawnie dekorator, stosujemy *adnotację* do dekorowanej funkcji:

```
def decorator(func):
    return func

@decorator
def adder(a, b):
    """Return sum of arguments."""
    return a + b

print(adder(5, 7))  # 12
```

Tak więc

```
@decorator
def adder(a, b):
    """Return sum of arguments."""
    return a + b
```

jest równoważne

```
def adder(a, b):
    """Return sum of arguments."""
    return a + b
adder = decorator(adder)
```

Oczywiście, normalnie zwracana przez dekorator funkcja nie jest identyczna z wejściową, ale ma pewne cechy dodane lub zmienione — może to być, między innymi, rejestrowanie gdzieś naszej funkcji, modyfikowanie argumentów lub wartości zwracanej, logowanie lub zliczanie wywołań, zapamiętywanie rezultatów wywołań do późniejszego wykorzystania, itd.

Rozważmy

```
def counting(func):                         # dekorator
    counter = 0
    def modified_func(a, b):
        nonlocal counter
        counter += 1
        return func(a, b)
    return modified_func
```

```
@counting
def adder(a, b):
    """Return sum of arguments."""
    return a + b

print(adder(-1, 2), adder(5, 7))           # 1 12
print(adder.__closure__[0].cell_contents) # 2
print(adder.__name__)                      # modified_func
print(adder.__doc__)                       # None
```

Teraz dekorator **counting** zwraca wejściową funkcję, ale z licznikiem wywołań dodanym do jej domknięcia (zob. str. 76). Jest jednak mały problem: jej własność `__name__` jest `modified_func` zamiast `adder` i jej docstring został utracony. Można to naprawić „ręcznie”:

```
def counting(func):                        # decorator
    counter = 0
    def modified_func(a, b):
        nonlocal counter
        counter += 1
        return func(a, b)
    modified_func.__name__ = func.__name__
    modified_func.__doc__  = func.__doc__
    return modified_func

@counting
def adder(a, b):
    """Return sum of arguments."""
    return a + b

print(adder(-1, 2), adder(5, 7))           # 1 12
print(adder.__closure__[0].cell_contents) # 2
print(adder.__name__)                      # adder
print(adder.__doc__)                       # Return sum of arguments.
```

Byłoby to jednak kłopotliwe i nie bylibyśmy pewni, czy na pewno wszystkie metadane funkcji zostały prawidłowo przekopiowane. Rozwiązaniem jest użycie specjalnej funkcji **wrap** z modułu ***functools*** (która sama jest dekoratorem)

```
import functools

def counting(func):                        # dekorator
    counter = 0
    @functools.wraps(func)                 # opakowanie
    def modified_func(a, b):
        nonlocal counter
        counter += 1
        return func(a, b)
    return modified_func

@counting
def adder(a, b):
```

```
        """Return sum of arguments."""
        return a + b

    print(adder(-1, 2), adder(5, 7))           # 1 12
    print(adder.__closure__[0].cell_contents) # 2
    print(adder.__name__)                      # adder
    print(adder.__doc__)                       # Return sum of arguments.
```

Dodawanie danych do metadanych funkcji stwarza możliwość zaimplementowania „memoizacji'"(ang. *memoization,* czyli zapamiętywania wyników wywołań funkcji tak, aby można było wykorzystać je w przyszłości.[1]

Rozpatrzmy liczby Fibonacciego; jak wiemy, obliczanie ich rekurencyjnie jest skrajnie niewydajne, ponieważ liczba wywołań rośnie jak 2^N dla N-tej liczby Fibonacciego. Ale staje się to trywialne i praktycznie natychmiastowe, jeśli możemy zapamiętywać wszystkie poprzednie wyniki dla mniejszych liczb.

W przykładzie poniżej, dekorator **memo** dodaje słownik cache jako atrybut dekorowanej funkcji. W słowniku tym zapamiętywane są wyniki wywołań z krotkami argumentów jako kluczami. Tak więc dla argumentów takich samych jak w którymś z poprzednich wywołań, wynik może być zwrócony natychmiast, bez żadnych obliczeń. Słownik jest dostępny jako atrybut __wrapped__.cache dekorowanej funkcji:

Listing 22 ABS-Deco/Deco.py

```
#!/usr/bin/env python

import functools

def memo(func):
    func.cache = {}
    @functools.wraps(func)
    def memo(*args):
        if args not in func.cache:
            func.cache[args] = func(*args)
        return func.cache[args]
    return memo

@memo
def fibo(n):
    if n < 2: return n
    else    : return fibo(n - 1) + fibo(n - 2)

for i in range(50, 201, 50):
    print(f'fibonacci[{i:3}] = {fibo(i)}')

print(fibo.__wrapped__.cache)
```

Program drukuje

[1] Odpowiedni dekorator jest zresztą już zdefiniowany w bibliotece standardowej jako lru_cache (**l**east **r**ecently **u**sed cache) z modułu ***functools*** — patrz dokumentacja.

```
fibonacci[ 50] = 12586269025
fibonacci[100] = 354224848179261915075
fibonacci[150] = 9969216677189303386214405760200
fibonacci[200] = 280571172992510140037611932413038677189525
{(1,): 1, (0,): 0, (2,): 1, (3,): 2, (4,): 3, (5,): 5, (6,): 8,
 ...
 (197,): 662338693530854862817581421557052068990 77,
 (198,): 107168651819712326877926895128666735145224,
 (199,): 173402521172797813159685037284371942044301,
 (200,): 280571172992510140037611932413038677189525}
```

Zauważmy, że dekorator **memo** może być zastosowany do szerokiej gamy funkcji, niekoniecznie dwuargumentowych czy rekurencyjnych.[1]

Dekoratory można zagnieżdżać: dekorator zewnętrzny jest wtedy stosowany do wyniku dekoratora wewnętrznego, jak w przykładzie poniżej:

Listing 23 ABT-DecoNest/DecoNest.py

```
#!/usr/bin/env python

import functools

def italic(func):
    @functools.wraps(func)
    def _f(s):
        return '<i>' + func(s) + '</i>'
    return _f

def strong(func):
    @functools.wraps(func)
    def _f(s):
        return '<strong>' + func(s) + '</strong>'
    return _f

@italic
def ital(s):
    return s

@strong
def stro(s):
    return s

@italic
@strong
def itst(s):
    return s

@strong
@italic
```

[1] Nie należy próbować wykonania powyższego programu *bez* kaszowania!

```
32 def stit(s):
33     return s
34
35 print(ital(" just ital "))
36 print(stro(" just strong "))
37 print(itst(" ital and strong "))
38 print(stit(" strong and ital "))
```

który drukuje

```
<i> just ital </i>
<strong> just strong </strong>
<i><strong> ital and strong </strong></i>
<strong><i> strong and ital </i></strong>
```

Często chcielibyśmy przekazać do dekoratora dodatkową informację od której działanie dekoratora byłoby jakoś zależne. Możemy to zrobić tworząc „fabrykę dekoratorów" — funkcję która w zależności od przekazanej przez argumenty informacji zwraca „właściwe" dekoratory, które następnie aplikowane są do konkretnych funkcji.

W przykładzie poniżej funkcja **rangecheck**[1] przyjmuje dwa argumenty, mn i mx, które określają przedział [mn, mx] w jakim argumenty pewnej funkcji powinny się mieścić. Dla danych mn i mx, funkcja **rangecheck** tworzy i zwraca funkcję **checker**: to jest właśnie właściwy dekorator, który, będąc dekoratorem, pobiera funkcję i zwraca funkcję (tu nazwaną **wrap**), która „zastąpi" dekorowaną funkcję (**color** w linii 18, i **fRGBColor** w linii 22). Co jest ważne, referencje mn i mx mogą być użyte wewnątrz **wrap** (według reguły LEGB) a zatem wynikowa udekorowana funkcja może od nich zależeć:

Listing 24 ABU-DecoParam/DecoParam.py

```
1  #!/usr/bin/env python
2
3  import functools
4
5  def rangecheck(mn, mx):
6      def checker(func):                # decorator to be returned
7          @functools.wraps(func)
8          def wrap(*args):              # wraper of func
9              for i in args:
10                 if i < mn or i > mx:
11                     raise ValueError(
12                         f'Argument {i} in {args} is out of range')
13             return func(*args)        # if args ok, just call func
14         return wrap
15     return checker
16
17 @rangecheck(0, 255)
18 def color(r, g, b):
19     return f'Color {r=}, {g=}, {b=}'
```

[1] Funkcja ta *nie jest* dekoratorem: ona *zwraca* dekorator.

```
20
21 @rangecheck(0,1)
22 def fRGBColor(r, g, b):
23     return f'fColor {r=:.1f}, {g=:.1f}, {b=:.1f}'
24
25 try:
26     print(color(  2, 255,  0))
27     print(color(  0, 128, 34))
28     print(color(103,  12, 88))
29     print(color(  0, 256,  0))
30 except ValueError as e:
31     print(e)
32
33 print()
34
35 try:
36     print(fRGBColor(0.0, 0.9, 0.0))
37     print(fRGBColor(0.5, 0.8, 0.0))
38     print(fRGBColor(0.2, 0.1, 0.8))
39     print(fRGBColor(1.1, 0.9, 0.3))
40 except ValueError as e:
41     print(e)
```

Program drukuje

```
Color r=2, g=255, b=0
Color r=0, g=128, b=34
Color r=103, g=12, b=88
Argument 256 in (0, 256, 0) is out of range

fColor r=0.0, g=0.9, b=0.0
fColor r=0.5, g=0.8, b=0.0
fColor r=0.2, g=0.1, b=0.8
Argument 1.1 in (1.1, 0.9, 0.3) is out of range
```

Dekoratory są wykonywane gdy moduł jest ładowany (importowany), więc dekorowane funkcje są w module od razu gotowe do użycia. Ilustruje to poniższy przykład:

Listing 25 ABY-DecoTime/DecoTime.py

```
1  #!/usr/bin/env python
2
3  import functools
4
5  def decorator(func):
6      print('Decorating', func.__name__)
7      @functools.wraps(func)
8      def wrap(arg):
9          val = func(arg)
10         print('Decorated', func.__name__, ':', arg, '->', val)
```

```
11          return val
12      return wrap
13
14  @decorator
15  def funA(x):
16      return x
17
18  @decorator
19  def funB(x):
20      return x*x
21
22  print('\n*** Starting the program\n')
23
24  print('Calling funA with arg =', 4)
25  v = funA(4)
26
27  print('Calling funB with arg =', 5)
28  v = funB(5)
```

Jak widzimy z wydruku

```
Decorating funA
Decorating funB

*** Starting the program

Calling funA with arg = 4
Decorated funA : 4 -> 4
Calling funB with arg = 5
Decorated funB : 5 -> 25
```

linie

```
Decorating funA
Decorating funB
```

pojawiają się *przed* rozpoczęciem wykonywania właściwego programu.

Rozdział 6

Kolekcje składane i generatory

Kolekcje są podstawowymi strukturami danych w Pythonie, tak jak są nimi obiekty iterowalne i iteratory. Jak pamiętamy, cokolwiek co da się wysłać do wbudowanej funkcji **iter** i dostać iterator *jest* iterowalne. Iterator z kolei to cokolwiek, co można wysłać do wbudowanej funkcji **next** i dostać następny element lub, jeśli już nie ma elementów do zwrócenia, wyjątek **StopIteration**. Aby funkcja **next** zadziałała, iteratory powinny mieć zaimplementowaną *metodę* **__next__**; jeśli jej nie ma, interpreter spróbuje jeszcze użyć metody **__getitem__** (która pobiera – liczony od zera – indeks).
Iteratory są też iterowalne, to znaczy funkcja **iter** zwraca dla nich iterator, najczęściej po prostu ten sam obiekt.

Zwykle nie wywołujemy funkcji **iter** i **next** „ręcznie", a używamy pętli **for**, która tak naprawdę jest pętlą **while** i „pod spodem" te funkcje wywołuje:

```
iterable = [1, 2, 3] # lista jest iterowalna
print('for loop:   ', end='')

for e in iterable:
    print(e, end=' ')

## równoważna pętla while

print('\nwhile loop: ', end='')

itr = iter(iterable)
while True:
    try:
        e = next(itr)
    except StopIteration:
        break
    else:
        print(e, end=' ')
del itr
```

drukuje

```
for loop:   1 2 3
while loop: 1 2 3
```

Składane kolekcje i generatory, jak pętle, są używane do iteracji. Pobierają one kolejne elementy z jakiegoś źródła i przetwarzają je po kolei.

6.1 Składanie kolekcji

Składanie kolekcji (po angielsku *comprehension*[1]) daje nam zwartą składnię służącą do tworzenia list, zbiorów i słowników z obiektów iterowalnych w deklaratywny/funkcjonalny sposób.
Składanie zawsze działa na istniejącym obiekcie iterowalnym i iterując po jego elementach daje rezultaty ich przetwarzania, które są zbierane do listy, zbioru lub słownika.

[1] Termin wzięty z *set comprehension* w matematyce.

Tak jak pętle, mogą być zagnieżdżane. Jeż też możliwe określanie warunków, jakie muszą być spełnione, aby dany rezultat wszedł do wynikowej kolekcji.

6.1.1 Składanie list

Składanie listy ma następującą postać

```
[ expr for var in iterable]
```

gdzie iterable jest obiektem iterowalnym (na przykład listą), var jest referencją do kolejnych elementów tego obiektu, a expr jest wyrażeniem (zwykle zależnym od var), którego wartość stanie się kolejnym elementem wynikowej listy.
Na przykład

```
>>> iterable = [1, 2, 3, 4, 5, 6]
>>> [val**2 for val in iterable]
[1, 4, 9, 16, 25, 36]
```

lub

```
>>> iterable = [ [1, 2], [3], [4, 5, 6] ]
>>> [sum(v) for v in iterable]
[3, 3, 15]
```

Za iterable można umieścić warunek, od którego spełnienia zależy, czy expr z danej iteracji wejdzie czy nie wejdzie do wynikowej listy:

```
>>> [v**2 for v in range(1,10) if v % 2]
[1, 9, 25, 49, 81]
```

Tu składamy kwadraty liczb z zakresu $[1, 9]$, ale tylko dla liczb nieparzystych (dla których v%2 jest 1, co jest interpretowane jako True).
Zauważmy, że zmienna użyta do iteracji (w naszym przypadku v) jest zmienną lokalną wewnątrz wyrażenia definiującego listę składaną; nie ma tu konfliktu z ewentualną inną zmienną o tej samej nazwie w zakresie otaczającym:

```
>>> v = 'Hello, World'
>>> [v**2 for v in range(1,10) if v % 2]
[1, 9, 25, 49, 81]
>>> v
'Hello, World'
```

Iteracje mogą być zagnieżdżane i dla każdej z nich można określać warunki:

```
>>> ls = [ [1, 2], 'abc', [3, 4, 5, 6,], 'xy' ]
>>> [k for v in ls if isinstance(v, list) for k in v if k%2 == 0]
[2, 4, 6]
```

co jest równoważne

```
ls = [ [1, 2], 'abc', [3, 4, 5, 6,], 'xy' ]
res = []
for v in ls:
    if isinstance(v, list):
        for k in v:
            if k % 2 == 0:
                res.append(k)
print(res)
```

i drukuje

```
[2, 4, 6]
```

Aby oswoić się ze składaniem list rozpatrzmy przykłady.

Przykład 1
Kwadraty liczb:

```
>>> [x*x for x in range(2,11)]
[4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Przykład 2
Kwadraty liczb parzystych:

```
>>> [x*x for x in range(2, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

Przykład 3
Różne potęgi dla liczb parzystych i nieparzystych:

```
>>> [x*x if x%2 else x*x*x for x in range(1,7)]
[1, 8, 9, 64, 25, 216]
```

Zauważmy wyrażenie x%2 jako warunek — będzie ono zinterpretowane jako True dla liczb nieparzystych.

Przykład 4
Konwersja napisów na liczby:

```
>>> [float(x) for x in ['3.3' , '1.25e2', 'NaN', '-1', '-INF']]
[3.3, 125.0, nan, -1.0, -inf]
```

Przykład 5
Wybieranie tylko liczb albo tylko napisów:

```
>>> [x for x in [1, 'a', 'bc', 1.2e2] if isinstance(x, int | float)]
[1, 120.0]
>>> [x for x in [1, 'a', 'bc', 1.2e2] if isinstance(x, str)]
['a', 'bc']
```

Przykład 6
Liczby cyfr w ogromnych liczbach:

```
>>> from math import factorial
>>> [len(str(factorial(x))) for x in range(100, 111)]
[158, 160, 162, 164, 167, 169, 171, 173, 175, 177, 179]
```

Przykład 7
Liczby pierwsze <= 31

```
>>> [k for k in range(2, 32)
...     if all(k%m != 0 for m in range(2, int(k**0.5)+1))]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```

Uwaga: funkcja **all** pobiera predykat i obiekt iterowalny i zwraca **True** wtedy i tylko wtedy, gdy wszystkie elementy spełniają dany predykat.

Przykład 8
Usuwanie spacji:

```
>>> ''.join([x for x in 'to be or not to be' if x != ' '])
'tobeornottobe'
```

Przykład 9
Wspólne elementy dwóch list:

```
>>> ls1 = [1, 'a', 3, 'b', 5, 'c']
>>> ls2 = [4, 'c', 5, 'f', 8]
>>> [a for a in ls1 if a in ls2]
[5, 'c']
```

Przykład 10
Największe elementy list:

```
>>> lst = [[1, 4, 3, 2], [19 , -1, 2], [-1, -20, -6], [5, 9]]
>>> [max(x) for x in lst]
[4, 19, -1, 9]
```

Przykład 11
Konwersja macierzy napisów na macierz liczb:

```
>>> ls = [['4', '8'],
...       ['4', '2', '28'],
...       ['1', '12'],
...       ['3', '6', '2']]
>>> [[int(e) for e in row] for row in ls]
[[4, 8], [4, 2, 28], [1, 12], [3, 6, 2]]
```

Zauważmy, że wartościami wyjściowej listy są tu listy składane.

Przykład 12
Spłaszczanie list:

```
>>> lst = [[1, 2, 3, 4], [5, 6], [7, 8, 9]]
>>> [ch for ls in lst for ch in ls]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Przykład 13
Pary z imionami na tę samą literę:

```
>>> boys  =  ['Adam', 'Burt', 'Joe', 'Jim', 'Sam']
>>> girls =  ['Bea', 'Jill', 'Ava', 'Eve']
>>> [(she, he) for she in girls for he in boys if she[0] == he[0]]
[('Bea', 'Burt'), ('Jill', 'Joe'), ('Jill', 'Jim'), ('Ava', 'Adam')]
```

Przykład 14
Trójki pitagorejskie dla a < b < c <= 50:

```
>>> from math import gcd
>>> def gcd3(a, b, c): return gcd(a, gcd(b, c))
...
>>> [(a, b, c) for a in range(1,49)
...            for b in range(a+1, 50)
...            for c in range(b+1, 51)
...                if a*a + b*b == c*c and gcd3(a, b, c) == 1]
[(3, 4, 5), (5, 12, 13), (7, 24, 25), (8, 15, 17),
(9, 40, 41), (12, 35, 37), (20, 21, 29)]
```

Zauważmy, że wybieramy tylko trójkąty parami niepodobne.

6.1.2 Składanie zbiorów

Składanie zbiorów ma składnię podobną do składania list, ale wartości expr są zbierane do zbiorów (czyli bez powtórzeń). Używamy tu nawiasów klamrowych, a nie kwadratowych.

Przykłady:

Przykład 1
Unikalne słowa:

```
>>> ss = ['Warsaw Paris Berlin', 'Praha Warsaw Berlin Caen']
>>> {word for s in ss for word in s.split()}
{'Paris', 'Warsaw', 'Praha', 'Berlin', 'Caen'}
```

Przykład 2
Wspólne elementy list:

```
>>> list1 = [1, 3, 7, 4]
>>> list2 = [3, 4, 5, 7]
>>> list3 = [4, 5, 6, 7]
>>> {x for x in list1 if x in list2 and x in list3}
{4, 7}
```

Przykład 3
Zbiór polskich samogłosek w tekście:

```
>>> string = 'Pozbądź się zbędnych kilogramów'
>>> {x for c in string if (x := c.lower()) in 'aąeęiouy'}
{'ę', 'y', 'ą', 'a', 'i', 'o'}
```

Przykład 4
Unikalne litery w tekście:

```
>>> string = 'carrot: 20; apples: 30'
>>> {c for c in string if c.isalpha()}
{'s', 'r', 'e', 'c', 'p', 'a', 'l', 't', 'o'}
```

Przykład 5
Iloczyn kartezjański zbiorów:

```
>>> l1 = [1, 2, 3]
>>> l2 = ['a', 'b']
>>> {(x, y) for x in l1 for y in l2}
{(1, 'b'), (3, 'a'), (2, 'a'), (2, 'b'), (1, 'a'), (3, 'b')}
```

Przykład 6
Zbiór liczb w liście:

```
>>> vals = [1, 'a', 2, 'b', 3, 'c']
>>> {x for x in vals if isinstance(x, int | float)}
{1, 2, 3}
```

6.1.3 Składanie słowników

W końcu, oddzielone dwukropkiem *pary* wartości

```
{ expr1 : expr for var in iterable }
```

składane są do słownika (tak jak dla zbiorów, używamy tu nawiasów klamrowych).

Przykłady:

Przykład 1
Słownik liczba → parzystość:

```
>>> nums = [1, 2, 3, 4]
>>> {n : 'odd' if n%2 else 'even' for n in nums}
{1: 'odd', 2: 'even', 3: 'odd', 4: 'even'}
```

Przykład 2
Zliczanie liter w tekście:

```
>>> string = 'to be or not to be'
>>> {c : string.count(c) for c in set(string) if c.isalpha()}
{'r': 1, 'n': 1, 'b': 2, 'e': 2, 't': 3, 'o': 4}
```

Przykład 3
Słownik nazwa → jej długość:

```
>>> names = ['Argentina', 'Poland', 'Spain', 'Italy']
>>> {name: len(name) for name in names}
{'Argentina': 9, 'Poland': 6, 'Spain': 5, 'Italy': 5}
```

Przykład 4
Konwersja krotek do słownika:

```
>>> lst = [('a', 'Alice'), ('b', 'Bea'), ('c', 'Cecilia')]
>>> {k: v for k, v in lst}
{'a': 'Alice', 'b': 'Bea', 'c': 'Cecilia'}
```

Przykład 5
Stopnie z wyników kolokwium:

```
>>> scores = {'Eve': 83, 'Bob': 47, 'Sue': 65, 'Dave': 77}
>>> {name:
...      '5'  if score >= 90 else
...      '4+' if score >= 80 else
...      '4'  if score >= 70 else
...      '3+' if score >= 60 else
...      '3'  if score >= 50 else '2'
...  for name, score in scores.items()}
{'Eve': '4+', 'Bob': '2', 'Sue': '3+', 'Dave': '4'}
```

Przykład 6
Filtrowanie słownika:

```
>>> dct = {'a': 9, 'b': 12, 'c': 7, 'd': 14}
>>> {k: v for k, v in dct.items() if v <= 10}
{'a': 9, 'c': 7}
```

Przykład 7
Odwracanie słownika:

```
>>> dct = {'Mary': 'John', 'Kate': 'Bill', 'Sophia': 'Kevin'}
>>> {v : k for k, v in dct.items()}
{'John': 'Mary', 'Bill': 'Kate', 'Kevin': 'Sophia'}
```

6.2 Funkcje generujące

Generatory są specjalnego typu obiektami, które mogą być użyte do „leniwego” zwracania sekwencji wartości. „Leniwe” znaczy tu, że kolejne elementy są wyliczane i zwracane tylko kiedy są potrzebne, na żądanie, a nie wszystkie zawczasu. Generatory są iterowalne — mają metodę `__iter__` zwracającą iterator. Główne korzyści z generatorów to:

- Wstrzymują działanie dopóki następna wartość nie jest potrzebna; wstrzymane nie zużywają czasu procesora, więc są całkowicie „leniwe”.
- Obliczają tylko wartości, których zażądano; wartości, które nie będą potrzebne w ogóle nie są wyliczane.
- Reprezentują strumienie danych o nieograniczonej długości, ponieważ nie przechowują nigdzie wartości, które mają być zwrócone; kiedy następna wartość jest wymagana, wyliczają tylko tę jedną wartość.

Z drugiej strony

- Użytkownik zazwyczaj nie wie, ile wartości pozostało do zwrócenia dopóki generator nie wyczerpie się.
- Do uzyskiwania elementów nie można stosować indeksów i wycinków.
- Aby uzyskać zadany element, trzeba przetworzyć wszystkie wcześniejsze.
- Generatora nie można zrestartować („przewinąć”).

Definicja funkcji generującej wygląda jak definicja funkcji, tyle że zamiast instrukcji **return** mamy tu **yield** — właśnie po obecności instrukcji **yield** interpreter poznaje, że jest to funkcja generująca, a nie „zwykła" funkcja (choć **return** też może w funkcji generującej występować). Wywołanie takiej „funkcji" też zachowuje się inaczej niż wywołanie „normalnej" funkcji: wywołanie zwraca obiekt typu **generator** bez wykonywania jakiegokolwiek kodu!

Zwrócony generator jest iterowalny: ma funkcję **__iter__**, która zwraca iterator, który z kolei ma metodę **__next__** dającą kolejne elementy zwracane przez **yield**. Wykonanie kodu generatora rozpoczyna się, gdy metoda **__next__** zostanie wywołana po raz pierwszy. Za każdym razem, gdy napotkana zostanie instrukcja **yield**, wartość wyrażenia za **yield** jest zwracana, a wykonywanie kodu generatora jest wstrzymywana. Jednak aktualny stan generatora jest zachowywany i wykonywanie zostanie wznowione po ponownym wywołaniu **__next__** dokładnie w miejscu gdzie się zatrzymało:

```
>>> def squares():
...     i = 1
...     while True:
...         yield i*i
...         i += 1
...
>>> gen = squares()
>>> type(squares), type(gen)
(<class 'function'>, <class 'generator'>)
>>> hasattr(gen, '__iter__')
True
>>> it = iter(gen)
>>> type(it)
<class 'generator'>
>>> hasattr(it, '__next__')
True
>>> it.__next__()
1
>>> next(it)
4
```

Zauważmy, że **yield** zachowuje się jak **return**, ale wykonanie zostaje tylko wstrzymane i wywołanie **next** wznawia je w miejscu, gdzie zostało wstrzymane (aż do napotkania znów instrukcji **yield**). Gdy przepływ sterowania wychodzi z funkcji (w sposób „naturalny" lub poprzez wykonanie **retrun**), podnoszony jest wyjątek **StopIteration**:

```
>>> def squares(n):
...     i = 1
...     while i < n:
...         yield i*i
...         i += 1
...
>>> gen = squares(3)
>>> it = iter(gen)
>>> next(it)
1
>>> next(it)
```

```
4
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Aby zatrzymać iterację, możemy też użyć „normalnej” instrukcji **return** — wtedy też dostaniemy wyjątek **StopIteration**:

```
>>> def squares():
...     i = 1
...     while True:
...         yield i*i
...         i += 1
...         if i == 3: return
...
>>> gen = squares()
>>> next(gen)
1
>>> next(gen)
4
>>> next(gen)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Inny przykład — generator liczb Fibonacciego:

```
def Fibo(n):
    a, b, count = 0, 1, 0
    while True:
        if (count > n): return
        yield a
        a, b = b, a + b
        count += 1

fib = Fibo(13)
for e in fib:
    print(e, end=' ')
print()
```

drukuje

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233
```

(wyjątek **StopIteration** został obsłużony przez pętlę **for**, więc go nie zobaczymy).

Jest jednak różnica pomiędzy użyciem **return** a po prostu wyjściem z generatora. Jeśli używamy **return**, możemy zwrócić jakąś *wartość,* i będzie ona zapamiętana jako atrybut obiektu-wyjątku; na przykład

```
def Fibo():
    a, b, count = 0, 1, 0
    while True:
```

```
            if (count > 2): return f'{count=}'
            yield a
            a, b = b, a + b
            count += 1

fib = Fibo()
print(next(fib))
print(next(fib))
print(next(fib))
print(next(fib))
```

drukuje

```
0
1
1
Traceback (most recent call last):
File "/usr/lib/python3.10/idlelib/run.py", line 578, in runcode
  exec(code, self.locals)
File "/home/werner/p.py", line 13, in <module>
  print(next(fib))
StopIteration: count=3
```

6.2.1 Generatory składane

Generatory mogą też być tworzone za pomocą składni podobnej do tej, jakiej używamy do składania list; wystarczy zamiast nawiasów kwadratowych użyć okrągłych (nazywamy to **generator expression**):

```
>>> g = (x**2 for x in [2, 4, 7])
>>> type(g)
<class 'generator'>
>>> for i in g:
...     print(i)
4
16
49
```

Zauważmy, że nie ma tu jawnie instrukcji **yield** i otrzymany generator może być iterowany jak lista (bo ma metody **__iter__** i **__next__**). Główną korzyścią jest to, że gdy tworzona jest lista, jest tworzona od razu ze wszystkimi elementami: generatory wyliczają kolejne elementy na żądanie, bez ich przechowywania gdziekolwiek. Tak więc nie zużywają praktycznie pamięci i tworzone są bardzo szybko

```
>>> from sys import getsizeof
>>> lst = [x**3 for x in range(10_000_000)]
>>> gen = (x**3 for x in range(10_000_000))
>>> lst.__sizeof__()
89095144
>>> gen.__sizeof__()
88
>>> getsizeof(lst)
89095160
```

```
>>> getsizeof(gen)
104
```

Jak widzimy, funkcja **getsizeof** zwraca trochę większy rozmiar niż metoda **__sizeof__**. W rzeczywistości wewnętrznie korzysta ona z niej, ale dodaje rozmiar dodatkowej informacji przeznaczonej dla odśmiecacza (zwykle jest to 16 bajtów, niezależnie od rozmiaru obiektu).

Poniższy przykład pokazuje, jak utworzyć generator kolejnych liczb ciągu Fibonacciego:

Listing 26 ABA-GenFib/GenFib.py

```
#!/usr/bin/env python3

def fibonacci(n):
    curr = 0
    prev = 1
    count = 0
    while count <= n:
        yield curr
        prev, curr = curr, prev + curr
        count += 1

fibogen = fibonacci(100)
print(type(fibonacci), type(fibogen))
for i, f in enumerate(fibogen):
    print('{:3d}:{:22d}'.format(i, f))
```

Program drukuje

```
<class 'function'> <class 'generator'>
  0:                     0
  1:                     1
  2:                     1
  3:                     2
  4:                     3
  5:                     5
  6:                     8
  7:                    13
            ...

 95:  31940434634990099905
 96:  51680708854858323072
 97:  83621143489848422977
 98: 135301852344706746049
 99: 218922995834555169026
100: 354224848179261915075
```

Inny przykład: generator czytający plik tekstowy i zwracający kolejne słowa tekstu, ale ignorując duplikaty:

Listing 27 ABL-UniqueWords/UniqueWords.py

```
#!/usr/bin/env python

def uniqueLines(inpFile):
    prevWords = set()
    with open(inpFile) as file:
        for line in file:
              # get rid of everything but letters and spaces
            line = ''.join([x.lower() for x in line
                            if x.isalpha() or x == ' '])
            for word in line.split():
                if word not in prevWords:
                    prevWords.add(word)
                    yield word           # yielding new word

for w in uniqueLines('UniqueWords.txt'): print(w)
```

Jeśli plik ***UniqueWords.txt*** zawiera

```
Kraków Paris Zürich
Warszawa Berlin
Paris
Kraków Zürich Warszawa
```

to wynikiem będzie

```
kraków
paris
zürich
warszawa
berlin
```

Inny przykład: tym razem tworzymy funkcję generującą **limiter**, która pobiera, jako pierwszy argument, inną funkcję generującą. Następnie wywołuje tę funkcję i iteruje po otrzymanym generatorze zadaną liczbę razy.[1] Zauważmy, że w takim razie możemy przekazać do naszej funkcji nawet potencjalnie nieskończony iterator; na przykład, jak poniżej, generator **Fibo** zwracający *ad infinitum* kolejne liczby Fibonacciego lub **Die** rzucający w nieskończoność kostką do gry:

Listing 28 ABR-GenOfGen/GenOfGen.py

```
#!/usr/bin/env python

from random import randint

def limiter(generator, n):
    g = generator()
    for i in range(n):
        yield next(g)
```

[1] Coś podobnego do **Stream::limit** z Javy.

```
9
10 def Fibo():
11     a, b = 0, 1
12     while True:
13         yield a
14         a, b = b, a+b
15
16 def Die():
17     while True:
18         yield randint(1, 6)
19
20 for e in limiter(Fibo, 10):
21     print(e, end=' ')
22 print()
23
24 for e in limiter(Die, 10):
25     print(e, end=' ')
26 print()
```

Program drukuje

```
0 1 1 2 3 5 8 13 21 34
2 1 2 5 1 4 3 3 1 2
```

6.3 Komunikacja z generatorami

Generatory nie tylko zwracają wartości, ale mogą też przyjmować je od strony wywołującego. Wartości są wysyłane do generatora poprzez wywołanie na nich metody **send**. Metoda ta wysyła wartość *do* generatora, ale też zwraca wartość otrzymaną *od* generatora. Aby móc wysyłać i otrzymywać wartości od generatora, wymagane jest by był on wstrzymany na linii zawierającej *wyrażenie* **yield** — wyrażenie, a więc na przykład po prawej stronie przypisania. Takie generatory są czasem nazywane **korutynami** (ang. *coroutine*).

Aby to uzyskać, należy generator „zapoczątkować" (ang. *prime*) poprzez wywołanie funkcji **next** (lub metody **send(None)**, co jest równoważne).

Przypuśćmy, że w linii, na której generator jest wstrzymany jest wyrażenie **yield** w formie `yield expr` Wtedy, po wywołaniu metody **send(v)**:

- Wartość `v` staje się wartością wyrażenia `yield expr` (ale *nie* samego `expr`!);[1]
- Wykonanie jest wznawiane aż do napotkania znów wyrażenia `yield expr` (lub zgłaszany jest wyjątek **StopIteration**, jeśli przepływ sterowania wychodzi z generatora bez napotkania na takie wyrażenie);
- Aktualna wartość `expr` jest zwracana do wywołującego;
- Wykonanie funkcji jest znów wstrzymywane.

Rozpatrzmy przykład ilustrujący to wszystko w akcji:

[1] Wartość wyrażenia `expr` i wartość wyrażenia `yield expr` to dwie różne rzeczy!

Listing 29 ABO-TwoWayGener/TwoWayGener.py

```
#!/usr/bin/env python

def twoWayGener():
    print("generator started")
    value = 'Init'
    while True:
        print('Ready to recive a value, value =', value)
        x = 10 + (yield value)
        print('Resuming execution; value =', value, 'x =', x)
        value = x

gen = twoWayGener()
ret = next(gen) # (or gen.send(None)) - priming
print('After priming: result =', ret, end='\n\n')

for n in range(3):
    print('Main: sending', n)
    ret = gen.send(n)
    print('Main: result =', ret, end='\n\n')
```

drukuje

```
generator started
Ready to recive a value, value = Init
After priming: result = Init

Main: sending 0
Resuming execution; value = Init x = 10
Ready to recive a value, value = 10
Main: result = 10

Main: sending 1
Resuming execution; value = 10 x = 11
Ready to recive a value, value = 11
Main: result = 11

Main: sending 2
Resuming execution; value = 11 x = 12
Ready to recive a value, value = 12
Main: result = 12
```

Następny przykład jest nieco bardziej złożony. Budujemy generator zwracający losowe elementy z przekazanej jako argument listy. Dodatkowo, jeśli wyślemy do generatora wartość nie będącą **None**, zostanie ona dodana do listy z której losowane są elementy:

Listing 30 ABP-Chooser/Chooser.py

```
1  #!/usr/bin/env python
2
3  from random import choice
4
5  def chooser(aList):
6      item = None
7      while True:
8          if item != None:
9              if item not in aList:
10                 aList.append(item)                    ❶
11             item = yield item                         ❷
12         else:
13             item = yield choice(aList)                ❸
14
15 ch = chooser(['Joe', 'Ben', 'Alice'])
16 # priming
17 next(ch)                                             ❹
18
19 for i in range(3):
20     print(next(ch)) # equivalently: print(ch.send(None))
21
22 print('*** Adding Eve and Kenny to the list')
23 ch.send('Eve')                                       ❺
24 ch.send('Kenny')                                     ❻
25
26 for i in range(4):                                   ❼
27     print(next(ch)) # equivalently: print(ch.send(None))
```

drukuje, na przykład, bo zwracane wartości są losowane:

```
Alice
Alice
Ben
*** Adding Eve and Kenny to the list
Kenny
Joe
Eve
Alice
```

Przeanalizujmy, co się tutaj dzieje. Zmienna item jest None, więc po „zapoczątkowaniu” (linia ❹) generator zatrzymuje się na linii ❸.
Teraz, w pętli, posyłamy do generatora None (next jest równoważne wysłaniu None): ta wartość staje się wartością wyrażenia `yield choice(aList)`, generator jest wznawiany, przypisuje to None do item i dociera znów do linii ❸. Teraz zwraca do wywołującego wartość `choice(aList)` i znów jest wstrzymany. I tak trzy razy.
Teraz wysyłamy `'Eve'` (linia ❺).
Ta wartość staje się wartością wyrażenia `yield choice(aList)` i jest przypisywana do item czyniąc tę zmienną nie-None. Wykonywanie jest wznawiane i dochodzi tym razem do linii ❶, dodając `'Eve'` do listy, następnie dochodzi do linii ❷, zwraca `'Eve'`

do wywołującego (który to ignoruje) i jest wstrzymywane. Teraz wysyłamy `'Kenny'` (❻). Ta wartość staje się wartością wyrażenia `yield item` i jest przypisywana do item, a wobec tego znowu dochodzimy do linii ❶ i ❷. Następna wartość wysłana do generatora będzie już None (z pętli w linii ❼), więc item będzie już None i znów trafimy do linii ❸ i zwrócony zostanie kolejny przypadkowy element listy. I tak powtarza się cztery razy.

Pamiętanie o „startowaniu" korutyn przed wysłaniem do nich danych może być kłopotliwe i można łatwo o tym zapomnieć. Standardową metodą radzenia sobie z tym problemem jest użycie dekoratora, który pobiera korutynę i zwraca gotowy generator już „wystartowany", jak to jest zilustrowane poniżej. Program drukuje też dodatkowe informacje ułatwiające zrozumienie logiki jego działania.

Korutyna **longlines** przyjmuje teksty wysyłane do niej i drukuje tylko takie, które zawierają więcej niż n słów. Aby jej używać bez „startowania", dekorujemy funkcję generującą **longlines** dekoratorem **coroutine**. Wywołuje on przekazaną funkcję generującą, otrzymuje generator, wywołuje na nim raz **next** i zwraca ten generator, który już nie potrzebuje „startowania":

Listing 31 ABV-Priming/Priming.py

```
#!/usr/bin/env python

import functools

def coroutine(func):
    print('** 2 ** Executing coroutine')
    @functools.wraps(func)
    def primed(*args, **kwargs):
        p = func(*args, **kwargs)     # p is a generator
        print('** 4 ** From primed')
        next(p)     # priming
        return p    # returning already primed generator
    return primed

print('** 1 ** Defining longlines')

@coroutine
def longlines(n):
    """Select lines with more than n words."""
    print('** 5 ** Longlines advances to yield')
    while True:
        line = yield
        if len(line.split()) > n: print('Got long line:', line)

print('** 3 ** Main starting, creating generator')
gen = longlines(3)
print(type(gen))
print(type(longlines), longlines.__name__, longlines.__doc__, '\n')

gen.send("one two")
```

```
31 gen.send("uks kaks kolm neli")
32 gen.send("eins zwei drei")
33 gen.send("jeden dwa")
34 gen.send("un deux trois")
35 gen.send("uno dos")
36 gen.send("uno due tre quattro")
37
38 gen.close()
39 print('\n** 6 ** Generator closed\n')
40
41 gen.send("ichi ni san")
```

Program drukuje

```
** 1 ** Defining longlines
** 2 ** Executing coroutine
** 3 ** Main starting, creating generator
** 4 ** From primed
** 5 ** Longlines advances to yield
<class 'generator'>
<class 'function'> longlines Select lines with more than n words.

Got long line: üks kaks kolm neli
Got long line: uno due tre quattro

** 6 ** Generator closed

Traceback (most recent call last):
  File "/home/werner/python/Priming.py", line 41, in <module>
    gen.send("ichi ni san")
StopIteration
```

Zilustrowaliśmy tu też zastosowanie metody **close()**, która zamyka generator: próba wysłania do niego informacji po zamknięciu powoduje zgłoszenie wyjątku.

6.4 Potoki generatorów i korutyn

Jak widzieliśmy, generatory i korutyny mogą otrzymywać i wysyłać informacje do innych generatorów/korutyn — pozwala to budować konfigurowalne potoki operacji, gdzie dane wyjściowe jednego generatora są wejściowymi dla następnego. Przypomina to strumienie w Javie z operacjami pośrednimi i terminalnymi.

Rozpatrzmy najpierw potoki korutyn. Będą one otrzymywać dane (poprzez użycie wyrażeń **yield**) z poprzedniej korutyny a następnie, po jakimś przetworzeniu, wysyłać je („pchać") do następnej (poprzez wywołanie **send**). Oczywiście, musimy zapewnić utworzenie jakiegoś początkowego strumienia danych i „wepchnąć" go do potoku, a w końcu też przerwać proces jakąś operacją (tzw. *sink,*) która już nie będzie przekazywać danych dalej.

Rozpatrzmy następujący przykład. W poniższym programie

Listing 32 ABW-PushPipe/PushPipe.py

```
#!/usr/bin/env python

import functools

  # decorator priming coroutines
def coroutine(f):
    @functools.wraps(f)
    def primed(*args,**kwargs):
        corou = f(*args,**kwargs)
        next(corou)
        return corou
    return primed

  # passes only n items
@coroutine
def limi(n, target):
    while True:
        item = (yield)
        if n > 0:
            print('lim'+str(item), end=' ')
            target.send(item)
        n -= 1

  # applies consumer, passes everything unchanged
@coroutine
def peek(consumer, target):
    while True:
        item = (yield)
        consumer(item)
        target.send(item)

  # passes only what meets predicate
@coroutine
def filt(predicate, target):
    while True:
        item = (yield)
        if predicate(item):
            print('fil'+str(item), end=' ')
            target.send(item)

  # passes everything mapped by func
@coroutine
def mapp(func, target):
    while True:
        item = (yield)
        print('map'+str(item), end=' ')
        target.send(func(item))

```

```
49    # pushes data to pipe
50  def gen(source, target):
51      for e in source: target.send(e)
52
53    # sink: appends everything to list
54  @coroutine
55  def toList(lst):
56      while True:
57          item = (yield)
58          print("sink" + str(item), end=' ')
59          lst.append(item)
60
61
62  src = (x for x in range(1, 7))      # for initial generator
63  pred = lambda n: not n%2            # for filtering
64  func = lambda x: 'Res'+str(x)       # for mapping
65  cons = lambda x: print('peek'+str(x), end=' ')  # for peek
66  lst = []                            # for toList
67
68    # either this...
69  crMapp = mapp(func, toList(lst))
70  crFilt = filt(pred, crMapp)
71  crLimi = limi(4, crFilt)
72  crPeek = peek(cons, crLimi)
73  gen(src, crPeek)                    # starting the pipe
74
75    # or this
76  #gen(src, peek(cons, limi(4, filt(pred, mapp(func, toList(lst))))))
77
78  print('\nResulting list:', lst)
```

- Funkcja **gen** iteruje po obiekcie iterowalnym source (zdefiniowanym w linii 62) i wysyła element, jeden po drugim, do target. W tym przykładzie „targetem” jest korutyna peek.[1]
- Korutyna **peek** aplikuje do elementów funkcję **consumer** (zdefiniowaną w linii 65) i „przepuszcza” je dalej, do **limi**.
- Korutyna **limi** „przepuszcza” tylko n (w przykładzie jest to 4) pierwszych elementów, ignorując pozostałe, i posyła je do **filt**.
- Korutyna **filt** pobiera predykat (zdefiniowany w linii 63) i odrzuca elementy go niespełniające; te, które go spełniają wysyłane są do **mapp**.
- Korutyna **mapp** aplikuje do elementów funkcję mapującą **func** (zdefiniowaną w linii 64) i przekazuje je do **toList**.
- Korutyna **toList** jest tu operacją końcową: wszystkie przychodzące elementy dodawane są do listy przekazanej przez argument.

Możemy z tych operacji skonstruować jeden łańcuch jak w linii 76, albo, dla czytelności, rozbić go na pojedyncze operacje, jak w liniach 69-73 (które powinny być czytane „wstecz”).

[1] Wszystkie korutyny zostały zapoczątkowane za pomocą dekoratora **coroutine**.

W obu przypadkach otrzymamy wydruk (dwie pierwsze linie to tak na prawdę jedna linia)

```
peek1 lim1 peek2 lim2 fil2 map2 sinkRes2 peek3 lim3
    peek4 lim4 fil4 map4 sinkRes4 peek5 peek6
Resulting list: ['Res2', 'Res4']
```

(jak widać, elementy 5 i 6 zostały wypchnięte do **peek**, chociaż nie są potrzebne wobec **limi**).

Rozpatrzmy teraz ten sam ciąg operacji, ale używając generatorów „wyciągających" dane z poprzednich.

Listing 33 ABX-PullPipe/PullPipe.py

```
#!/usr/bin/env python

import functools

  # passes only n items
def limi(n, source):
    for _ in range(n):
        item = next(source)
        print('lim'+str(item), end=' ')
        yield item

  # applies consumer, passes everything unchanged
def peek(consumer, source):
    for item in source:
        consumer(item)
        yield item

  # passes only what meets predicate
def filt(predicate, source):
    for item in source:
        if predicate(item):
            print('fil'+str(item), end=' ')
            yield item

  # passes everything mapped by func
def mapp(func, source):
    for item in source:
        print('map'+str(item), end=' ')
        yield func(item)

  # pushes data to pipe
def gen(source, target):
    for e in source:
        target.send(e)

  # sink: appends everything to list
def toList(source):
```

```
38      lst = []
39      for item in source:
40          print("sink" + str(item), end=' ')
41          lst.append(item)
42      return lst
43
44
45  src = (x for x in range(1, 7))      # for initial generator
46  pred = lambda n: not n%2            # for filtering
47  func = lambda x: 'Res'+str(x)       # for mapping
48  cons = lambda x: print('peek'+str(x), end=' ')  # for peek
49  lst = []                            # for toList
50
51    # either this
52  grPeek = peek(cons, src)
53  grLimi = limi(4, grPeek)
54  grFilt = filt(pred, grLimi)
55  grMapp = mapp(func, grFilt)
56  res = toList(grMapp)                # starting the pipe
57
58    # or this
59  #res = toList(mapp(func, filt(pred, limi(4, peek(cons, src)))))
60
61  print('\nResulting list:', res)
```

Tym razem zaczynamy od drugiego końca:

- Funkcja **toList** zbiera elementy do listy; wyciąga je z **mapp**.
- Generator **mapp** aplikuje funkcję **func** (zdefiniowaną w linii 47) do wszystkich elementów „wyciągniętych” ze źródła, którym jest **filt**.
- Generator **filt** pobiera predykat (zdefiniowany w linii 46) i odrzuca elementy go niespełniające; dane są pobierane z **limi**.
- Generator **limi** przepuszcza tylko `n` pierwszych elementów (w przykładzie jest to 4), ignorując wszystkie pozostałe; dane pochodzą z **peek**.
- Generator **peek** aplikuje funkcję **consumer** (zdefiniowaną w linii 48) do wszystkich elementów, ale przepuszcza wszystkie elementy niezmienione; dane są „ciągnięte” ze źródła zdefiniowanego jako generator w linii 45.

Znów możemy skonstruować jeden łańcuch operacji, jak w linii 59, albo, dal jasności, rozbić problem na pojedyncze operacje, jak w liniach 52-56 (które znów powinny być czytane od końca).

W obu przypadkach rezultat jest (dwie pierwsze linie to tak na prawdę jedna linia)

```
peek1 lim1 peek2 lim2 fil2 map2 sinkRes2 peek3 lim3
    peek4 lim4 fil4 map4 sinkRes4
Resulting list: ['Res2', 'Res4']
```

(jak widzimy, tym razem elementy 5 i 6 *nie* zostały wyciągnięte przez **peek**, bo **limi** zażądał tylko czterech elementów).

6.5 Parę narzędzi z modułu itertools

Moduł *itertools* zawiera szereg narzędzi do operowania na obiektach iterowalnych. Jeden ważny przykład już poznaliśmy, mianowicie funkcję **accumulate** (patrz rozdz. 5.4.4, str. 82).

Innymi użytecznymi funkcjami z tego modułu są funkcje **dropwhile** i **takewhile**. Obie pobierają predykat i obiekt iterowalny a zwracają też obiekt iterowalny. Pierwsza z nich *ignoruje* elementy z początku przekazanego obiektu, dopóki spełniają one przekazany predykat, podczas gdy druga zwraca *tylko* te elementy:

```
>>> from itertools import dropwhile, takewhile
>>> ls = [1, 2, 3, 4, 5, 6]
>>>
>>> drop = dropwhile(lambda x: x < 4, ls)
>>> type(drop)
<class 'itertools.dropwhile'>
>>> drop
<itertools.dropwhile object at 0x7f8d1050ddc0>
>>> list(drop)
[4, 5, 6]
>>>
>>> take = takewhile(lambda x: x < 5, ls)
>>> type(take)
<class 'itertools.takewhile'>
>>> take
<itertools.takewhile object at 0x7f8d1050e080>
>>> list(take)
[1, 2, 3, 4]
```

Zauważmy, że funkcja **takewhile** może być użyta dla potencjalnie nieskończonych obiektów iterowalnych gdy musimy zapewnić zakończenie iteracji po skończonej liczbie kroków.

Trochę związana z poprzednimi jest funkcja **islice**. Pozwala ona wybrać z iteracji tylko niektóre zwracane elementy. Składnia jest podobna do składni **range** (i do wycinków)

```
itertools.islice(iterable, stop)
itertools.islice(iterable, start, stop[, step])
```

Funkcja zwraca obiekt iterowalny, który będzie zwracał tylko wartości z wejściowego obiektu wybrane poprzez wartości start, step i step.

Wywołanie z jednym argumentem (nie licząc iterable) jest równoważne

```
itertools.islice(iterable, 0, stop, 1)])
```

czyli wybranych będzie tylko pierwszych stop elementów (moglibyśmy powiedzieć, że są to elementy o indeksach od 0 włącznie do stop wyłącznie, ale oczywiście wejściowy obiekt iterowalny nie musi być indeksowalny).

Z dwoma argumentami (nie licząc iterable), wywołanie jest równoważne

```
itertools.islice(iterable, start, stop)])
```

czyli pierwsze start elementów będzie pominiętych, a wybranych zostanie tylko stop−start następnych; jeśli stop jest None, to oznacza *aż do wyczerpania wejściowego obiektu iterowalnego* (moglibyśmy zatem powiedzieć, że wybrane zostaną elementy o indeksach *od* start *włącznie do* stop *wyłącznie*).

W obu poprzednich przypadkach step przyjmowało domyślną wartość 1, czyli wybierane były wszystkie elementy z danego przedziału. Tak jak dla **range** (i wycinków), ustalając wartość step większą od 1, powiedzmy n, możemy wybrać z danego przedziały tylko co n-ty element; trzeba wtedy podać trzy argumenty, choć dwa pierwsze mogą być None (dla start znaczy to 0, dla stop – *do końca*).

Na przykład:

Listing 34 ABM-Islice/Islice.py

```
#!/usr/bin/env python

from sys import maxsize
from itertools import islice

iterable = '0123456789'

for e in islice(iterable, 5):
    print(e, end=' ')   # 0 1 2 3 4
print()

for e in islice(iterable, 2, 8):
    print(e, end=' ')   # 2 3 4 5 6 7
print()

for e in islice(iterable, None, None, 3):
    print(e, end=' ')   # 0 3 6 9
print()


  # 'almost infinite' sequence of primes
iterable = (k for k in range(2, maxsize)
            if all(k%m != 0 for m in range(2, int(k**0.5)+1)))
  # first 10 primes
for e in islice(iterable, 10):
    print(e, end=' ')   # 2 3 5 7 11 13 17 19 23 29
print()

  # we have to recreate the iterable, because
  # generators cannot be 'rewound'!
iterable = (k for k in range(2, maxsize)
            if all(k%m != 0 for m in range(2, int(k**0.5)+1)))
 # 100-th to 105-th primes
for e in islice(iterable, 100, 106):
    print(e, end=' ')   # 547 557 563 569 571 577
print()

```

```
38 iterable = (k for k in range(2, maxsize)
39              if all(k%m != 0 for m in range(2, int(k**0.5)+1)))
40  # every third prime number from 1000-th to 1021-th
41 for e in islice(iterable, 1000, 1022, 3):
42     print(e, end=' ')    # 7927 7949 7993 8017 8059 8087 8101 8123
43 print()
```

Funkcja **count** jest inną pożyteczną funkcją. Jest nieco podobna do **range**, ale daje zawsze ciąg nieskończony od start co step, przy czym start i step nie muszą tu być całkowite (bo nie odpowiadają "indeksom")

```
itertools.count(start=0, step=1)
```

Na przykład

```
>>> from itertools import count, takewhile, islice
>>>
>>> cnt = count()
>>> type(cnt)
<class 'itertools.count'>
>>> list( takewhile(lambda x: x < 10, cnt) )
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> list( islice(count(1.5, 2.5), 100, 106) )
[251.5, 254.0, 256.5, 259.0, 261.5, 264.0]
```

Trzeba pamiętać, że **count** daje zawsze nieskończony ciąg wartości, więc musimy zadbać o wybranie z niego skończonego podciągu, na przykład, jak w powyższym fragmencie, poprzez zastosowanie **takewhile** lub **islice**).

Jako ostatni przykład funkcji z modułu *itertools* omówmy krótko funkcję **groupby**

```
itertools.groupby(iterable, key=None)
```

Pobiera ona obiekt iterowalny i funkcję klucza, podobną do tej jaką stosujemy w funkcji **sorted**/**sort**. Funkcja klucza powinna dla elementów wejściowego obiektu iterowalnego dawać wartości, których chcemy użyć jako kluczy do grupowania elementów (domyślnie funkcją tą jest identyczność). Elementy odpowiadające tej samej wartości klucza, które wobec tego powinny trafić do tej samej grupy, muszą być położone kolejno obok siebie. Dlatego bardzo często wejściowy obiekt iterowalny trzeba najpierw posortować za pomocą tej samej funkcji klucza.
Funkcja zwraca obiekt iterowalny, iteracja którego daje pary z kluczem jako pierwszym elementem; drugim elementem jest obiekt iterowalny dający wartości należące do danej grupy.
Na przykład

Listing 35 ABN-Groups/Groups.py

```
1 #!/usr/bin/env python
2
```

```
3  from itertools import groupby
4
5  lst = ['frog', 'goat', 'France', 'Greg', 'floor']
6
7    # key function: by first letter case insensitively
8  keyfun = lambda s: s.lower()[0]
9
10   # sorting
11 lst.sort(key=keyfun)
12
13 print(lst, end='\n\n') # ['frog', 'France', 'floor', 'goat',
   ↪  'Greg']
14
15 groups = groupby(lst, key=keyfun)
16 print(type(groups), end='\n\n')  # <class 'itertools.groupby'>
17
18 for k, v in groups:
19     print(type(v))
20     print(f"Group '{k}' -> {list(v)}\n")
```

Program drukuje

```
['frog', 'France', 'floor', 'goat', 'Greg']

<class 'itertools.groupby'>

<class 'itertools._grouper'>
Group 'f' -> ['frog', 'France', 'floor']

<class 'itertools._grouper'>
Group 'g' -> ['goat', 'Greg']
```

Rozdział 7

Kolekcje

Kolekcje, takie jak listy, krotki czy słowniki, już spotykaliśmy: tutaj zbierzemy podstawowe informacje o różnych kolekcjach, zarówno tych, o których już mówiliśmy jak i kilku innych.

Pyton wyróżnia się tym, że wiele typów kolekcyjnych jest wbudowanych w język, również takie, które w innych językach są raczej częścią ich bibliotek standardowych (przykładem mogą być listy, słowniki, zbiory, krotki). Jest jednak kilka typów kolekcji, które i w Pytonie są częścią biblioteki standardowej.

Kolekcja (*collection*) w Pytonie musi spełniać wymagania protokołu **Collection**, a zatem

- implementować protokół **Container** czyli odpowiadać na operatory**in** i **not in** (tak będzie, jeśli ma metodę **__contains__**);
- być iterowalna — implementować protokół **Iterable** — czyli reagować na funkcję **iter** (tak będzie, jeśli ma metodę **__iter__**);
- implementować protokół **Sized**, czyli reagować na funkcję **len** (tak będzie, jeśli ma metodę **__len__**).

7.1 Kolekcje sekwencyjne

Kolekcja sekwencyjna, prócz tego, że musi być w ogóle kolekcją, powinna dodatkowo być **Reversible**, czyli reagować na funkcje **index**, **count**, **reversed** (tak będzie, jeśli ma metody **__getitem__** i **__reversed__**).[1]

Kolekcje sekwencyjne mogą być modyfikowalne (jak **list**) lub nie (jak **tuple**). Jeśli mają być modyfikowalne (ang. *mutable*), to prócz tego, że muszą być kolekcjami sekwencyjnymi, powinny wypełniać protokół **MutableSequence**, a zatem reagować na funkcje **insert**, **append**, **extend**, **pop**, **remove** (tak będzie jeśli dodatkowo mają metody **__setitem__**, **__delitem__** i **__add__**).

7.1.1 Operacje niemodyfikujące

Niemodyfikujące operacje na kolekcjach sekwencyjnych mogą być stosowane zarówno dla kolekcji niemodyfikowalnych jak i modyfikowalnych. Przedstawiamy je tu na przykładzie list, krotek i napisów.

in, **not in**

```
>>> lst = [2, 3, 4, 5, 6, 7, 8]
>>> k = 7
>>> k in lst
True
>>> k+2 not in lst
True
>>> k+2 in lst
False
```

[1] W przeciwieństwie do seq[::-1], funkcja **reversed(seq)** (która wywołuje **__reversed__**) nie tworzy żadnych kopii — zwraca tylko iterator przebiegający sekwencję w tył.

Zwykle operatory **in** i **not in** sprawdzają czy dana wartość jest czy nie jest równa wartości jednego z elementów kolekcji. Dla typu **str** (a również **bytes** i **bytearray**) wartością szukaną może jednak być podciąg elementów listy, a niekoniecznie pojedynczy element:

```
>>> s, s1, s2 = 'abcdef', 'cde', 'efg'
>>> s1 in s
True
>>> s2 in s
False
```

Konkatenacja

Konkatenacja (złożenie) dwóch sekwencji zwraca nową sekwencję. Sekwencje powinny być tego samego typu:

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> 'abc' + 'def'
'abcdef'
>>> (1, 2) + [3, 4]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple (not "list") to tuple
```

Konkatenacja nie jest dostępna dla obiektów typu **range**.
Jeśli chcemy dodać tę samą kolekcję do siebie kilka razy, można zastosować „mnożenie”:

```
>>> [1, 2]*3
[1, 2, 1, 2, 1, 2]
>>> 4*'abc'
'abcabcabcabc'
```

Trzeba jednak zwracać uwagę na to czy „replikowany” obiekt jest czy nie jest modyfikowalny, bowiem to co jest „replikowane”, to *referencje*. Jeśli odnoszą się do obiektów niemodyfikowalnych, nie ma problemu. Ale jeśli te obiekty są modyfikowalne, to ich zmiana dotyczy również obiektów wskazywanych przez pozostałe kopie „zreplikowanej” referencji, bo są to po prostu te same obiekty:

```
>>> ls = [1, [2, 3]]
>>> ls
[1, [2, 3]]
>>> ls2 = 2*ls
>>> ls2
[1, [2, 3], 1, [2, 3]]
>>> ls[1][1] = 9
>>> ls2
[1, [2, 9], 1, [2, 9]]
>>> id(ls[1]), id(ls2[1]), id(ls2[3])
(140148710923968, 140148710923968, 140148710923968)
```

Indeksowanie i wycinki

Ten temat omówiliśmy już wcześniej — zob. rozdz. 4.5, str. 43. Zauważmy tutaj tylko,

że dla obiektów **range** też można stosować indeksy i wycinki, choć może to czasem być mylące i trzeba zwracać uwagę na to które argumenty odnoszą się do wartości, a które do indeksów. Na przykład

```
>>> r = range(10, 29, 3)
>>> r
range(10, 29, 3)
>>> type(r)
<class 'range'>
>>> list(r)
[10, 13, 16, 19, 22, 25, 28]
>>>
>>> q = r[1:4:2]
>>> q
range(13, 22, 6)
>>> type(q)
<class 'range'>
>>> list(q)
[13, 19]
>>>
>>> s = r[::-1]
>>> s
range(28, 7, -3)
>>> type(s)
<class 'range'>
>>> list(s)
[28, 25, 22, 19, 16, 13, 10]
```

Zaczynamy od r, które reprezentuje ciąg arytmetyczny od *wartości* 10 do, ale nie włącznie, wartości 29 i przy różnicy ciągu 3; odpowiada to sekwencji

$$[10, 13, 16, 19, 22, 25, 28]$$

Następnie bierzemy wycinek [1:4:2] — te liczby odnoszą się do *indeksów,* a nie wartości. Dostajemy zatem ciąg liczb od elementu o indeksie 1, który ma wartość 13, do, ale bez, elementu o indeksie 4 o wartości 22, z krokiem w indeksach 2, czyli bierzemy co drugi element. Rezultatem jest zatem ciąg [13, 19], który odpowiada obiektowi range(13, 22, 6).[1]

len

Wszystkie kolekcje muszą mieć dobrze określoną liczbę elementów dostępną za pomocą funkcji **len**.

```
>>> ls = [1, 2, (3, 4, (5, 6, 7)), [(8, 9), (10,)]]
>>>
>>> len(ls)
4
>>> len(ls[2])
3
>>> len(ls[2][2])
3
>>> len(ls[3][1])
1
```

[1] Oczywiście, **range(13,20,6)** czy **range(13,25,6)** reprezentowałyby dokładnie tę samą sekwencję liczb.

min, max
Znajdowanie najmniejszego lub największego elementu jest możliwe, jeśli elementy te są porównywalne; na przykład można porównywać krotki z krotkami, ale nie liczby z krotkami. Do porównywania będzie używany operator $<$, więc musi on być dobrze zdefiniowany dla dowolnych dwóch elementów kolekcji:

```
>>> ls = [(1,), (2,3)]
>>> min(ls)
(1,)
>>> ls = [1, (2,3)]
>>> min(ls)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'tuple' and 'int'
```

index(x[, i[, j]])
zwraca indeks pierwszego wystąpienia wartości x w sekwencji (lub jej podciągu od elementu o indeksie i do, ale bez, elementu o indeksie j). Jeśli taki element nie istnieje, zgłaszany jest wyjątek **ValueError**. Nie wszystkie sekwencje dopuszczają dodatkowe argumenty (i i j):

```
>>> r = range(10, 31, 3)
>>> list(r)
[10, 13, 16, 19, 22, 25, 28]
>>> r.index(19)
3
>>> r.index(19, 1, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: range.index() takes exactly one argument (3 given)
```

count(x)
zwraca liczbę wystąpień x w sekwencji. Działa również dla obiektów **range**:

```
>>> r = range(10, 31, 3)
>>> r.count(19)
1
>>> r.count(20)
0
```

7.1.2 Operacje modyfikujące

Kolekcje modyfikowalne wspierają więcej operacji niż niemodyfikowalne — te, które modyfikują kolekcję, a zatem nie mogą zostać użyte dla kolekcji niemodyfikowalnych.

Modyfikacja elementów

```
>>> ls = [0, 1, 2, 3, 4, 5, 6, 7]
>>> ls[2] = 'two'
>>> ls
[0, 1, 'two', 3, 4, 5, 6, 7]
```

Można też przypisywać do wycinków kolekcji, ale wtedy prawa strona jest traktowana jako obiekt iterowalny i jego elementy są przypisywane do elementów wycinka jeden po drugim:

```
>>> ls = [0, 1, 2, 3]
>>> ls[1:3] = 'one two'
>>> ls
[0, 'o', 'n', 'e', ' ', 't', 'w', 'o', 3]
```

Jeśli to o co nam chodzi to napis jako całość, możemy go uczynić pojedynczym elementem krotki (zauważ przecinek):

```
>>> ls = [0, 1, 2, 3]
>>> ls[1:3] = 'one two',  # zauważ przecinek
>>> ls
[0, 'one two', 3]
```

Jeśli wycinek ma krok (różnicę) różny od jedynki, prawa strona musi być obiektem iterowalnym o dokładnie takiej liczbie elementów jak liczba elementów wycinka:

```
>>> ls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ls[::3] = 'zero', 'three', 'six', 'nine'
>>> ls
['zero', 1, 2, 'three', 4, 5, 'six', 7, 8, 'nine']
>>>
>>> ls = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> ls[::3] = 'zero', 'three', 'six'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of
size 3 to extended slice of size 4
```

Dodawanie elementów

Nowe elementy mogą być dodawane do sekwencji za pomocą metody **insert**, która pobiera indeks określający miejsce wstawienia i wartość do wstawienia. Jeśli wymiar sekwencji wynosi L, to indeksy większe od L są równoważne L (nie ma wyjątku!). Indeksy mogą być ujemne ze zwykłą interpretacją: -1 odpowiada ostatniemu elementowi, a -L pierwszemu. Znów, indeksy mniejsze od -L są równoważne -L (nie ma wyjątku):

```
>>> l1 = ['b', 'd']
>>> l2 = ['b', 'd']
>>> l3 = ['b', 'd']
>>> l4 = ['b', 'd']
>>> l5 = ['b', 'd']
>>> l1.insert(2, 'e')
>>> l2.insert(3, 'e')
>>> l1, l2
(['b', 'd', 'e'], ['b', 'd', 'e'])
>>> l3.insert(-2, 'a')
>>> l4.insert(-3, 'a')
>>> l3, l4
(['a', 'b', 'd'], ['a', 'b', 'd'])
```

```
>>> l5.insert(-1, 'c')
>>> l5
['b', 'c', 'd']
```

Metoda **append** dodaje pojedynczy element na koniec sekwencji (nawet jeśli jest to obiekt iterowalny):

```
>>> ls = [1, 'a']
>>> ls.append(['X', 'Y'])
>>> ls
[1, 'a', ['X', 'Y']]
```

Natomiast metoda **extend** pobiera obiekt iterowalny i dodaje jego elementy pojedynczo:

```
>>> ls = [1, 'a']
>>> ls.extend('one')
>>> ls
[1, 'a', 'o', 'n', 'e']
>>>
>>> ls = [1, 'a']
>>> ls.extend(('one',))  # zauważ przecinek
>>> ls
[1, 'a', 'one']
```

Zamiast wywoływania **extend** można też użyć operatora +=:

```
>>> ls = [1, 'a']
>>> ls += 'one'
>>> ls
[1, 'a', 'o', 'n', 'e']
>>>
>>> ls = [1, 'a']
>>> ls += 'one',          # zauważ przecinek
>>> ls
[1, 'a', 'one']
```

Usuwanie elementów

Za pomocą instrukcji **del** można usunąć wybrane elementy (lub całą kolekcję; **del** może usunąć dowolny obiekt). Możemy usuwać pojedyncze elementy lub wycinki:

```
>>> ls = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> del ls[2]
>>> ls
['a', 'b', 'd', 'e', 'f', 'g', 'h']
>>> del ls[1:3]
>>> ls
['a', 'e', 'f', 'g', 'h']
>>> del ls[::2]
>>> ls
['e', 'g']
```

Do usuwania elementów sekwencji można też użyć metody **remove**, która pobiera wartość i usuwa pierwszy element o takiej wartości, lub zgłasza wyjątek **ValueError** jeśli takiego nie znajdzie:

```
>>> ls = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> ls.remove('d')
>>> ls.remove('g')
>>> ls
['a', 'b', 'c', 'e', 'f', 'h']
>>>
>>> ls.remove('x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

W końcu można też użyć metody **pop**. Pobiera ona indeks (który ma wartość domyślną -1, co odpowiada *ostatniemu* elementowi) i nie tylko usuwa element o tym indeksie, ale również zwraca jego wartość:

```
>>> ls = ['a', 'b', 'c', 'd', 'e']
>>> x = ls.pop()
>>> x, ls
('e', ['a', 'b', 'c', 'd'])
>>>
>>> y = ls.pop(1)
>>> y, ls
('b', ['a', 'c', 'd'])
>>>
>>> z = ls.pop(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: pop index out of range
```

Kopiowanie

Metoda **copy** zwraca *płytką* kopię kolekcji:

```
>>> ls = [1, [2, 3], 'a']
>>> lc = ls.copy()
>>> ls[1][1] = 9
>>> lc
[1, [2, 9], 'a']  # lc też zmienione
```

Możemy też zastąpić kolekcję przez n płytkich kopii jej samej używając operatora *= (płytkich, bo kopiowane są referencje a nie wartości):

```
>>> ls = [1, [2, 3]]
>>> ls *= 3
>>> ls
[1, [2, 3], 1, [2, 3], 1, [2, 3]]
>>> ls[1][1] = 9
>>> ls
[1, [2, 9], 1, [2, 9], 1, [2, 9]] # wszystkie kopie zmodyfikowane
```

Odwracanie i czyszczenie

W końcu metoda **clear clear (funkcja)** „czyści” kolekcję — staje się ona pusta, ale w dalszym ciągu istnieje. Metoda **reverse** odwraca kolejność elementów *in situ* i zwraca **None**:

```
>>> ls = [1, [2, 3], 'a']
>>> x = ls.reverse()        # odwracanie in situ
>>> print(x)
None
>>> ls
['a', [2, 3], 1]
>>> ls.clear()
>>> ls
[]
```

7.1.3 Obiekty *bytes*

Obiekty typu **bytes** reprezentują *niemodyfikowalne* sekwencje bajtów. Implementowane są za pomocą sekwencji jednobajtowych liczb całkowitych bez znaku z przedziału $[0, 255]$. Literały tego typu zapisujemy podobnie do napisów, tyle że z literą 'b' (lub 'B') przed cudzysłowem otwierającym (może być pojedynczy apostrof albo podwójny znak cudzysłowu; wersja z trzema cudzysłowami jest też możliwa). Prócz 'b' można też dodać prefiks 'r' (lub 'R') aby uzyskać interpretację „surową", w której znak odwróconego ukośnika (\) nie jest traktowany specjalnie i oznacza sam siebie.

Pisząc literał tej klasy można użyć po prostu pojedynczego znaku dla bajtów odpowiadających znakom ASCII. Dla innych wartości bajtów, tych większych od 126 ($= 7E_{16}$) lub odpowiadających znakom kontrolnym, czyli mniejszym od 32 ($= 20_{16}$), należy użyć zapisu \xhh, gdzie hh oznacza dokładnie dwie cyfry liczby w zapisie szesnastkowym: jedna cyfra na każdy półbajt (cztery bity, ang. *nibble*).[1] Taka jest też forma w jakiej wartości typu **bytes** są wyświetlane. Na przykład, w `b'B\xc4\x85k'` 'B' oznacza 66 (bo 66 jest kodem Unicode litery 'B'), a 'k' na końcu to 107 (kod Unicode tej litery). Dwa środkowe bajty odpowiadają liczbom 196 i 133 — wzięte razem odpowiadają one dwubajtowemu kodowi polskiej litery 'ą' w UTF-8.

```
>>> bąk = b'B\xc4\x85k'
>>> bąk
b'B\xc4\x85k'
```

Jeśli obiekty **bytes** odpowiadają napisom, a nie danym binarnym, ma sens użycie metod konwertujących takie obiekty w napisy i odwrotnie. Na przykład, aby skonwertować obiekt **bytes** na napis, można użyć konstruktora klasy **str** określając przy tym kodowanie[2]

```
>>> bąk = b'B\xc4\x85k'  # to jest 'bąk' w UTF-8
>>> type(bąk)
<class 'bytes'>
>>> st = str(bąk, 'utf_8')
>>> type(st)
<class 'str'>
>>> st
'Bąk'
```

i w podobny sposób dokonywać konwersji odwrotnej

[1] Notacje \uhhhh i \Uhhhhhhhh dla 16- i 32-bitowych kodów Unicode mogą być używane tylko dla literałów typu **str**.

[2] Wiele nazw kodowań zawiera podkreślnik (_), jak 'utf_8', 'koi8_u' itd. Czasem forma z myślnikiem jest również akceptowalna (na przykład 'utf-8') jako alias, ale używanie aliasów nie jest wskazane, bo może prowadzić do niewielkiej straty efektywności.

```
>>> bt = bytes('Bąk', 'utf_8')
>>> type(bt)
<class 'bytes'>
>>> bt
b'B\xc4\x85k'
```

Obiekty **bytes** są tak naprawdę sekwencjami liczb (bajtów); możemy to zobaczyć przekazując je do konstruktora klasy **list**

```
>>> bt = bytes('Żółć', 'utf_8')
>>> list(bt)
[197, 187, 195, 179, 197, 130, 196, 135]
```

lub tworząc obiekt **bytes** z obiektu iterowalnego zwracającego liczby, na przykład listy

```
>>> bt = bytes([197, 187, 195, 179, 197, 130, 196, 135])
>>> str(bt, 'utf_8')
'Żółć'
```

Funkcja **fromhex** może być użyta do utworzenia obiektu **bytes** z napisu zawierającego liczby w zapisie szesnastkowym: dwie cyfry szesnastkowe odpowiadają jednemu bajtowi. Napis może zawierać białe znaki dla poprawienia czytelności — będą one zignorowane:

```
>>> bt = bytes.fromhex('42 c485 6b')
>>> str(bt, 'utf_8')
'Bąk'
```

Operacja odwrotna, od obiektu **bytes** do napisu zawierającego liczby w zapisie szesnastkowym, jest możliwa za pomocą metody **hex** (można przesłać separator jaki ma zostać użyty między liczbami):

```
>>> bt = bytes('Żółć', 'utf_8')
>>> type(bt)
<class 'bytes'>
>>> bt.hex()
'c5bbc3b3c582c487'
>>> bt.hex('-')
'c5-bb-c3-b3-c5-82-c4-87'
```

Zwykle obiekty **bytes** zawierają nie napisy, a dane binarne. Źródłem tych danych zazwyczaj są pliki, zatem musimy je z plików wczytywać i do plików zapisywać (zob. rozdz. **??**, str. **??**):

```
>>> bt = bytes('Żółć', 'utf_8')
>>> with open('bytes.dat', 'wb') as f:
...     f.write(bt)
...
8
>>> with open('bytes.dat', 'rb') as f:
...     bb = f.read()
...
>>> str(bb, 'utf_8')
'Żółć'
```

Pliki, oczywiście, muszą być otwierane w trybie binarnym. Funkcja **write** zwraca liczbę zapisanych bajtów. Czasami logika programu wymaga czytania/pisania bajtów pojedynczo, bajt po bajcie

```
1   >>> bt = bytes('Żółć', 'utf_8')
2   >>> with open('bytes.dat', 'wb') as f:
3   ...     for i in range(len(bt)):
4   ...         f.write(bt[i:i+1])
5   ...
6   >>> with open('bytes.dat', 'rb') as f:
7   ...     bt = bytes()
8   ...     while (b := f.read(1)):
9   ...         bt += b
10  ...
11  >>> str(bt4, 'utf_8')
12  'Żółć'
```

Zauważmy, że w linii 4 użyliśmy jednoelementowego wycinka, bo jego typ to **bytes** — po prostu `bt[i]` byłoby typu **int**.
W linii 8 użyliśmy funkcji **read** z argumentem 1 — mówi to, że wczytany ma być tylko jeden bajt. Zastosowaliśmy też wyrażenie przypisania (*walrus*): to działa, bo kiedy osiągnięty został koniec pliku, **read** zwraca napis pusty, co jest interpretowane jako **False**.
Linia 9 jest oczywiście bardzo nieefektywna: obiekty **bytes** są niemodyfikowalne, więc w każdej iteracji `bt += b` tworzy nowy obiekt.

Obiekty **bytes** są niemodyfikowalnymi sekwencjami, zatem możemy dla nich stosować niemodyfikujące metody opisane w rozdz. 7.1.1, str. 118.
Prócz tego są też bardzo podobne do obiektów typu **str**, a zatem również metody tej klasy (opisane w rozdz. 8, str. 159) mogą być stosowane dla obiektów **bytes**. Pamiętać trzeba jednak przy tym, że analogiczne metody klasy **str** mające argumenty typu **str**, dla obiektów **bytes** wymagają argumentów typu bajtowego, a więc **bytes** lub **bytearray**:

```
>>> bt = b'abcdef'
>>> a = bt.replace(b'c', b'C')
>>> type(a), a
(<class 'bytes'>, b'abCdef')
>>> b = bt.replace('c', 'C')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: a bytes-like object is required, not 'str'
```

7.1.4 Obiekty *bytearray*

Obiekty typu **bytearray** reprezentują, jak **bytes**, sekwencje bajtów, a więc liczb całkowitych w zakresie $[0, 255]$, ale, w odróżnieniu od **bytes**, są modyfikowalne.

Nie ma literałów tego typu. Natomiast metoda **__str__** zwraca dla obiektów **bytearray** napis w formacie jak dla obiektów **bytes**, czyli liczby z zakresu $[32, 126]$ są wypisywane jako odpowiadające im znaki, a pozostałe jako `\xhh`, gdzie `hh` to dokładnie dwie cyfry szesnastkowe.

```
>>> bytearray([30, 31, 32, 33, 125, 126, 127, 128])
bytearray(b'\x1e\x1f !}~\x7f\x80')
```

Skonstruować obiekty **bytearray** można na kilka sposobów:

- Nie przekazując do konstruktora żadnych argumentów — tworzona jest wtedy tablica pusta:

```
>>> bytearray()
bytearray(b'')
```

- Przekazując do konstruktora liczbę całkowitą — tworzona jest wtedy tablica o podanym rozmiarze wypełniona zerami

```
>>> bytearray(8)
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
```

- Przekazując do konstruktora obiekt iterowalny dostarczający liczby całkowite z zakresu $[0, 255]$ — tworzona jest wtedy tablica zawierająca podane bajty:

```
>>> bytearray(x for x in range(65, 91))
bytearray(b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

- Przekazując do konstruktora napis oraz, jako drugi argument, kodowanie tego napisu — napis jest wtedy konwertowany na sekwencję bajtów odpowiednią dla danego kodowania:

```
>>> bytearray("ABC Żółć DEF", 'utf_8')
bytearray(b'ABC \xc5\xbb\xc3\xb3\xc5\x82\xc4\x87 DEF')
```

- Przekazując do konstruktora obiekt posiadający bufor binarny, taki jak na przykład obiekt **bytes** — bufor jest wtedy kopiowany:

```
>>> bytearray(b'abc')
bytearray(b'abc')
```

- Wywołując statyczną metodę klasy **bytearray** **fromhex** i przekazując do niej napis zawierający bajty w postaci `hh` — dwóch cyfr szesnastkowych (można w tym napisie umieszczać, dla czytelności, dowolne białe znaki; będą one zignorowane):

```
>>> bytearray.fromhex('CAFE BABE')
bytearray(b'\xca\xfe\xba\xbe')
```

Obiekty **bytearray** są modyfikowalnymi sekwencjami, zatem możemy dla nich stosować metody opisane zarówno w rozdz. 7.1.1, str. 118, jak i w rozdz. 7.1.2, str. 121
Podobnie jak dla klasy **bytes**, również obiekty klasy **bytearray** mają metody analogiczne do tych z klasy **str** (patrz rozdz. 8, str. 159), ale tam gdzie argumentami tych metod dla **str** są napisy, tu muszą to być obiekty bajtowe, czyli typu albo **bytes** albo **bytearray**;

```
>>> ba = bytearray(b'abcdef')
>>> ba.endswith(b'def')
True
>>> ba.endswith('def')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: endswith first arg must be bytes or a tuple of bytes, not str
```

7.2 Obiekty haszowalne

Obiekty w Pytonie mogą być haszowalne lub nie. Jest to ważne rozróżnienie, jeśli chcemy umieszczać je jako elementy zbiorów (**set**), czy jako klucze w słownikach.

W Javie słowniki (zwane tam mapami) jak i zbiory[1] występują w dwóch odmianach: implementowane za pomocą drzew czerwono-czarnych – **TreeMap**, **TreeSet** – oraz za pomocą tablic asocjacyjnych – **HashMap**, **HashSet**. Jednak w standardzie Pythona słowniki i zbiory są oparte tylko na tablicach asocjacyjnych. Jest to implementacja nieco szybsza — $\mathcal{O}(1)$ vs. $\mathcal{O}(\log(n))$ dla drzew — i nie wymaga, aby obiekty były porównywalne. Z drugiej strony, jak pamiętamy z Javy, implementacja z haszowaniem wymaga funkcji haszującej, która dla każdego obiektu oblicza jego **hash**: liczbę całkowitą, na podstawie której wyliczany jest indeks kubełka gdzie dany obiekt będzie umieszczony (albo wyszukiwany).[2] Aby rozwiązać sytuacje konfliktowe, gdy dla więcej niż jednego obiektu wypadł ten sam indeks, potrzebna jest też funkcja (bi-predykat), która będzie umiała odpowiedzieć na pytanie czy dwa obiekty są, czy nie są, równe.

Aby to wszystko działało, hasz obiektu musi być stały. Normalnie jest on wyliczany na podstawie wartości związanych z obiektem — znaczy to, że sam obiekt powinien być niemodyfikowalny. I rzeczywiście, obiekty typów niemodyfikowalnych, jak liczby czy napisy, mają dobrze zdefiniowaną metodę **__hash__** (która jest wywoływana przez wbudowaną funkcję **hash**):

```
>>> hash(17), (17).__hash__()
(17, 17)
>>> hash('Alice'), 'Alice'.__hash__()
(-5185723981070768863, -5185723981070768863)
```

Liczby i napisy reprezentują pojedyncze wartości, od których zależy ich hash. A co z kolekcjami, reprezentującymi agregaty wartości? Tylko niezmienne kolekcje (jak krotki) mogą być haszowalne — a i to tylko wtedy, gdy nie zawierają elementów modyfikowalnych, a więc niehaszowalnych:

```
>>> tp = 1, 'abc', (2, 'de')  # wszystkie elementy haszowalne
>>> tp
(1, 'abc', (2, 'de'))
>>> hash(tp)
-1823045235319932986
>>>
>>> tq = 1, 'abc', [2, 'de']  # listy są modyfikowalne
>>> tq
(1, 'abc', [2, 'de'])
>>> hash(tq)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Ogólnie, obiekty w Pythonie są haszowalne, jeśli posiadają hasze (liczy całkowite obliczane za pomocą metody **__hash__**), które, raz obliczone, nigdy się nie zmieniają. Muszą też mieć metodę **__eq__** taką, że

[1] Implementacja zbiorów zwykle opiera się na słownikach (mapach).
[2] https://en.wikipedia.org/wiki/Hash_table

jeśli dwa obiekty są równe według __eq__, to ich hasze muszą być identyczne.

Ponieważ hasze są obliczane na podstawie wartości, modyfikowalne kolekcje[1] nie mogą zapewnić ich niezmienności. To samo dotyczy kolekcji niemodyfikowalnych,[2] jeśli zawierają elementy modyfikowalne.

Zauważmy, że obiekty typów zdefiniowanych przez użytkownika *są* domyślnie haszowalne, gdyż dziedziczą z klasy **object** implementacje metod __hash__ i __eq__. Jednak działanie tych domyślnych implementacji oparte jest wyłącznie na identyfikatorach id obiektów (czyli efektywnie ich adresach). Jeśli zatem potrzebujemy czegoś bardziej znaczącego, metody __hash__ i __eq__ musimy zaimplementować sami.

7.3 Zbiory

Zbiory reprezentują *nieuprządkowane* kolekcje *różnych*[3] nhaszowalnych obiektów. Implementacja zbiorów opiera się na haszach obiektów, a zatem one muszą być haszowalne i w konsekwencji niemodyfikowalne.

Jako kolekcje, zbiory wspierają operatory **in** i **not in**, jak również funkcję **len**. Są też iterowalne, chociaż kolejność elementów przy iteracji w zasadzie jest nieprzewidywalna. Jednakże, *nie* są sekwencyjne, a zatem nie można dla nich stosować indeksowania ani wycinków.
Są w Pythonie dwa wbudowane typy reprezentujące zbiory: **set** oraz **frozenset**. W obu przypadkach elementy muszą być haszowalne (niemodyfikowalne), ale same obiekty typu **set** *są* modyfikowalne — można do nich dodawać elementy i je usuwać. Tak więc obiekty typu **set** nie są haszowalne i nie mogą być elementami innych zbiorów:

```
>>> st = {'1', 2, '3', (4, '5')}
>>> id(st)
139913779179840
>>> st.add(6)
>>> id(st)
139913779179840  # ten sam obiekt (zmodyfikowany)
>>> st
{2, 6, '3', '1', (4, '5')}  # kolejność nieprzewidywalna
>>> newset = {st, '7', 8}   # zbiór jako element zbioru?
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

Obiekty typu **frozenset** są natomiast niemodyfikowalne i haszowalne. Po utworzeniu, żadne elementy nie mogą być do nich dodane ani z nich usunięte. Tak więc mogą one być elementami innych zbiorów czy kluczami słowników:

```
>>> st = frozenset(['1', 2, '3', (4, '5')])
>>> newset = {st, '7', 8}
>>> newset
{8, '7', frozenset({'3', '1', 2, (4, '5')})}
```

[1] Takie jak na przykład **list**, **dict**, **set**, **bytearray**.
[2] Jak **int**, **float**, **decimal**, **complex**, **bool**, **str**, **tuple**, **range**, **frozenset**, **bytes**.
[3] Dla dowolnych dwóch elementów zbioru, a i b, wartością wyrażenia a == b musi być **False**.

Niepuste zbiory mogą być utworzone poprzez przekazanie do konstruktora obiektu iterowalnego — ewentualne powtórzenia zostaną automatycznie zignorowane. Obiekty typu **set** (ale nie **frozenset**) mogą być też utworzone za pomocą literału: ujętej w nawiasy klamrowe listy oddzielonych przecinkami elementów.[1] Oczywiście w obu przypadkach wszystkie elementy, które mają się znaleźć w zbiorze muszą być haszowalne.

```
>>> s1 = set()
>>> s2 = {1, '2', (3, '4')}  # tylko set, nie frozenset
>>> s3 = set(int(x)**2 for x in [5, '6'])
>>> s4 = frozenset('789')
>>> s5 = frozenset(('789',)) # zauważ przecinek
>>> s1, s2, s3
(set(), {1, (3, '4'), '2'}, {25, 36})
>>> s4, s5
(frozenset({'8', '7', '9'}), frozenset({'789'}))
```

Tak jak dla innych kolekcji, dla zbiorów można stosować operatory **in** i **not in** oraz funkcję **len**:

```
>>> st = {1, '2', 3, '4'}
>>> len(st)
4
>>> 2 in st
False
>>> '2' in st
True
>>> '3' not in st
True
```

Dla zbiorów (ale nie typu **frozenset**), można dodawać (haszowalne) elementy za pomocą metody **add**, jak również je usuwać. Do usuwania służą trzy metody:

- **remove(elem)** — usuwa element elem, a jeśli takiego nie ma podnosi wyjątek **KeyError**;
- **discard** — usuwa element elem, ale *nie* podnosi wyjątku, jeśli takiego nie ma;
- **pop()** — usuwa jeden, ale dowolny, element, zwracając go, a podnosi wyjątek **KeyError** jeśli zbiór jest pusty.

Na przykład:

```
>>> st = {1, '2', 3, '4'}
>>> st.add(5)
>>> st.discard('6')  # elementu '6' nie ma, ale OK
>>> st.remove('2')
>>> st
{1, 3, 5, '4'}
>>> st.pop()
5
>>> st
{1, 3, '4'}
>>> st.remove(7)     # elementu 7 nie ma -> wyjątek
Traceback (most recent call last):
```

[1] Ale puste nawiasy klamrowe są literałem pustego *słownika*, a nie zbioru!

```
  File "<stdin>", line 1, in <module>
KeyError: 7
```

Zbiory mogą być kopiowane za pomocą metody **copy**. Jest to płytka kopia, ale zwykle nie stanowi to problemu, bo elementy zbiorów i tak nie są modyfikowalne:

```
>>> A = {1, 2, 3, 4}
>>> B = frozenset(A)
>>> Ac = A.copy()
>>> Bc = B.copy()
>>> type(Ac), Ac
(<class 'set'>, {1, 2, 3, 4})
>>> type(Bc), Bc
(<class 'frozenset'>, frozenset({1, 2, 3, 4}))
```

Zbiory mogą być porównywane poprzez wywoływanie metod, albo poprzez zastosowanie operatorów binarnych, którego oba operandy są zbiorami (ale może być tak, że jeden jest typu **set** a drugi **frozenset**). Wszystkie zwracają **True** lub **False**.

Poniżej A i B oznaczają zbiory.

- `A == B`, `A != B` — odpowiadają na pytanie, czy A jest (nie jest) takim samym zbiorem jak B; zbiory są równe, jeśli $A \subset B$ oraz $B \subset A$ (nie ma *metod* odpowiadających tym operatorom):[1]

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> A == B, A != B
(False, True)
```

- `A.isdisjoint(B)` — odpowiada na pytanie, czy przecięcie $A \cap B$ jest zbiorem pustym (nie ma operatora odpowiadającego tej metodzie):

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> C = {4, 5}
>>> A.isdisjoint(B), A.isdisjoint(C)
(False, True)
```

- `A.issubset(B)` lub `A <= B` — odpowiadają na pytanie czy $A \subset B$, czyli czy każdy element A należy też do B; w szczególności tak jest, jeśli `A == B`:

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> A.issubset(B), B.issubset(A)
(False, True)
>>> A <= B, B <= A
(False, True)
```

[1] Ale *można* użyć „magicznych” metod **__eq__** i **__ne__**.

- `A < B` — odpowiadają na pytanie czy $A \subset B$, ale $A \neq B$ czyli czy każdy element A należy też do B, ale `A != B` (nie ma metody odpowiadającej temu operatorowi):[1]

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> C = {1, 3}
>>> B < A, C < B
(True, False)
```

- `A.issuperset(B)` lub `A >= B` — odpowiadają na pytanie czy $A \supset B$, czyli czy każdy element B należy też do A; w szczególności tak jest, jeśli `A == B`:

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> A.issuperset(B), B.issuperset(A)
(True, False)
>>> A >= B, B >= A
(True, False)
```

- `A > B` — odpowiadają na pytanie czy $A \supset B$, ale $A \neq B$, czyli czy każdy element B należy też do A, ale `A != B` (nie ma metody odpowiadającej temu operatorowi):[2]

```
>>> A = {1, 2, 3}
>>> B = {1, 3}
>>> C = {1, 3}
>>> A > B, C > B
(True, False)
```

Zauważmy, że relacje zawierania się zbiorów i ich porównywania *nie* definiują porządku liniowego (*total order*), bo, na przykład, możliwa jest sytuacja, gdy wszystkie wyrażenia $A \subset B$, $B \subset A$ i $A = B$ są fałszywe. Tak więc kolekcje zbiorów nie są sortowalne.

Następna grupa operacji odpowiada działaniom na zbiorach dającym w wyniku zbiór. Tu znów można zastosować metodę, podając inny zbiór jako argument, albo operator. Różnica jest taka, że dla metod argumentem nie musi być **set** — może to być dowolny obiekt iterowalny dostarczający elementy zbioru; w dalszym ciągu będziemy ten obiekt oznaczać *elems. Dla operatorów jednakże, oba operandy muszą być zbiorami, choć niekoniecznie tego samego typu: jeden może być typu **set** a drugi **frozenset**. Typ wyniku będzie taki jak typ pierwszego operandu.

- `A.union(*elems)` lub `A | set1 | set2 | ...` — zwraca sumę zbiorów:

```
>>> A = {1, 2, 3}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.union(BS)
>>> type(C), C
```

[1] Ale *można* użyć „magicznej” metody **__lt__**.
[2] Ale *można* użyć „magicznej” metody **__gt__**.

```
(<class 'set'>, {1, 2, 3, 6})
>>>
>>> D = A | BF   # result is set
>>> type(D), D
(<class 'set'>, {1, 2, 3, 6})
>>>
>>> E = BF | A
>>> type(E), E   # result is frozenset
(<class 'frozenset'>, frozenset({1, 2, 3, 6}))
>>>
>>> F = A | BF | {7} | {'8'}  # operands must be sets
>>> type(F), F
(<class 'set'>, {1, 2, 3, 6, 7, '8'})
>>>
>>> G = A | [2, 3, 6]         # operand is not a set
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

- `A.intersection(*elems)` lub `A & set1 & set2 & ...` — zwraca przecięcie zbiorów:

```
>>> A = {1, 2, 3}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.intersection(BS)
>>> type(C), C
(<class 'set'>, {2, 3})
>>>
>>> D = A & BF
>>> type(D), D
(<class 'set'>, {2, 3})
>>>
>>> E = BF & A
>>> type(E), E
(<class 'frozenset'>, frozenset({2, 3}))
>>>
>>> F = A & BF & {3, 4}
>>> type(F), F
(<class 'set'>, {3})
```

- `A.difference(*elems)` lub `A - set1 - set2 - ...` — zwraca różnicę zbiorów:

```
>>> A = {1, 2, 3, 4}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.difference(BS)
>>> type(C), C
(<class 'set'>, {1, 4})
```

```
>>>
>>> D = A - BF
>>> type(D), D
(<class 'set'>, {1, 4})
>>>
>>> E = BF - A
>>> type(E), E
(<class 'frozenset'>, frozenset({6}))
>>>
>>> F = A - {3, 4}
>>> type(F), F
(<class 'set'>, {1, 2})
```

- `A.symmetric_difference(*elems)` lub `A ^ set1 ^ set2 ^ ...` — zwraca różnicę symetryczną zbiorów:

```
>>> A = {1, 2, 3, 4}
>>> BS = {2, 3, 6}
>>> BF = frozenset(BS)
>>>
>>> C = A.symmetric_difference(BS)
>>> type(C), C
(<class 'set'>, {1, 4, 6})
>>>
>>> D = A ^ BF
>>> type(D), D
(<class 'set'>, {1, 4, 6})
>>>
>>> E = BF ^ A
>>> type(E), E
(<class 'frozenset'>, frozenset({1, 4, 6}))
>>>
>>> F = A ^ {3, 4}
>>> type(F), F
(<class 'set'>, {1, 2})
```

Wymienione wyżej metody mogą być zastosowane zarówno do zbiorów typu **set** jak i **frozenset**. Są jednak inne, które modyfikują zbiór: te mogą być stosowane *tylko* dla zbiorów typu **set**. Wszystkie one zwracają **None**.

- `A.update(*elems)` lub `A |= set1 | set2 | ...` — modyfikuje A dodając podane elementy:

```
>>> A = {1, 2, 3, 4}
>>>
>>> id(A), A
(139927670937568, {1, 2, 3, 4})
>>>
>>> A.update({2, 4, 5, 6})
>>> id(A), A
(139927670937568, {1, 2, 3, 4, 5, 6})     # to samo id
```

```
>>>
>>> A |= {6} | {7}
>>> id(A), A
(139927670937568, {1, 2, 3, 4, 5, 6, 7}) # to samo id
```

- `A.intersection_update(*elems)` lub `A &= set1 & set2 & ...` — modyfikuje A pozostawiając tylko przecięcie A i podanych elementów:

```
>>> A = {1, 2, 3, 4}
>>>
>>> id(A), A
(139927690835584, {1, 2, 3, 4})
>>>
>>> A.intersection_update({2, 4, 5, 6})
>>> id(A), A
(139927690835584, {2, 4})
>>>
>>> A &= {4, 6} | {4, 7}
>>> id(A), A
(139927690835584, {4})
```

- `A.difference_update(*elems)` lub `A -= set1 | set2 | ...` — modyfikuje A pozostawiając tylko różnicę A i podanych elementów:

```
>>> A = {1, 2, 3, 4, 5, 6, 7, 8, 9}
>>>
>>> id(A), A
(139927670938688, {1, 2, 3, 4, 5, 6, 7, 8, 9})
>>>
>>> A.difference_update({2, 4})
>>> id(A), A
(139927670938688, {1, 3, 5, 6, 7, 8, 9})
>>>
>>> A -= {1, 2} | {6, 7}
>>> id(A), A
(139927670938688, {3, 5, 8, 9})
```

- `A.symmetric_difference_update(*elems)` lub `A ^= set1` — modyfikuje A tak, że zawiera tylko elementy A lub set1, ale nie obu:

```
>>> A = {1, 2, 3, 4, 5}
>>> B = A.copy()
>>> A.symmetric_difference_update({4, 5, 6, 7})
>>> B ^= {4, 5, 6, 7}
>>> A, B
({1, 2, 3, 6, 7}, {1, 2, 3, 6, 7})
```

- `A.clear()` — modyfikuje A tak, że staje się zbiorem pustym:

```
>>> A = {1, 2, 3, 4, 5}
>>> A
{1, 2, 3, 4, 5}
>>> A.clear()
>>> A
set()
```

7.4 Słowniki

Typy mapujące (ang. *mappings*) w Pythonie odpowiadają mapom znanym nam z Javy. Są to w tym języku struktury danych o kluczowym znaczeniu — właściwie implementacja samego języka jest oparta w dużej mierze na mapach.

Jest tylko jeden wbudowany typ mapujący, **dict** (od *dictionary*), która odpowiada klasie **LinkedHashMap** z Javy. Jest to struktura danych, której elementami są pary klucz/wartość. Jak wiemy, klucze muszą być unikalne, a ponieważ implementacja jest oparta na haszowaniu, również muszą być haszowalne (nie ma odpowiednika klasy **TreeMap** z Javy).[1] Słowniki „pamiętają" kolejność w jakiej ich elementy (pary klucz/wartość) były do nich wstawiane i zwracają je w tej właśnie kolejności podczas iteracji.

Tworzyć słowniki możemy na kilka sposobów:

- pusty słownik można utworzyć używając konstruktora z pustą listą argumentów, lub podając literał **{}** (puste nawiasy klamrowe):

  ```
  >>> d1, d2 = dict(), {}
  >>> type(d1), type(d2)
  (<class 'dict'>, <class 'dict'>)
  >>> len(d1), len(d2)
  (0, 0)
  ```

- używając literału w formie ujętej w klamry listy oddzielonych przecinkami par klucz/wartość, które z kolei oddzielone są znakiem dwukropka:

  ```
  >>> d = {'1': 'one', 'two': 2, 3: 3}
  >>> d
  {'1': 'one', 'two': 2, 3: 3}
  ```

- stosując w konstruktorze argumenty nazwane (ich nazwy, jako napisy, staną się kluczami; oczywiście muszą one mieć formę legalnych identyfikatorów):

  ```
  >>> d = dict(one=1, two='two', three='3')
  >>> d
  {'one': 1, 'two': 'two', 'three': '3'}
  ```

- jako rezultat składania:

  ```
  >>> d = {x: y for x in range(10) if 20 < (y := x**2) < 85}
  >>> d
  {5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
  ```

- poprzez przekazanie do konstruktora obiektu iterowalnego, którego każdy element sam jest obiektem iterowalnym zwracającym dokładnie dwie wartości — pierwsza stanie się kluczem, a druga skojarzoną z nim wartością; za obiektem iterowalnym można dodać argumenty nazwane:

[1] Lub **map** z C++. W C++ mamy klasę **unordered_map**, która jest oparta na haszowaniu, ale nie jest „linked" — porządek elementów jest w zasadzie nieprzewidywalny.

```
>>> ls = [ (2, 3), ['7', 7], ((x, x**2) for x in range(4, 6)) ]
>>> d = dict(ls)
>>> d
{2: 3, '7': 7, (4, 16): (5, 25)}
>>>
>>> keys = [1, 2, 3]
>>> vals = ('one', 'two', 'three')
>>> d1 = dict(zip(keys, vals))
>>> d1
{1: 'one', 2: 'two', 3: 'three'}
>>> d2 = dict(zip(keys, vals), four='four', six=6)
>>> d2
{1: 'one', 2: 'two', 3: 'three', 'four': 'four', 'six': 6}
```

- poprzez przekazanie do konstruktora innego słownika; znów, być może dodając argumenty nazwane:

```
>>> d = {'john': 'Doe', 'mary': 'Cooper'}
>>> d1 = dict(d, bill='Gatsby', kate='Coe')
>>> d1
{'john': 'Doe', 'mary': 'Cooper', 'bill': 'Gatsby', 'kate': 'Coe'}
```

- poprzez użycie metody klasowej **fromkeys(itarable[, val)**, która zwraca nowy słownik z kluczami wziętymi z iterable i wszystkimi wartościami ustawionymi na val (**None**, jeśli val nie podano). Wszystkie wartości są referencjami do dokładnie tego samego obiektu, więc zwykle nie ma sensu wstawiać tu obiektu modyfikowalnego, jak na przykład listy:

```
>>> ls = ['john', 'mary']
>>> dict.fromkeys(ls)
{'john': None, 'mary': None}
>>> dict.fromkeys(ls, 0)
{'john': 0, 'mary': 0}
>>> d = dict.fromkeys(ls, [])     # john and mary wskazują
>>> d['john'].extend([1, 2, 3])   # na tę samą listę!
>>> d['mary']
[1, 2, 3]
```

Może się zdarzyć, że przy tworzeniu słownika ten sam klucz zostanie użyty więcej niż raz z różnymi wartościami — „wygrywa” wtedy wartość ostatnia, ale oryginalna kolejność jest zachowana:

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d1 = dict(d, three=33, one=11)
>>> d1
{'one': 11, 'two': 2, 'three': 33}
```

Podsumujmy operacje, których możemy użyć dla słowników (poniżej d oznacza słownik):

- `len(d)` — zwraca liczbę kluczy (albo wpisów, co jest tym samym) w d.
- `list(d)` — zwraca listę *kluczy*, (nie wpisów) w d:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> list(d)
['Joan', 'Bob', 'Tina']
```

- `d[key]` — zwraca wartość skojarzoną z `key` w `d`; zgłasza wyjątek **KeyError** jeśli w `d` takiego klucza nie ma (zobacz też metodę **setdefault** poniżej:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> d['Bob']
'Dylan'
>>> d['Beth']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Beth'
```

- `d[key]=val` — związuje wartość `val` z kluczem `key`; jeśli taki klucz istnieje, nowa wartość zastępuje starą, jeśli nie, wstawiany do `d` jest nowy wpis:

```
>>> d = dict(Joan='Baez', Bob='Cocker')
>>> d['Bob'] = 'Dylan'
>>> d['Tina'] = 'Turner'
>>> d
{'Joan': 'Baez', 'Bob': 'Dylan', 'Tina': 'Turner'}
```

- `del d[key]` — usuwa wpis z podanym kluczem z `d`; jeśli taki klucz nie istnieje, zgłasza wyjątek **KeyError** (zobacz też metodę **pop** poniżej):

```
>>> d = dict(Joan='Baez', Bob='Cocker')
>>> del d['Bob']
>>> d
{'Joan': 'Baez'}
>>> del d['Tina']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Tina'
```

- `key in d`, `key not in d` — opowiadają na pytanie, czy taki klucz występuje (nie występuje) w `d`:

```
>>> d = dict(Joan='Baez', Bob='Cocker')
>>> 'Joan' in d
True
>>> 'Tina' in d
False
>>> 'Tina' not in d
True
```

- `iter(d)` — zwraca iterator po *kluczach* z `d`; aby iterować po wpisach, czyli parach klucz/wartość (jako 2-krotkach) iteruj po `d.items()`:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> for e in d:      # równoważne 'for e in iter(d):'
...     print(e)     # oraz 'for e in d.keys():'
...
Joan
Bob
Tina
>>>
>>> for e in d.items():
...     print(e)
...
('Joan', 'Baez')
('Bob', 'Dylan')
('Tina', 'Turner')
```

- `d.clear()` — zmienia słownik `d` na pusty.

- `d.copy()` — zwraca *płytką* kopię słownika `d`:

```
>>> d = {'Joan': [1, 2], 'Bob': 'Turner'}
>>> c = d.copy()
>>> d['Joan'].append('X')
>>> d['Bob'] = 'Dylan'
>>> d
{'Joan': [1, 2, 'X'], 'Bob': 'Dylan'}
>>> c
{'Joan': [1, 2, 'X'], 'Bob': 'Turner'}
```

- `d.get(key[, default])` — zwraca wartość skojarzoną z `key` jeśli istnieje; jeśli nie, zwraca `default` (`None` jeśli `default` pominięto). Nigdy nie modyfikuje słownika i nie zgłasza wyjątku `KeyError`.

- `d.setdefault(key[, default])` — zwraca wartość skojarzoną z `key` jeśli istnieje; w przeciwnym przypadku dodaje nowy wpis do słownika `d` z podanym kluczem i wartością `default` (`None` jeśli `default` nie podano) i zwraca tę wartość:[1]

```
>>> d = {'Joan': 'Baez'}
>>> d.setdefault('Joan', 'Turner')
'Baez'
>>> d.setdefault('Bob', 'Dylan')
'Dylan'
>>> d
{'Joan': 'Baez', 'Bob': 'Dylan'}
```

- `d.pop(key[, default])` — jeśli wpis z kluczem `key` istnieje w słowniku, to jest usuwany, a wartość z nim skojarzona jest zwracana. W przeciwnym przypadku słownik nie jest modyfikowany, a zwracana jest wartość `default`, która w tym przypadku musi być podana (jeśli nie jest, wystąpi wyjątek `KeyError`):

[1] Podobnie do **putIfAbsent** dla map w Javie.

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> d.pop('Bob')
'Dylan'
>>> d
{'Joan': 'Baez', 'Tina': 'Turner'}
>>> d.pop('Joe', 'No such element')
'No such element'
>>> d.pop('Joe')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Joe'
```

- `d.popitem()` — zwraca ostatnio wstawiony wpis[1] (parę klucz/wartość jako 2-krotkę) i usuwa go ze słownika; zgłasza wyjątek KeyError, jeśli słownik jest pusty:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> while (d):
...     print(d.popitem())
...
('Tina', 'Turner')
('Bob', 'Dylan')
('Joan', 'Baez')
```

- `reversed(d)` — zwraca odwrotny iterator po kluczach d:

```
>>> d = dict(Joan='Baez', Bob='Dylan', Tina='Turner')
>>> for n in reversed(d):
...     print(n)
...
Tina
Bob
Joan
```

- `d.update(arg)` — jeśli arg jest słownikiem, jego elementy są dodawane do d, przy czym dla kluczy istniejących w d nowa wartość zastępuje starą; jeśli arg jest obiektem iterowalnym, jego elementy same muszą być iterowalne i dostarczać dokładnie dwie wartości, z których pierwsza będzie kluczem a druga wartością nowych wpisów:

```
>>> d = dict(Joan='Turner', Bob='Cocker')
>>> d1 = {'Joan': 'Baez'}
>>> d.update(d1)
>>> d
{'Joan': 'Baez', 'Bob': 'Cocker'}
>>> ls = [('Bob', 'Dylan'), ['Joe', 'Cocker']]
>>> d.update(ls)
>>> d
{'Joan': 'Baez', 'Bob': 'Dylan', 'Joe': 'Cocker'}
```

- `d | dic` — zwraca nowy słownik będący złożeniem wpisów z d i z dic; dla powtarzających się kluczy, wartość będzie wzięta z dic:

[1] Jak mówiliśmy, słowniki „pamiętają” kolejność dodawanych wpisów.

```
>>> d1 = dict(Joan='Turner', Bob='Cocker')
>>> d2 = dict(Bob='Dylan', Joe='Cocker', Joan='Baez')
>>> d3 = d1 | d2
>>> d3
{'Joan': 'Baez', 'Bob': 'Dylan', 'Joe': 'Cocker'}
```

- `d.keys()`, `d.values()`, `d.items()` — zwracają *widok* (nie kopię) odpowiednio wartości, kluczy i wpisów słownika `d`. Widoki zapewniają *dynamiczny obraz* danych słownika, czyli modyfikacje słownika są widoczne w jego widokach. Widoki są iterowalne i można dla nich stosować testy przynależności. Na przykład:

```
d = dict(Joan='Baez', Bob='Cocker', Tina='Turner')
v = d.items()
print(type(v), '\n')

for it in v:
    print(it)

print()

d.update(Bob='Dylan')
for it in v:
    print(it)

print()

print(('Joan', 'Baez')  in v)
print(('Joe', 'Cocker') in v)
```

prints

```
<class 'dict_items'>

('Joan', 'Baez')
('Bob', 'Cocker')
('Tina', 'Turner')

('Joan', 'Baez')
('Bob', 'Dylan')
('Tina', 'Turner')

True
False
```

Słowniki mogą być porównywane na równość (ale nie ma dla nich relacji mniejsze/większe, jak $<=$ or $>$). Dwa słowniki są uważane za równe, jeśli zawierają takie same pary klucz/wartość, choć niekoniecznie w tej samej kolejności:

```
>>> d1 = {'Bob': 'Dylan', 'Joan': 'Baez'}
>>> d2 = {'Joan': 'Baez', 'Bob': 'Dylan'}
>>> d1 == d2
True
```

7.5 Inne kolekcje

Oprócz kolekcji wbudowanych w język, jest w standardowej bibliotece kilka dodatkowych typów kolekcji, z których wymienimy tu najważniejsze.

7.5.1 Factory function *namedtuple*

Funkcja **namedtuple** (z modułu ***collections***) zwraca *typ,* który jest podklasą „normalnej" krotki (**tuple**). Typ ten, zwany krotką nazwaną, dodaje kilka pożytecznych cech do zwykłych krotek. Dla nich dostęp do elementów mamy poprzez indeksowanie; w krotkach nazwanych liczba elementów jest ustalona i elementy na każdej pozycji mają swoją nazwę (tak jak nazwy składowych w C-strukturach). Oczywiście, indeksów też możemy używać, jeśli chcemy, bo nazwane krotki *są* krotkami.

Dodatkowo, typy zwracane przez **namedtuple** mają już zdefiniowane rozsądne komentarze dokumentacyjne i już zaimplementowaną metodę **__repr__**.

Funkcja pochodzi z modułu ***collections*** i może być wywołana tak:

```
collections.namedtuple(typename, field_names, *,
                       rename=False, defaults=None, module=None)
```

Pierwszym argumentem jest nazwa, jaką chcemy nadać tworzonemu typowi. Rezultat, który jest typem, powinien być przypisany do zmiennej o tej samej nazwie, choć formalnie nie jest to wymagane. Po nazwie podajemy nazwy składowych krotki na kolejnych pozycjach: albo jako sekwencję napisów (na przykład `['red', 'green', 'blue']`), albo jako jeden napis z nazwami oddzielonymi białymi znakami i/lub przecinkami (na przykład `'r g b'` lub `'r, g, b'`). Pozostałe argumenty są opcjonalne i rzadko używane.

Gdy mamy już nowy typ, możemy tworzyć jego obiekty poprzez przekazanie do konstruktora odpowiednich wartości — możemy użyć argumentów pozycyjnych, albo nazwanych — w tym drugim przypadku, kolejność jest dowolna i można też rozpakować istniejący słownik (jak w przykładzie poniżej). Krotki nazwane mają też dodatkową metodę klasową, **_make(it)**, która tworzy obiekt na podstawie it, gdzie it jest obiektem iterowalnym zwracającym wartości dla wszystkich składowych nazwanej krotki.

Do składowych nazwanej krotki możemy się odnosić albo poprzez indeksy, albo za pomocą notacji z kropką (jak do składowych C-struktury), albo wywołując wbudowaną funkcję **getattr**. Ponieważ nazwane krotki są krotkami, można ich elementy rozpakowywać do nazwanych zmiennych (zob. rozdz. 7.6, str. 154).

```
>>> import collections
>>> Color = collections.namedtuple('Color', 'r g b')
>>>                               # typ      pola
>>>
>>> orange = Color(255, 165, 0)       # argumenty pozycyjne
>>> orange[0], orange[1], orange[2]   # po indeksie
(255, 165, 0)
>>> orange.r, orange.g, orange.b      # po nazwie
(255, 165, 0)
>>> for e in orange:                  # iteracja
...     print(e)
...
255
```

```
165
0
>>> peach = Color(b=180, g=229, r=255) # argumenty nazwane
>>> peach
Color(r=255, g=229, b=180)             # __repr__ zdefiniowana
>>>
>>> red, green, blue = peach           # rozpakowywanie
>>> red, green, blue
(255, 229, 180)
>>>
>>> it = [255, 127, 80]
>>> coral = Color._make(it)            # z ob. iterowalnego
>>> coral
Color(r=255, g=127, b=80)
>>> getattr(coral, 'g')                # dostęp do składowych
127
>>>
>>> d = {'r': 93, 'g': 63, 'b': 211}
>>> iris = Color(**d)                  # rozpakowywanie słownika
>>> iris
Color(r=93, g=63, b=211)
```

Nazwane krotki, będąc krotkami, mają wszystkie metody zwykłych krotek, ale dodatkowo trzy metody i dwa pola. Jedną z tych metod, _make, już wymieniliśmy; dwie pozostałe to (poniżej nt oznacza nazwaną krotkę):

- `nt._asdict()` — zwraca słownik reprezentujący nt jako słownik z nazwami składowych jako kluczami:

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(2, 5)
>>> p._asdict()
{'x': 2, 'y': 5}
```

- `nt._replace(**kwargs)` — zwraca nazwaną krotkę z niektórymi albo wszystkimi polami zastąpionymi przez wartości pochodzące z kwargs:

```
>>> Point = collections.namedtuple('Point', ['x', 'y'])
>>> p = Point(2, 5)
>>> p1 = p._replace(y=4)
>>> p2 = p._replace(x=3, y=6)
>>> p1, p2
(Point(x=2, y=4), Point(x=3, y=6))
```

- `nt._fields` — zwraca atrybut _fields, który jest krotką nazw składowych nt:

```
>>> from collections import namedtuple
>>> Point = namedtuple('Point', ['x', 'y'])
>>> p = Point(2, 5)
>>> p._fields
('x', 'y')
>>> Point._fields   # to jest atrybut klasy
('x', 'y')
```

Atrybut ten może być użyty do tworzenia nowych typów nazwanych krotek „rozszerzających", w pewnym sensie, już istniejące:

```
>>> Pixel = namedtuple('Pixel', ('shade',)+Point._fields)
>>> px = Pixel(255, 2, 5)
>>> px
Pixel(shade=255, x=2, y=5)
```

7.5.2 Klasa *defaultdict*

Klasa **defaultdict** z modułu ***collections*** dziedziczy po wbudowanej klasie **dict**, a zatem posiada jej wszystkie własności. Jest jednak różnica w sposobie działania, gdy próbujemy odwołać się elementu słownika za pomocą nieistniejącego w nim klucza, co dla „zwykłych" słowników prowadzi do wyjątku. W przypadku **defaultdict**, zamiast najpierw sprawdzać, czy klucz istnieje, możemy zdefiniować sposób na dostarczenie wartości dla tego klucza, gdy do tej pory nie istniał. Jak zobaczymy może to uprościć kod.

Konstruktor **defaultdict** może być wywołany tak:

```
collections.defaultdict(callableOb=None, /[, ...])
```

Pierwszy argument, jeśli nie jest **None** (co jest jego wartością domyślną) musi być obiektem wywoływalnym. Pozostałe argumenty są takie jak dla „zwykłych" słowników (zob. rozdz. 7.4, p. 137).

Jeśli callableOb został pominięty lub jest **None**, zwrócony obiekt zachowuje się tak jak „zwykły" słownik; w szczególności odwołania do nieistniejącego klucza spowoduje wyjątek **KeyError**.

Co się stanie jeśli jednak callableOb istnieje? Wtedy przy próbie odwołania się do nieistniejącego klucza obiekt zostanie wywołany bez żadnych argumentów i to co zostanie zwrócone będzie wartością dla podanego klucza i taki nowy wpis zostanie dodany do słownika.

Cokolwiek co może być wywołane bez argumentów może pełnić rolę callableOb. Jednak najczęściej jest to po prostu nazwa klasy: jak pamiętamy obiekty reprezentujące klasę są wywoływalne i bez argumentów zwracają domyślnie zainicjowany obiekt danej klasy (zero dla typów numerycznych, pustą kolekcję dla kolekcyjnych itd.).

Przykład. Mamy listę par (2-krotek) miast, gdzie *(city1, city2)* oznacza, że jest połączenie lotnicze z *city1* do *city2*. To co chcemy, to słownik z miastami jako kluczami i zbiorami miast osiągalnych z tego miasta jako wartościami. To jest jedna jednolinijkowa pętla, bez żadnych **if**ów!

Listing 36 ACC-DefDict/DefDict.py

```
1 #!/usr/bin/env python
2
3 from collections import defaultdict
4
5 flights = [ ('Athens', 'Basel'),      ('Athens', 'Copenhagen'),
6            ('Basel', 'Athens'),      ('Basel', 'Delhi'),
7            ('Copenhagen', 'Athens'), ('Delhi', 'Athens'),
8            ('Delhi', 'Basel'),       ('Delhi', 'Copenhagen') ]
```

```
9
10 destinations = defaultdict(set)  ❶
11
12 for start, to in flights:
13     destinations[start].add(to)  ❷
14
15   # just printing
16 for start, dests in destinations.items():
17     print(f'{start:10} -> {str(dests)}')
```

W linii ❶ tworzymy obiekt **defaultdict** z **set** jako obiektem wywoływalnym. Teraz, w linii ❷,

- jeśli miasto reprezentowane przez start pojawia się po raz pierwszy, nowy wpis jest tworzony z start jako kluczem i pustym zbiorem jako wartością, do którego to zbioru natychmiast dodajemy to;
- jeśli miasto start pojawiło się już wcześniej, to wartość to jest po prostu dodawana do już istniejącego zbioru miast, do których można dolecieć z miasta start.

Program drukuje

```
Athens     -> {'Basel', 'Copenhagen'}
Basel      -> {'Delhi', 'Athens'}
Copenhagen -> {'Athens'}
Delhi      -> {'Basel', 'Athens', 'Copenhagen'}
```

Inny przykład, tym razem z **int** jako callableOb. Mamy listę zakupów wraz z cenami, a chcemy policzyć całkowitą kwotę zapłaconą za poszczególne produkty:

Listing 37 ACE-Purchases/Purchases.py

```
1  #!/usr/bin/env python
2
3  from collections import defaultdict
4
5  purchases = [ ('Carrot', 3),   ('Beetroot', 7),
6                ('Apples', 9),   ('Beetroot', 8),
7                ('Apples', 8),   ('Apples', 10),
8                ('Beetroot', 6), ('Carrot', 4) ]
9
10 totals = defaultdict(int)
11
12 for prod, price in purchases:
13     totals[prod] += price      ❶
14
15   # just printing
16 for prod, tot in totals.items():
17     print(f'{prod:8} -> {tot:2}')
```

i wydruk jest

```
Carrot   ->  7
Beetroot -> 21
Apples   -> 27
```

To znowu jedna pętla z jednolinijkowym całem!

7.5.3 Klasa *Counter*

Klasa **Counter** (z modułu ***collections***) jest podklasą **dict** zaprojektowaną do przechowywania informacji o liczbie wystąpień poszczególnych elementów (kluczy). Tak więc jest to kolekcja elementów, przechowywanych jako klucze (a więc muszą to być elementy haszowalne!) z ich krotnościami jako skojarzonymi wartościami.[1] Może z tego wynikać, że wartości zawsze muszą tu być dodatnimi liczbami całkowitymi, ale w rzeczywistości zera i liczby ujemne *są* dozwolone. Nawet typ wartości nie musi formalnie być **int** — ułamki (**Fraction**), liczby zmiennoprzecinkowe (**float**) czy liczby typu **Decimal** też mogą działać, jak i inne typy wspierające dodawanie, odejmowanie i porównywanie. Jednak w praktyce są to zwykle liczy całkowite.

Konstruktor klasy **Counter** może być wywołany tak:

```
collections.Counter([iterable-or-mapping])
```

Jeśli argumentem konstruktora jest obiekt iterowalny, wystąpienia elementów będą przeliczone:

```
>>> import collections
>>> collections.Counter('abracadabra')
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
```

Jak widzimy, elementy są tu drukowane w kolejności malejących wartości (liczników). Jednak przy iteracji kolejność jest taka jak dla słowników — według kolejności pierwszego wystąpienia

```
>>> import collections
>>> c = collections.Counter([1, 5, 4, 2, 4, 4, 1, 2, 3, 3])
>>> c
Counter({4: 3, 1: 2, 2: 2, 3: 2, 5: 1})
>>> for p in c.items():
...     p
...
(1, 2)
(5, 1)
(4, 3)
(2, 2)
(3, 2)
```

Można też posłać do konstruktora słownik, w postaci obiektów **dict**, **defaultdict** czy **Counter** albo w postaci argumentów nazwanych:

```
>>> from collections import Counter
>>> Counter()                                    # empty
Counter()
```

[1] Podobnie do multisetów lub tzw. *bags* znanych z innych języków; w tym sensie są to raczej zbiory, ale w których elementy mogą występować wielokrotnie.

```
>>> Counter('azazello')                          # sequence
Counter({'a': 2, 'z': 2, 'l': 2, 'e': 1, 'o': 1})
>>> Counter({'a': 3, 'b': 2})                    # mapping
Counter({'a': 3, 'b': 2})
>>> Counter(a=3, b=2)                            # keyword args.
Counter({'a': 3, 'b': 2})
```

Próba odniesienia się do elementu o nieistniejącym kluczu nie powoduje wyjątku, ale zwraca zero; jeśli jest to przypisanie, to tworzony jest nowy wpis z licznikiem o wartości 0 (którą od razu możemy zmienić):

```
>>> from collections import Counter
>>> c = Counter()
>>> for letter in 'arrabal':
...     c[letter] += 1  # item with key letter added
...
>>> c
Counter({'a': 3, 'r': 2, 'b': 1, 'l': 1})
>>> c['z'] # no 'z', no assignment; 0 returned without adding
0
>>> c
Counter({'a': 3, 'r': 2, 'b': 1, 'l': 1})
```

Obiekty **Counter** są słownikami, więc dziedziczą metody z **dict**. Jednakże, są pewne zmiany: implementacja metody **fromkeys** jest usunięta, zaś implementacja **update** zmodyfikowana: dla już istniejących kluczy nowe wartości nie nadpisują poprzednich, ale są do nich dodawane:

```
>>> from collections import Counter
>>> c = Counter({'a': 3, 'b': 2, 'c': 1})
>>> d = dict(a=2, b=-2, c=-2, d=3)
>>> c.update(d)
>>> c
Counter({'a': 5, 'd': 3, 'b': 0, 'c': -1})
```

Prócz tego dodane są do klasy **Counter** dwie metody, których nie było w **dict** (poniżej cn oznacza obiekt **Counter**):

- `cn.elements()` — zwraca iterator po kluczach, przy czym każdy wystąpi tyle razy, ile wynosi jego krotność (licznik); elementy z niedodatnimi licznikami zostaną pominięte:

```
>>> from collections import Counter
>>> c = Counter({'a': 2, 'b': 0, 'c': -1, 'd': 3})
>>> for e in c.elements():
...     e
...
'a'
'a'
'd'
'd'
'd'
```

- `cn.most_common([n])` — zwraca listę `n` elementów (wpisów) o największych krotnościach, uporządkowane według malejących krotności. Jeśli `n` jest pominięte lub wynosi 0, zwracane są wszystkie elementy. Ponieważ jest to lista (2-krotek), można użyć wycinków dla uzyskania `n` elementów o *najmniejszych* krotnościach (na przykład według wzrastających krotności, jak w przykładzie poniżej):

```
>>> from collections import Counter
>>> c = Counter({'a': 6, 'b': 10, 'c': 1, 'd': -2})
>>> n = 2
>>> c.most_common(n)
[('b', 10), ('a', 6)]
>>> c.most_common()[:-n-1:-1]
[('d', -2), ('c', 1)]
```

- `cn.subtract([iterable-or-mapping])` — jak **update**, ale odejmuje krotności, a nie dodaje (brakujące wartości są traktowane jako 0):

```
>>> from collections import Counter
>>> c1 = Counter({'a': 6, 'b': 10, 'c': 1})
>>> c2 = Counter({'a': 4, 'b': 10, 'd': 2})
>>> c1.subtract(c2)
>>> c1
Counter({'a': 2, 'c': 1, 'b': 0, 'd': -2})
```

- `cn.total()` — zwraca sumę krotności wszystkich elementów:

```
>>> from collections import Counter
>>> c = Counter({'a': 6, 'b': 10, 'c': -2})
>>> c.total()
14
```

Dla obiektów **Counter** możemy używać pełnego zestawu operatorów porównania (==, !=, <, <=, >, >=) tak jak dla „normalnych" zbiorów (zob. rozdz. 7.3, str. 130. Jeśli element występuje w obiekcie po jednej stronie operatora, ale nie występuje po drugiej, zakłada się, że brakująca wartość krotności wynosi 0, więc

```
>>> Counter(b=1) == Counter(a=0, b=1, c=0)
True
```

To, co jest porównywane, to krotności, więc na przykład

```
>>> Counter(a=-1, b=1) < Counter(b=2, c=1)
True
```

bo po lewej brakująca krotność 'c' przyjmowana jest za 0, i podobnie dla 'a' po prawej.

Na obiektach **Counter** można też wykonywać operacje jak na zbiorach, ale wszystkie operują na krotnościach, znów zakładając, że brakujące elementy mają krotność zerową. Obiekty po lewej i po prawej stronie mogą zawierać elementy z niedodatnimi krotnościami, ale wynik jest zawsze „prawdziwym" multisetem: zachowane zostaną tylko elementy o dodatniej krotności.

Dodawanie i odejmowanie dodaje i odejmuje krotności (i usuwa z wyniku elementy o niedodatnie):

```
>>> Counter(a=-1, b=1) + Counter(a=2, b=2, c=1, d=-1)
Counter({'b': 3, 'a': 1, 'c': 1})
>>> Counter(a=2, b=2, c=1, d=-1) - Counter(a=3, b=-3, c=-1)
Counter({'b': 5, 'c': 2})
```

Przecięcie i suma (unia) obliczają minima i maksima odpowiadających sobie krotności:

```
>>> Counter(a=2, b=2, c=1, d=-1) & Counter(a=3, b=-3, c=-1)
Counter({'a': 2})
>>> Counter(a=2, b=2, c=1, d=-1) | Counter(a=3, b=-3, c=-1)
Counter({'a': 3, 'b': 2, 'c': 1})
```

Są też w końcu jednoargumentowe (unarne) operatory $+$ i $-$: pierwszy jest równoważny dodawaniu (za pomocą $+$) pustego obiektu **Counter**, co sprowadza się do usunięcia elementów o niedodatniej krotności:

```
>>> Counter(a=2, b=2, c=0, d=-1) + Counter()
Counter({'a': 2, 'b': 2})
>>> +Counter(a=2, b=2, c=0, d=-1)
Counter({'a': 2, 'b': 2})
```

a drugi do odjęcia (za pomocą $-$) danego obiektu *od* obiektu pustego, czyli zamianie znaku wszystkich krotności i zachowaniu tylko elementów, które po tej operacji mają krotność dodatnią:

```
>>> Counter() - Counter(a=2, b=-2, c=0, d=-1)
Counter({'b': 2, 'd': 1})
>>> -Counter(a=2, b=-2, c=0, d=-1)
Counter({'b': 2, 'd': 1})
```

7.5.4 Klasa *deque*

Obiekty klasy **deque** reprezentują kolejki podwójne (ang. *deque:*)[1] czyli kolejki z szybkim (złożoność $\mathcal{O}(1)$) dostępem do elementu pierwszego i ostatniego, ale wolniejszym, $\mathcal{O}(n)$, do elementów wewnątrz. W Pythonie kolejki podwójne implementowane są za pomocą podwójnie wiązanych list. Mogą być podstawą do prostej implementacji stosów i kolejek. Zwykle implementowanie ich za pomocą podwójnie wiązanej listy to przesada, ale jest akceptowalne w Pythonie, gdyż klasa **deque** jest w całości zaimplementowana w C (przynajmniej w standardowym CPythonie), więc wydajność jest zadowalająca.

Obiekty **deque** tworzymy za pomocą konstruktora

```
collections.deque([iterable[, maxlen]])
```

który zwraca obiekt zainicjowany danymi z iterable (pusty, jeśli iterable nie jest podane). Opcjonalny drugi argument, maxlen, ustala maksymalny rozmiar kolekcji. Kiedy potem dodajemy elementy na jednym końcu, a kolejka jest pełna (osiągnęła rozmiar maxlen), odpowiednia liczba elementów jest usuwana na drugim końcu. Jeśli maxlen nie jest określone (lub jest **None**), rozmiar kolejki nie jest ograniczony. Wartość maxlen jest dostępna jako pole maxlen obiektu (**None**, jeśli kolejka jest nieograniczona).

[1] Wymawiane *dek*; od *double-ended queue*.

Obiekty **deque** są sekwencjami, zatem można po nich iterować, stosować dla nich funkcje **len** i **reversed**, można je kopiować, sprawdzać zawartość za pomocą **in** i **not in**; również indeksowanie i wycinki mogą być stosowane, choć należy pamiętać, że indeksowanie jest na ogół operacją o złożoności $\mathcal{O}(n)$.

Poniżej opisujemy metody, które mogą być zastosowane dla obiektów **deque** (d oznacza taki obiekt):

- `d.append(e)`, `d.appendleft(e)` — dodaje element na początku (na końcu) d. Jeśli kolejka jest ograniczona i pełna, pierwszy element z przeciwnej strony jest usuwany:

```
>>> from collections import deque
>>> d = deque([], 3)  # pusta i ograniczona
>>> d
deque([], maxlen=3)
>>> d.append(2)
>>> d.appendleft(1)
>>> d.append(3)
>>> d.append(4)       # czwarty element?
>>> d
deque([2, 3, 4], maxlen=3)
```

- `d.insert(ind, e)` — dodaje element e na pozycji ind. Zgłasza wyjątek **IndexError**, jeśli prowadziłoby to do przekroczenia liczby elementów maxlen:

```
>>> from collections import deque
>>> d = deque([1, 2, 5, 6], 5)
>>> d.insert(2, 3)
>>> d
deque([1, 2, 3, 5, 6], maxlen=5)
>>> d.insert(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: deque already at its maximum size
```

- `d.extend(iterable)`, `d.extendleft(iterable)` — dodaje elementy z iterable na początku (końcu) d. Jeśli kolejka jest ograniczona, elementy z drugiej strony są usuwane, tak, aby całkowita liczba elementów nie przekroczyła maxlen:

```
>>> from collections import deque
>>> d = deque([3, 4], 6)     # ograniczona
>>> d.extendleft((2, 1))     # zauważ kolejność
>>> d
deque([1, 2, 3, 4], maxlen=6)
>>> d.extend([5, 6, 7, 8])
>>> d
deque([3, 4, 5, 6, 7, 8], maxlen=6)
```

- `d.pop()`, `d.popleft()` — zwraca ostatni (pierwszy) element i usuwa go:

```
>>> from collections import deque
>>> d = deque([1, 2, 3, 4, 5])
>>> d.pop()
5
>>> d.pop()
4
>>> d.popleft()
1
>>> d
deque([2, 3])
```

- `d.remove(val)` — usuwa pierwszy element równy val, lub zgłasza wyjątek **ValueError** jeśli takiego elementu nie ma:

```
>>> from collections import deque
>>> d = deque([4, 3, 5, 2, 1])
>>> d.remove(5)
>>> d
deque([4, 3, 2, 1])
>>> d.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in deque
```

- `d.reverse()` — odwraca w miejscu kolejność elementów w d:

```
>>> from collections import deque
>>> d = deque([4, 3, 2, 1])
>>> d
deque([4, 3, 2, 1])
>>> d.reverse()
>>> d
deque([1, 2, 3, 4])
```

- `d.index(val[, start[, stop]])` — zwraca indeks pierwszego elementu równego val, opcjonalnie przeszukując tylko elementy z zakresu indeksów od start do stop-1; zgłasza **ValueError** jeśli takiego elementu nie ma. Wartości domyślne start i stop to odpowiednio 0 i len(d):

```
>>> from collections import deque
>>> d = deque([1, 2, 3, 3, 1, 4])
>>> d.index(2)
1
>>> d.index(1, 1)
4
>>> d.index(1, 0, 4)
0
>>> d.index(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in deque
```

- `d.clear()` — usuwa wszystkie elementy z `d` pozostawiając obiekt pusty.
- `d.copy()` — zwraca *płytką* kopię `d`:

```
>>> from collections import deque
>>> d = deque([ (1, 2), [3, 4] ])
>>> dc = d.copy()
>>> d
deque([(1, 2), [3, 4]])
>>> dc
deque([(1, 2), [3, 4]])
>>> d[1][1] = 'X'
>>> d
deque([(1, 2), [3, 'X']])
>>> dc
deque([(1, 2), [3, 'X']])
```

- `d.count(val)` — zwraca liczbę elementów równych `val`:
- `d.rotate(n=1)` — rotuje elementy z `d` o `n` pozycji w prawo (w lewo dla `n` ujemnego):

```
>>> from collections import deque
>>> d = deque([1, 2, 3, 4, 5, 6, 7])
>>> d.rotate(2)
>>> d
deque([6, 7, 1, 2, 3, 4, 5])
>>> d.rotate(-4)
>>> d
deque([3, 4, 5, 6, 7, 1, 2])
>>> d.rotate()
>>> d
deque([2, 3, 4, 5, 6, 7, 1])
```

Ograniczone podwójne kolejki mogą być użyte w częstej sytuacji, gdy chcemy pamiętać ostatnich n wartości z sekwencji danych, na przykład n ostatnich rozmów telefonicznych, albo ostatnich n zapytań do bazy danych, itd.:

```
>>> from collections import deque
>>>
>>> calls = ['Mike', 'Jim', 'Alice', 'Kate', 'Bob', 'Sue']
>>> lastcalls = deque([], maxlen=3)
>>> for call in calls:
...     lastcalls.append(call)
...     lastcalls
...
deque(['Mike'], maxlen=3)
deque(['Mike', 'Jim'], maxlen=3)
deque(['Mike', 'Jim', 'Alice'], maxlen=3)
deque(['Jim', 'Alice', 'Kate'], maxlen=3)
deque(['Alice', 'Kate', 'Bob'], maxlen=3)
deque(['Kate', 'Bob', 'Sue'], maxlen=3)
```

7.6 Rozpakowywanie obiektów iterowalnych

Obiekty iterowalne (raczej „małe”...) łatwo jest „rozpakować” do krotki lub listy pojedynczych zmiennych o różnych nazwach. Jest to swego rodzaju przypisanie: po lewej stronie umieszczamy krotkę (zazwyczaj bez nawiasów) lub listę zmiennych, do których mają być przypisane kolejne elementy obiektu iterowalnego: liczba tych elementów musi być równa liczbie zmiennych po lewej stronie.
Obiekt po prawej stronie może być czymkolwiek, co jest iterowalne: może to być napis, lista, krotka, wycinek, obiekt **range**, zbiór, słownik, generator:

```
  # 1. do krotki
a, b, c, d = range(4)
print('** 1 ** ', f'{a=}, {b=}, {c=}, {d=}')

  # 2. do listy
[e, f, g, h] = range(4)
print('** 2 ** ', f'{e=}, {f=}, {g=}, {h=}')

  # 3. z wycinka
ls = [0, 1, 2, 'three', 4, 5, 6, 'seven', 8, 'nine']
i, j, k, l = ls[1:8:2]
print('** 3 ** ', f'{i=}, {j=}, {k=}, {l=}')

  # 4. elementy mogą same być krotkami lub listami
m, n, o, p = (1, 2, (3, 4, 5), [6, 7])
print('** 4 ** ', f'{m=}, {n=}, {o=}, {p=}')

  # 5. z napisu
q, r, s = 'QRS'
print('** 5 ** ', f'{q=}, {r=}, {s=}')

  # 6. z wycinka napisu
t, u, v = '012345678'[2:9:3]
print('** 6 ** ', f'{t=}, {u=}, {v=}')

  # 7. z generatora
w, x, y = (z**3 for z in range(1,10) if z%3 == 1)
print('** 7 ** ', f'{w=}, {x=}, {y=}')

  # 8. liczby elementów niezgodne -> wyjątek
p, q, r = range(4)
```

Program drukuje

```
** 1 **  a=0, b=1, c=2, d=3
** 2 **  e=0, f=1, g=2, h=3
** 3 **  i=1, j='three', k=5, l='seven'
** 4 **  m=1, n=2, o=(3, 4, 5), p=[6, 7]
** 5 **  q='Q', r='R', s='S'
** 6 **  t='2', u='5', v='8'
** 7 **  w=1, x=64, y=343
Traceback (most recent call last):
```

```
  File "/usr/lib/python3.10/idlelib/run.py", line 578,
    in runcode exec(code, self.locals)
  File "/home/werner/p.py", line 32, in <module>
    p, q, r = range(4)
ValueError: too many values to unpack (expected 3)
```

Zbiory są też iterowalne, choć dla nich pewnym problemem może być to, że nie możemy przewidzieć kolejności w jakiej elementy zbioru będą przypisywane do zmiennych po lewej stronie:

```
a, b, c, d = {1, 'Ann', 'Barbra', (2, 'c')}
print(f'{a=}, {b=}, {c=}, {d=}')
```

wydrukowało

```
a='Ann', b=(2, 'c'), c='Barbra', d=1
```

ale nie ma gwarancji, że kolejność zawsze będzie właśnie taka.

Iteracja dla słowników daje klucze. Możemy też iterować po parach (krotkach) klucz-wartość wywołując na rzecz słownika metodę **items**. Ale te krotki same są iterowalne, więc je też możemy rozpakować do osobnych zmiennych, jak pokazuje trzeci przykład z poniższego programu

```
dc = dict(x='Ann', y=23, z=[1, 2])

  # 1. klucze
a, b, c = dc
print('*1*', f'{a=}, {b=}, {c=}')

  # 2. wpisy (krotki klucz/wartość)
d, e, f = dc.items()
print('*2*', f'{d=}, {e=}, {f=}')

  # 3. rozpakowywanie zagnieżdżone
(k1, v1), kv, (k3, v3) = dc.items()
print('*3*', f'{k1=}, {v1=}, {kv=}, {k3=}, {v3=}')
```

który drukuje

```
*1* a='x', b='y', c='z'
*2* d=('x', 'Ann'), e=('y', 23), f=('z', [1, 2])
*3* k1='x', v1='Ann', kv=('y', 23), k3='z', v3=[1, 2]
```

Szczególnie często tego typu rozpakowywanie stosuje się w pętlach, gdy poszczególne elementy same są iterowalne (na przykład są wpisami słownika, czyli krotkami). Tu też możemy stosować rozpakowywanie zagnieżdżone; na przykład fragment

```
triangle = ( ("A", (1, 5)), ("B", (5, 3)), ("C", (2, 1)))

for symb, (x, y) in triangle:
    print(f'{symb} = ({x=}, {y=})')
```

drukuje

```
A = (x=1, y=5)
B = (x=5, y=3)
C = (x=2, y=1)
```

Nazwa jednej (i najwyżej jednej) zmiennej po lewej stronie może być poprzedzona gwiazdką, na przykład *v. Znaczy to, że „zebrane" do niej, w postaci listy, zostaną wszystkie elementy, które nie zostały dopasowane do „niegwiazdkowanych" zmiennych występujących przed i po *v. Na przykład program

```
tp = (1, 'two', [3, 4], {5, 'six'})

u, x, y, z = tp
print('*1*', f'{u=}, {x=}, {y=}, {z=}')

u, x, *y = tp
print('*2*', f'{u=}, {x=}, {y=}')

u, *x, z = tp
print('*3*', f'{u=}, {x=}, {z=}')

u, *_, z = tp  # convention: only u and z are of interest
print('*4*', f'{u=}, {z=}')

*u, y, z = tp
print('*5*', f'{u=}, {y=}, {z=}')
```

drukuje

```
*1* u=1, x='two', y=[3, 4], z={5, 'six'}
*2* u=1, x='two', y=[[3, 4], {5, 'six'}]
*3* u=1, x=['two', [3, 4]], z={5, 'six'}
*4* u=1, z={5, 'six'}
*5* u=[1, 'two'], y=[3, 4], z={5, 'six'}
```

Często lista odpowiadająca gwiazdkowanej zmiennej to elementy, które nas nie interesują — chodzi nam o „wyłowienie" tylko tych niegwiazdkowanych. Popularną konwencją jest wtedy nazwanie tej zmiennej _ (podkreślnik, jak w przykładzie 4 w powyższym programie); jest to prawidłowa nazwa, ale wskazuje czytelnikowi, że pojawia się tylko ze względu na wymagania składniowe, a jej wartość nie jest dla nas interesująca i nie będzie używana.

Rozpakowywanie krotek (lub list) jest też często używane do przestawiania wartości zmiennych (*swap*):

```
>>> a, b = 1, 2
>>> a, b
(1, 2)
>>> b, a = a, b    # swap
>>> a, b
(2, 1)
>>>
>>> a, b, c, d = 'three', 'four', 'one', 'two'
```

```
>>> a, b, c, d
('three', 'four', 'one', 'two')
>>> a, b, c, d = c, d, a, b
>>> a, b, c, d
('one', 'two', 'three', 'four')
```

Wiele funkcji zwraca rezultat w postaci krotki (lub generatora, jak w poniższym przykładzie). Wtedy też często rozpakowujemy od razu te krotki do nazwanych zmiennych (nieinteresujące nas elementy do _), jak w programie

```
def powers1_5(n):
    v = 1
    for i in range(1, 6):
        yield (v := v*n)

a, b, c, d, e = powers1_5(3)
print(a, b, c, d, e)

_, square, cube, *_ = powers1_5(3)
print(f'{square=}, {cube=}')
```

który drukuje

```
3 9 27 81 243
square=9, cube=27
```

Rozpakowywanie za pomocą gwiazdki może też uprościć składanie kilku sekwencji (ogólniej: obiektów iterowalnych) do jednej, jak w następującym fragmencie

```
st = {1, 2}
ls = [3, 4, 5]
def seq(n, m):
    for i in range(n, m+1):
        yield i

r = [*st, *ls, *seq(6, 7)]
print(r)
```

drukuje

```
[1, 2, 3, 4, 5, 6, 7]
```

Za pomocą podwójnej gwiazdki można też rozpakowywać słowniki; można też je rozpakowywać za pomocą jednej gwiazdki, ale wtedy iteracja da nam same klucze, a nie wpisy:

```
d1 = dict(a=1, b=2, c=3)
d2 = dict(c='three', d=4)

dk = {*d1, *d2}   # rozpakowywanie kluczy, dk jest zbiorem
print(dk)

di = {**d1, **d2} # rozpakowywanie wpisów, di jest słownikiem
print(di)
```

```
def getdic():
    return { 'e': 5, 'f': 6}

dd = {**di, **getdic()}  # dd jest słownikiem
print(dd)
```

drukuje

```
{'a', 'c', 'd', 'b'}
{'a': 1, 'b': 2, 'c': 'three', 'd': 4}
{'a': 1, 'b': 2, 'c': 'three', 'd': 4, 'e': 5, 'f': 6}
```

Rozdział 8

Napisy

Szczególnym rodzajem *niemodyfikowalnych* kolekcji są napisy — obiekty typu **str**. Ze względu na ich wyjątkowe znaczenie i specyficzną interpretację, posiadają one wiele dodatkowych metod, których inne kolekcje nie posiadają.

Napisy są sekwencją znaków, choć pamiętać trzeba, że nie ma w Pytonie osobnego typu **char** — obiektom typu **char** znanym z innych języków odpowiadają po prostu jednoznakowe napisy. Z definicji napisy są wewnętrznie zapisywane w kodowaniu UTF-8, a więc każdemu znakowi może odpowiadać od jednego do czterech bajtów.[1]

Literały napisowe mogą być ujęte w apostrofy ('like this') lub w cudzysłowy ("like this"). W pierwszym przypadku cudzysłowy oznaczają wewnątrz napisu same siebie, bez konieczności stosowania odwrotnych ukośników, a w drugim na odwrót. Dwa literały obok siebie oddzielone wyłącznie białymi znakami są przetwarzane do jednego ('This ' "and""that"jest równoważne 'This and that'). Jest też firma "potrójnie cytowana'"("'tak"' lub """tak"""); w tej formie wszystkie znaki białe, łącznie ze znakiem nowej linii, są zachowywane w napisie:

```
s = '''All white-space
characters
    included'''
print(s)
```

prints

```
All white-space
characters
    included
```

Wewnątrz literałów napisowych niektóre „specjalne" znaki muszą być wprowadzane za pomocą sekwencji ze znakiem \ jako pierwszym znakiem, podobnie jak w C/C++/Javie:

Tablica 3: Znaki „specjalne"

Sekwencja	Opis
\<newline>	ignorowane
\\	ukośnik wsteczny
\'	apostrof
\"	cudzysłów
\a	BEL (alarm)
\b	BS (backspace)
\f	FF (nowa strona)
\n	LF (koniec linii)
\r	CR (powrót karetki)
\t	TAB (tabulator poziomy)
\v	VT (tabulator pionowy)
\ooo	znak o kodzie ósemkowym *ooo*
\hh	8-bitowy znak o kodzie *hh* szesnastkowo

[1] https://en.wikipedia.org/wiki/UTF-8

Tablica 3: (kontynuacja)

Sekwencja	Opis
\uhhhh	16-bitowy znak o kodzie *hhhh* szesnastkowo
\Uhhhhhhhh	32-bitowy znak o kodzie *hhhhhhhh* szesnastkowo

Jednakże, dla „surowych” literałów (*raw* strings), z prefiksem ‘r’ lub ‘R’, ukośnik wsteczny jest interpretowany dosłownie — jest to bardzo wygodne przy definiowaniu wyrażeń regularnych:

```
>>> ss =  '\a\b\r\n'
>>> sr = r'\a\b\r\n'
>>> len(ss), len(sr)
(4, 8)
>>> sr
'\\a\\b\\r\\n'
```

Jednoparametrowy konstruktor klasy **str**, przyjmujący dowolny obiekt, jest odpowiednikiem metody **toString** z Javy — zwraca napis opisujący przesłany obiekt. Robi to wywołując na rzecz tego obiektu metodę **__str__**, a jeśli jej nie ma, to **__repr__**. Jeśli i tej nie ma, użyta zostanie domyślna implementacja **__str__** odziedziczona z klasy **object**:

```
class A:                        # tylko str
    def __str__(self):
        return "A(__str__)"

class B:                        # tylko repr
    def __repr__(self):
        return "B(__repr__)"

class C:                        # obie
    def __str__(self):
        return "C(__str__)"
    def __repr__(self):
        return "C(__repr__)"

class D:                        # żadnej
    pass

print(str(A()), str(B()), str(C()))
print(str(D()))
```

drukuje

```
A(__str__) B(__repr__) C(__str__)
<__main__.D object at 0x7f8f1a63bd60>
```

Jeśli pierwszym argumentem jest obiekt typu bajtowego (jak **bytes** lub **bytearray**) i podaliśmy jeszcze co najmniej jeden argument (określający kodowanie), to działanie jest nieco inne (patrz następne podrozdziały).

Spośród wielu metod klasy **str**[1] wymienimy tylko niektóre, najczęściej stosowane:

Sprawdzanie typu znaków

Szereg metod sprawdza typ znaków składających się na napis. Wszystkie one zwracają **True** lub **False**. Ich nazwy są podobne do tych jakie znamy ze standardowego C[2]; podobne występują też w Javie jako metody statyczne w klasie **Character**. Różnica jest taka, że w Pythonie są to niestatyczne metody klasy **str** i działają na *całych* napisach, a nie dla pojedynczych znaków. Tak więc

```
>>> '+123'.isdigit()
False
>>> '123'.isdigit()
True
```

bo w pierwszym przypadku napis zawiera znak '+', który cyfrą nie jest. Z drugiej strony metody **isupper** czy **islower** sprawdzają tylko znaki, dla których wielkość znaku ma sens, tak więc

```
>>> 'U.S.A.'.isupper()
True
>>> 'England'.isupper()
False
```

Do omawianej grupy metod należą następujące metody klasy **str**: **isalnum()**, **isalpha()**, **isascii()**, **isdecimal()**, **isdigit()**, **isidentifier()**, **keyword.iskeyword()**, **islower()**, **isnumeric()**, **isprintable()**, **isspace()**, **istitle()**, **isupper()**.

Przetwarzanie napisów

Inna grupa metod zwraca napis, który jest jakąś transformacją napisu na rzecz którego metody te są wywoływane. Możemy tu wymienić następujące metody (s oznacza napis na rzecz którego metoda jest wywoływana):

s.capitalize(), **s.title()** — **capitalize** zwraca napis z pierwszym znakiem przekształconym na dużą literę, jeśli jest małą literą; wszystkie inne litery są przekształcane na małą, a nie-litery nie są zmieniane. Metoda **title** robi to samo, ale dla każdego słowa z osobna, gdzie przez słowo rozumiana jest sekwencja liter:

```
>>> 'aDaM 12, BARBRA 18'.capitalize()
'Adam 12, barbra 18'
>>> '12 aDaM, 18 BARBRA'.capitalize()
'12 adam, 18 barbra'
>>> '12 aDaM, 18 BARBRA'.title()
'12 Adam, 18 Barbra'
```

s.ljust(width[, fillchar]), **s.rjust(width[, fillchar])**, **s.center(width[, fillchar])** — zwraca napis o długości width zawierający s na początku (**ljust**), na końcu (**rjust**) lub na środku (**center**) i wypełniony odpowiednio za, przed, lub po obu stronach s powtórzonym odpowiednią liczbę razy znakiem fillchar (domyślnie spacjami) aby uzyskać napis o długości width; jeśli width nie jest większe od len(s), zwracane jest s:

[1] https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
[2] Z nagłówka ***ctype.h*** w C lub ***cctype*** w C++.

```
>>> 'Adam'.ljust(10)
'Adam      '
>>> 'Adam'.ljust(10, '-')
'Adam------'
>>> 'Adam'.rjust(10, '=')
'======Adam'
>>> 'Adam'.center(10, '.')
'...Adam...'
```

s.zfill(width) — podobne do **rjust**, ale dodawane na początku są znaki '0' (zero), a jeśli *pierwszym* znakiem s jest '−' (minus), to pierwsze z zer jest zastępowane przez znak '−'; jeśli width nie jest większe od len(s), zwracane jest s. Metoda jest użyteczna dla napisów reprezentujących liczby:

```
>>> '123'.zfill(6)
'000123'
>>> '-123'.zfill(6)
'-00123'
>>> ' -123'.zfill(6)    # minus nie jest pierwszym znakiem
'0 -123'
>>> '-abc'.zfill(6)
'-00abc'
```

s.lstrip([chars]), **s.rstrip([chars])**, **s.strip([chars])** — zwraca napis bez początkowej (dla **lstrip**), końcowej (dla **rstrip**) lub i początkowej i końcowej (dla **strip**) części składającej się wyłącznie ze znaków zawartych w [chars]. Jeśli argumentu [chars] nie ma, albo jest None, „usuwane" są białe znaki:

```
>>> '11 23 Adam 99 34'.lstrip('0123456789 ')
'Adam 99 34'
>>> '11 23 Adam 99 34'.rstrip('0123456789 ')
'11 23 Adam'
>>> '    Adam\t\r\n'.strip()
'Adam'
```

s.removeprefix(prefix), **s.removesuffix(suffix)** — zwraca napis bez początkowego (dla **removeprefix**) lub końcowego (dla **removesuffix**) fragmentu jeśli jest on dokładnie taki jak podany argument; jeśli nie, to zwracane jest s:

```
>>> 'Mr Jones'.removeprefix('Mr ')
'Jones'
>>> 'Mona Lisa.jpg'.removesuffix('.jpg')
'Mona Lisa'
```

s.lower(), **s.upper()**, **s.swapcase()** — zwraca napis ze wszystkimi dużymi literami zastąpionymi przez odpowiadające im małe litery (**lower**), odwrotnie (**upper**) lub jedno i drugie (**swapcase**):

```
>>> 'bArBrA 18'.lower()
'barbra 18'
```

```
>>> 'bArBrA 18'.upper()
'BARBRA 18'
>>> 'bArBrA 18'.swapcase()
'BaRbRa 18'
```

s.replace(old, new[, count]) — zwraca kopię s zastępując wszystkie wystąpienia old napisem new. Jeśli count jest podane, tylko count pierwszych wystąpień old będzie zastąpione.

```
>>> 'AA BB AAA C AAAA D AAAAA'.replace('AA', 'X')
'X BB XA C XX D XXA'
>>> 'AA BB AAA C AAAA D AAAAA'.replace('AA', 'X', 3)
'X BB XA C XAA D AAAAA'
```

Wyszukiwanie podnapisów

Kilka metod klasy **str** dotyczy wyszukiwania podnapisów w danym napisie. Następujące metody należą do tej kategorii.

s.find(sub[, start[, end]]), **s.rfind(sub[, start[, end]])**, **s.index(sub[, start[, end]])**, **s.index(sub[, start[, end]])** — zwraca najmniejszy (**find** i **index**) lub największy (**rfind** i **rindex**) indeks w s gdzie rozpoczyna się podnapis sub. Jeśli argumenty opcjonalne zostały podane, sub jest poszukiwany tylko w części s[start:end]; domyślna wartość start to 0, a end to len(s). Jeśli sub w ogóle nie występuje w s, zwracane jest -1 dla **find** i **rfind**, natomiast podnoszony jest wyjątek **ValueError** dla **index** i **rindex**:

```
>>> s = 'barbaric'
>>> s.find('bar'), s.rfind('bar')
(0, 3)
>>> s.index('bar'), s.rindex('bar')
(0, 3)
>>> s.find('br')
-1
>>> s.index('br')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

s.count(sub[, start[, end]]) — zwraca liczbę niepokrywających się wystąpień podnapisu sub w s, lub w wycinku s[start:end] jeśli podane zostały argumenty opcjonalne:

```
>>> 'bcbcbcb'.count('bcb')
2
```

Zauważ że podnapis 'bcb' rozpoczynający się od drugiego 'b' nie został policzony, gdyż częściowo pokrywa się z pierwszym wystąpieniem 'bcb' na początku napisu.

s.endswith(sub[, start[, end]]), **s.startswith(sub[, start[, end]])** — sprawdza, czy s, lub wycinek s[start:end] jeśli argumenty opcjonalne zostały podane, kończy/rozpoczyna się dokładnie podnapisem sub; zwraca **True** lub **False**. Argument sub jest traktowany dosłownie, *nie* jako wyrażenie regularne.

```
>>> 'caravaggio.jpg'.endswith('.jpg')
True
>>> 'caravaggio.jpg'.startswith('Car')
False
```

Podział napisu na „słowa”

Kilka metod pozwala rozbić napis na listę „słów”.

s.split(sep=None, maxsplit=-1), **s.rsplit(sep=None, maxsplit=-1)** — zwraca listę „słów” w s, traktując sep jako separator między nimi. Jeśli maxsplit jest podany, będzie to maksymalna liczba podziałów, czyli maksymalna liczba „słów” będzie wynosiła maxsplit+1 i ostatni element wynikowej listy będzie zawierał wszystkie pozostałe „słowa” (**split**). Metoda **rsplit** działa „od końca” — wykonuje co najwyżej maxsplit podziałów od prawej strony, a pozostałe z lewej strony „słowa” znajdą się w pierwszym elemencie wynikowej listy. Jeśli maxsplit nie jest podane, albo jest -1, nie ma ograniczeń na liczbę podziałów. Jeżeli sep jest podane, to dwa separatory obok siebie ograniczają napis pusty (który, wobec tego znajdzie się na wynikowej liście). Jeśli s zaczyna się od separatora, pierwszym zwróconym słowem będzie napis pusty; podobnie, jeśli s kończy się separatorem, pusty napis będzie ostatnim „słowem”. Separator sep może być napisem o dowolnej długości, niekoniecznie jednoznakowym. Jeżeli sep nie jest podane, albo jest **None**, każda niepusta sekwencja białych znaków będzie traktowana jako pojedynczy separator. Wtedy też takie sekwencje na początku i na końcu będą ignorowane, a zatem na wynikowej liście nigdy *nie* pojawią się elementy puste.

```
>>> ' a   b c   d '.split()
['a', 'b', 'c', 'd']
>>> ':a::b:'.split(':')
['', 'a', '', 'b', '']
>>> 'a b c d e f'.split(maxsplit=3)
['a', 'b', 'c', 'd e f']
>>> 'a b c d e f'.rsplit(maxsplit=3)
['a b c', 'd', 'e', 'f']
```

s.splitlines(keepends=False) — zwraca listę linii w s. Za separatory pomiędzy liniami uważane są znaki nowej linii (\n), powrotu karetki (\r), pionowego tabulatora (\v) nowej strony (\f) i kilka innych, rzadziej używanych. Jednak kombinacja CR-LF (\r\n) jest uznawana za jeden pojedynczy separator. Same separatory nie są dołączane do wynikowych linii, chyba że keepends jest **True**. Separator na samym końcu s jest ignorowany (ale tylko jeden, nie ich sekwencja). Separator na początku s *nie* jest ignorowany.

```
>>> '\nA\r\nB\rC\nD\n\nE\v\f'.splitlines()
['', 'A', 'B', 'C', 'D', '', 'E', '']
```

s.partition(sep), **s.rpartition(sep)** — zwraca 3-krotkę zawierającą część przed pierwszym separatorem sep, sam separator, i pozostałą część s. Jeżeli sep nie występuje w s, zwracana jest 3-krotka zawierająca samo s, i dwa napisy puste. Metoda **rpartition** jest podobna, ale dzieli s na *ostatnim* wystąpieniu sep, a jeśli sep nie występuje w s, to zwracana 3-krotka zawiera dwa napisy puste i za nimi samo s.

```
>>> 'A:B:C:D'.partition(':')
('A', ':', 'B:C:D')
>>> 'A:B:C:D'.rpartition(':')
('A:B:C', ':', 'D')
>>> 'A:B:C:D'.partition('-')
('A:B:C:D', '', '')
>>> 'A:B:C:D'.rpartition('-')
('', '', 'A:B:C:D')
```

Łączenie napisów

Nie ma w Pythonie typu odpowiadającego modyfikowalnym napisom, jak **string** w C++ czy **StringBuffer**/**StringBuilder** w Javie. Jednakże w wielu sytuacjach można podobną funkcjonalność osiągnąć za pomocą metody **join** klasy **str**.

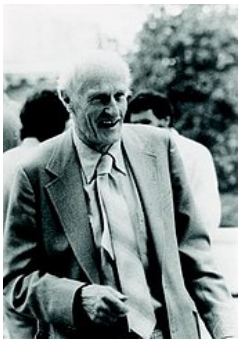
s.join(iterable) — zwraca napis będący złożeniem (konkatenacją) wszystkich wartości z iterable — muszą one być typu **str**; jakakolwiek wartość innego typu spowoduje wyjątek **TypeError**. W wynikowym napisie poszczególne elementu będą oddzielone separatorem s, który może być, i często jest, napisem pustym.

```
>>> lst = ['To', 'be', 'or', 'not', 'to', 'be']
>>> ' '.join(lst)
'To be or not to be'
>>> '-'.join(a+b for a in ['A', 'B'] for b in ('1', '2'))
'A1-A2-B1-B2'
```

Rozdział 9

Wyrażenia regularne

Wyrażenia regularne (w skrócie – *regeksy*) to sekwencje znaków znaków definiujących wzorzec (pod)napisów które chcemy odnajdywać w zadanym tekście (dowolnie długim). Regeksy są „kompilowane" przez interpreter do formy przypominającej funkcje i później wykonywane przez tzw. silnik wyrażeń regularnych — większość współczesnych języków programowania wspiera wyrażenia regularne (wbudowane w język bądź jako część ich bibliotek standardowych). Teoria stojąca za obsługą wyrażeń regularnych jest raczej złożona i jej pełne zrozumienie wymaga sporej wiedzy matematycznej; opracował ją wybitny logik Stephen Cole Kleene (wymawiane jak KLAY-nee) w latch pięćdziesiątych i została po raz pierwszy wykorzystana we wczesnych implementacjach Unixowych procesorach tekstu i innych programach użytkowych tego systemu (Ken Thompson). Większość współczesnych implementacji jest oparta na tej opracowanej dla języka Perl (Larry Wall, późne lata osiemdziesiąte).



Stephen C. Kleene

Szczegóły implementacji w Pytonie można znaleźć w dokumentacji,[1] (warto też zajrzeć do tutoriala[2]).

9.1 Podstawy

Przypuśćmy, że szukamy w tekście słowa `elephant`. Taki dosłownie teks znajdziemy definiując regeks `"elephant"`. Ale co jeśli w tekście jest `Elephant`? To nie będzie pasować, bo pierwsza litera jest inna. Albo szukamy `cat`, ale jako oddzielnego słowa, a nie jako części jakiegoś słowa (jak w `tomcat` albo `caterpillar`). Albo szukamy liczb (sekwencji cyfr), ale nie wiemy z góry jakie to będą cyfry i ile ich będzie. Wszystkie te problemy rozwiążemy za pomocą wyrażeń regularnych, które pozwalają na sformułowanie takich warunków jak 'sekwencja liter', 'duża litera, za którą jest kropka', 'sekwencja cyfr, przynajmniej czterech ale najwyżej siedmiu, z których pierwsza nie jest zerem', 'dwa słowa oddzielone jedną lub wieloma spacjami albo znakami tabulacji', itd.

Narzędzia do obsługi wyrażeń regularnych są w Pytonie zawarte w module ***re*** należącym do biblioteki standardowej. Jest też zewnętrzna popularna biblioteka ***regex*** z API zgodnym ze standardowym modułem ***re***, ale oferująca dodatkowe funkcjonalności — można ją znaleźć na stronie PyPI[3].

9.1.1 Metaznaki

Większość znaków w wyrażeniach regularnych oznacza siebie same, ale są takie o specjalnym znaczeniu, tak zwane **metaznaki**; aby ich użyć w regeksie dosłownie, należy zwykle poprzedzić je wstecznym ukośnikiem `\`. Są to następujące znaki:

- `.` — kropka;
- `^` — „daszek" (caret);

[1] https://docs.python.org/3/library/re.html#re-syntax
[2] https://docs.python.org/3/howto/regex.html
[3] https://pypi.org/project/regex

- $ — znak dolara (dollar sign);
- * — gwiazdka (asterisk);
- + — plus;
- ? — pytajnik (question mark);
- | — kreska pionowa (pipe);
- \ — ukośnik wsteczny (backslash);
- (and) — nawiasy (parentheses, round brackets);
- [and] — nawiasy kwadratowe (square brackets);
- { and } — nawiasy klamrowe, klamry (braces, curly brackets).

To czy jakiś znak z tej listy jest czy nie jest specjalny, zależy czasem od kontekstu, o czym powiemy dalej.

9.1.2 Klasy znaków

Klasy definiowane są jako zbiory znaków ujęte w nawiasach kwadratowych — są użyteczne, gdy nie szukamy jednego konkretnego znaku, ale raczej jakiegokolwiek znaku z pewnego zbioru. Znak myślnika pomiędzy znakami oznacza zakres, znak caret („daszek"), ^, negację, ale tylko na początku, w innym przypadku znak ten oznacza sam siebie. Na przykład:

- `[abc]` — którakolwiek litera z trzech: 'a', 'b' lub 'c';
- `[a-d]` — którakolwiek z czterech liter: 'a', 'b', 'c' lub 'd';
- `[a-cu-z]` — którakolwiek mała litera z zakresów `[a-c]` lub `[u-z]`;
- `[a-zA-Z]` — którakolwiek mała lub duża litera łacińska;
- `[a-zA-Z0-9_]` — którakolwiek litera łacińska, albo cyfra, albo znak podkreślenia;
- `[^0-9]` — jakikolwiek znak, ale *nie* cyfra.

Regeksy nie mogą być ANDowane (jak w Javie), ale mogą być ORowane za pomocą pionowej kreski -: regeks `cat|dog` will look for 'cat' OR 'dog'.

Niektóre metaznaki tracą swoje specjalne znaczenie wewnątrz definicji klasy, na przykład (,), +, *, . (kropka). Wewnątrz klasy oznaczają same siebie. Ale niektóre specjalne symbole, jak `\w` lub `\s` (patrz niżej) mogą być użyte wewnątrz klasy zachowując swoje specjalne znaczenie; na przykład `\w` znaczy *litera, cyfra, lub podkreślnik*, możemy zatem zdefiniować klasę `[\w.,]` i będzie to znaczyć *litera, cyfra, podkreślnik, kropka lub przecinek.*

Znak caret ^ oznacza negację całej klasy, ale tylko jeśli występuje, gdziekolwiek indziej oznacza sam siebie. Tak więc `[^0-9]` znaczy *cokolwiek, ale* not *cyfra*, podczas gdy `[0-9^]` znaczy *cyfra lub caret*. Aby literał] w klasie znaczył sam siebie, należy poprzedzić go znakiem odwróconego ukośnika.

9.1.3 Predefiniowane klasy znaków

Niektóre (niestety nie tak liczne jak w innych językach) klasy znaków są predefiniowane i oznaczane specjalnymi symbolami; zwykle jest to jedna mała litera poprzedzona odwróconym ukośnikiem — w tych przypadkach ta sama ale duża litera oznacza to samo ale zanegowane. Na przykład:

- `\d` — jakakolwiek cyfra, `\D` — nie-cyfra;
- `\s` — jakikolwiek biały znak (spacja, tabulator, znak nowej linii), `\S` — cokolwiek z wyjątkiem białego znaku;

- \w — litera, cyfra lub podkreślnik, \W — cokolwiek, ale nie litera, cyfra lub podkreślnik;
- . (kropka) — jakikolwiek znak z wyjątkiem znaku nowej linii (lub łącznie z nim, jeśli zaznaczona jest opcja DOTALL, patrz rozdz. 9.4.2, str. 184);
- itd.

Uwaga: Ponieważ do reprezentacji napisów Pyton używa UNICODE, *cyfra* znaczy dowolna cyfra, niekoniecznie arabska, a *litera* znaczy dowolna litera, w dowolnym języku.

Niestety, moduł ***re*** w Pytonie nie wspiera kilku ważnych klas predefiniowanych znanych z Javy: \p{L} – dowolna litera, w dowolnym języku (ale *tylko* litera, bez cyfr i podkreślnika), i tego negacja \P{L} (duże 'P'), a również \p{P} – dowolny znak interpunkcyjny, w dowolnym języku, i tego negacja \P{P}.

Można sobie bez nich poradzić definiując własne wersje tych klas:

```
import string
LETTER    = r'[^\W\d_]'       # any letter (no digits, no underscores)
NONLETTER = r'[\W\d_]'        # any non-letter
PUNCT     = '[' + re.escape(string.punctuation) + ']'   # any punct.
NONPUNCT  = '[^' + re.escape(string.punctuation) + ']' # any non-punct.
```

Użyliśmy tu napisu string.punctuation, który zawiera znaki interpunkcyjne

```
>>> import string
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
```

Niektóre z tych znaków są metaznakami, ale chcemy ich użyć dosłownie; funkcja **re.escape** zwraca ten sam napis, ale ze wszystkimi metaznakami traktowanymi dosłownie, więc wewnątrz regeksów reprezentujących same siebie:

```
>>> import re
>>> import string
>>> re.escape(string.punctuation)
'!"\\#\\$%\\&\'\\(\\)\\*\\+,\\-\\./:;<=>\\?@\\[\\\\\\]\\^_`\\{\\|\\}\\~'
```

9.1.4 Lokalizacje specjalne

Te symbole oznaczają nie znaki, ale raczej specjalne pozycje w analizowanym tekście. Na przykład:

- \A — początek tekstu;
- ^ — początek tekstu, ale też linii, jeśli aktywna jest opcja MULTILINE – patrz rozdz. 9.4.1, str. 184;
- \b — granica słowa — przed pierwszą lub za ostatnią literą słowa (słowo to ciąg znaków pasujących do \w); \B — nie na początku lub końcu słowa;
- $ — koniec teksu, albo też linii, jeśli zaznaczona jest opcja MULTILINE;
- \Z — koniec tekstu.

Ważne:
Znak odwróconego ukośnika jest bardzo intensywnie używany w regeksach. Ale dla interpretera Pythona znak ten w literałach napisowych też ma specjalne znaczenia, jako znak modyfikacji (*escape character*). To stwarza problem. Przypuśćmy, że szukamy ciągu trzech znaków `'a\b'`. Silnik wyrażeń regularnych musi zobaczyć podwójny odwrócony ukośnik, żeby potraktować to jako *jeden* ukośnik, a nie jako znak modyfikujący. Czyli w naszym literale napisowym musimy mieć dwa ukośniki. Ale aby umieścić w literale jeden odwrócony ukośnik, musimy napisać dwa, bo inaczej byłby on zrozumiany jako znak modyfikacji przez interpreter Pytona. Tak więc dochodzimy do `'a\\\\b'`. Aby uniknąć tak monstrualnych wyrażeń, przy definiowaniu regeksów używamy zatem „surowych" napisów (*raw string*), z literą `r` (lub `R`) przed otwierającym literał napisowy apostrofem lub znakiem cudzysłowu (również w ich potrójnej formie). W takich literałach znak odwróconego ukośnika jest interpretowany dosłownie, jak „zwykły" znak (patrz rozdz. **??**, str. **??**). Na przykład `'\n'` to jeden znak (nowej linii), ale `r'\n'` to dwa znaki: odwrócony ukośnik i litera 'n'.

9.1.5 Kwantyfikatory

Zaraz za elementem regeksu można umieścić kwantyfikator, który określa liczbę powtórzeń danego elementu w tekście. Na przykład:

- `+` — raz lub więcej;
- `*` — zero razy lub więcej;
- `?` — zero lub jeden raz;
- `{n,m}` — liczba powtórzeń w zakresie $[n, m]$;
- `{n,}` — co najmniej `n` powtórzeń;
- `{,m}` — co najwyżej `m` powtórzeń (być może zero).

Wszystkie te kwantyfikatory są domyślnie **zachłanne** (ang. *greedy*). Znaczy to, że dopasowany zostanie możliwie *najdłuższy* podciąg znaków. Na przykład, jeśli regeksem jest `a.*z` a przeszukiwanym tekstem `"abzczdz"`, dopasowany zostanie cały tekst, choć podnapisy `"abz"` i `"abzcz"` pasowałyby również, ale są krótsze.
Jeśli to nie jest to czego chcemy, możemy uczynić regeks **wstrzemięźliwym** („leniwym", ang. *reluctant*), czyli wyszukującym *najkrótszy* pasujący podciąg — w powyższym przykładzie byłby to podciąg `"abz"`. Aby kwantyfikator był wstrzemięźliwy, należy zaraz za nim umieścić znak zapytania (?). Kontynuując przykład, regeks `a.*?z` znalazłby podciąg najkrótszy (`"abz"`), podczas gdy regeks `a.*z` – najdłuższy (`"abzczdz"`).

Pyton nie obsługuje kwantyfikatora *possessive* (zwany czasem po polsku „zaborczym"), który jest w Javie, choć rzadko używany.

9.1.6 Zamienniki korzystających z regeksów metod klasy *String* z Javy

W Javie kilka bardzo użytecznych metod klasy **String** korzysta z regeksów: **split**, **replaceAll** i **matches**. W Pytonie można zbliżoną funkcjonalność uzyskać korzystając z funkcji pakietu ***re***.

Załóżmy na przykład, że chcemy rozdzielić tekst na słowa (sekwencje liter, ale tylko liter – bez podkreślników czy cyfr. Możemy wtedy założyć, że separatorami między słowami są niepuste sekwencje nie-liter. W Javie użylibyśmy metody **split** z klasy **String** podając jako separator regeks `\P{L}+`; w Pytonie użyjemy funkcji **split**

z pakietu *re*. Nie ma w Pytonie klasy `\P{L}`, ale, jak już wiemy, można ją zastąpić regeksem `r'[\W\d_]'`:

```
1   import re
2
3   NONLETTERs = r'[\W\d_]+' # any non-letter
4
5   s = ",,, __ 123 Łódź - 0.7; London - 8.8; Tokyo - 13.6"
6
7   ls = re.split(NONLETTERs, s)  # sequence of non-letters
8   print(ls)
9   ls = [i for i in re.split(NONLETTERs, s) if i]
10  print(ls)
```

drukuje

```
['', 'Łódź', 'London', 'Tokyo', '']
['Łódź', 'London', 'Tokyo']
```

W przykładowym tekście wystąpiły tu separatory (ciągi nie-liter) zarówno przed pierwszym jak i za ostatnim słowem. W rezultacie otrzymaliśmy puste napisy jako pierwszy i ostatni element wynikowej listy. Linia 9 pokazuje, jak możemy się ich pozbyć.[1]

Regeksy mogą też być ORowane, jak pokazuje następny przykład: tu separator jest zdefiniowany jako niepusta sekwencja białych znaków LUB jako znak interpunkcyjny otoczony być może, ale niekoniecznie, białymi znakami. Sam tekst jest czytany z pliku linia po linii:

Listing 38 REA-Split/Splitting.py

```
1  #!/usr/bin/env python
2
3  import re
4  import string
5
6  pP = '[' + re.escape(string.punctuation) + ']'
7
8  sep = re.compile(r'\s*' + pP + '\s*|\s+')
9  with open('splitting1.txt', 'r') as f:
10     for line in (line.rstrip() for line in f):
11         words = sep.split(line)
12         print('|' + "|".join(words) + '|')
```

Jeśli plik zawiera

```
John,Mary    Charles    ;    Zoe
carrot    parsley:potato
```

to program drukuje

```
|John|Mary|Charles|Zoe|
|carrot|parsley|potato|
```

[1] W Javie pierwszy pusty element też by się pojawił, ale ostatni nie; niestety konwencje są różne w różnych językach...

(pionowe kreski zostały dodane, aby widzieć gdzie są granice słów).

Druga metoda Javy, **String.replaceAll**, pobiera regeks i napis zastępujący, i zwraca napis w którym wszystkie podnapisy pasujące do regeksu są zastąpione napisem zastępującym. W Pytonie podobną funkcjonalność można uzyskać za pomocą funkcji **sub** (od *sub*stitute); pobiera ona regeks, napis zastępujący i tekst:

```
import re

string = "cat, caterpillar, tomcat, cat"
print( re.sub(r'\bcat\b', 'dog', string) )
```

drukuje `dog, caterpillar, tomcat, dog`, gdyż zastępujemy wystąpienia `cat` słowem `dog`, ale tylko jeśli `cat` jest z obu stron otoczone granicami słów, czyli gdy jest to osobne słowo, a nie podnapis słowa: zauważmy, że nie było zastąpienia w słowie `tomcat` bo przed literą 'c' nie ma granicy słowa i podobnie za literą 't' w słowie `caterpillar`.

Trzecią metodą z Javy, którą chcielibyśmy zastąpić odpowiednią konstrukcją Pytona jest **String.matches**. Pobiera ona regeks i odpowiada na pytanie 'czy cały tekst pasuje do tego regeksu'. W Pytonie możemy do tego celu skorzystać z funkcji **fullmatch** z pakietu ***re***: zwraca ona tak zwany **match object**, który w kontekście logicznym jest "truthy" jeśli cały tekst pasuje do regeksu, a **None** (które jest „falsy") w przeciwnym przypadku:

```
import re

pLseq = r'[^\W\d_]+'        # sequence of letters

string = 'Madagascar!'
print('True' if re.fullmatch(pLseq, string) else 'False')
print('True' if re.fullmatch(pLseq+'.', string) else 'False')
```

drukuje

```
False
True
```

W pierwszym przypadku cały tekst musi być sekwencją liter, ale nie jest; w drugim dopuszczamy dowolny znak na końcu, co pasuje do końcowego wykrzyknika.

9.2 Obiekty Pattern

Regeksy, przed użyciem, muszą być skompilowane. Służy do tego funkcja **compile**, która zwraca obiekt klasy **Pattern**, tak zwany **pattern object**, reprezentujący skompilowaną formę regeksu:

```
>>> import re
>>> reg = r'[^\W\d_]+'      # sequence of letters
>>> pat = re.compile(reg)
>>> type(pat)
<class 're.Pattern'>
>>> str(pat)
"re.compile('[^\\\\W\\\\d_]+')"
```

Jak widzimy, metoda **__str__** klasy **Pattern** daje oryginalny regeks, choć w formie „normalnego”, nie surowego, napisu.

Skompilowany regeks jest w zasadzie funkcją (na ogół bardzo skomplikowaną!), która pobiera tekst i zwraca obiekt reprezentujący wynik zastosowania regeksu do tego konkretnego tekstu: może to być **None**, lista, krotka, albo obiekt typu **Matcher**, tak zwany **match object**, który z kolei można „odpytywać” aby uzyskać różne informacje.

W dotychczasowych przykładach nie kompilowaliśmy jawnie regeksów. Tym niemniej były one kompilowane „pod spodem” i wynikowe obiekty **Pattern** były użyte. To jest akceptowalne, jeśli dany regeks jest użyty raz lub dwa. Jeśli jednak ma być używany wielokrotnie, lepiej jest skompilować go raz i potem używać już skompilowanej formy dla różnych tekstów, bo proces kompilacji jest skomplikowany i czasochłonny.[1]

Wiele funkcji z modułu ***re*** ma dwie formy: jako funkcje

```
re.fun(regeks, tekst, ...
```

i jako metody klasy **Pattern** wywoływane na rzecz skompilowanego regeksu (*pattern object*), powiedzmy `pat`

```
pat.fun(tekst, ...)
```

(przykłady za chwilę).

9.3 Grupy przechwytujące

Grupy (ang. *capturing groups*) pozwalają pamiętać podciąg tekstu pasujący do danego fragmentu regeksu i wykorzystać później tę informację. Grupy tworzymy poprzez ujęcie fragmentu regeksu w nawiasy okrągłe; niektóre symbole za nawiasem otwierającym zmieniają interpretację, o czym powiemy później.

9.3.1 „Zwykłe” grupy — (...)

Rozważmy następujący przykład:

Listing 39 REB-Groups/Groups.py

```
#!/usr/bin/env python

import re

LETTER    = r'[^\W\d_]' # any letter (no digits, no underscores)
NONLETTER = r'[\W\d_]'  # any non-letter

reg = f'({LETTER}+){NONLETTER}+({LETTER}+)'
#        <group 1>              <group 2>
print('reg: ' + str(reg))

string = 'John <_0_> Mary'
m = re.fullmatch(reg, string)

print('type of m: ' + str(type(m)))
```

[1] Interpreter Pytona zapamiętuje jednak pewną liczbę ostatnio skompilowanych regeksów i może je wykorzystać bez rekompilacji.

```
16 print('m: ' + str(m))
17
18 print('m.group():        ' + m.group())
19 print('m.group(0):       ' + m.group(0))
20 print('len(m.groups()): ' + str(len(m.groups())))
21 print('m.lastindex:      ' + str(m.lastindex))
22 print('m.groups():       ' + str(m.groups()))
23 print('m.group(1):       ' + m.group(1))
24 print('m.group(2):       ' + m.group(2))
25 print('m.start(2):       ' + str(m.start(2)))
26 print('m.end(2):         ' + str(m.end(2)))
27 print('m.span(2):        ' + str(m.span(2)))
```

Regeks zdefiniowany w linii 8 znaczy *dwa słowa oddzielone niepustą sekwencją nie-liter*. Zauważmy, że części regeksu odpowiadające słowom (ale nie sekwencji oddzielającej) są wzięte w nawiasy. Są dwie pary nawiasów i cokolwiek zostanie dopasowane do znajdujących się w nich regeksów, będzie zapamiętane jako grupa numer 1 i grupa numer 2.

W lonii 13 wywołujemy funkcję **fullmatch**, którą już znamy: zwraca ona obiekt typu *match object* (*matcher*) jeśli cały tekst pasuje do regeksu albo **None** w przeciwnym przypadku. Nasz tekst (linia 12) pasuje do regeksu, więc zmienna m będzie obiektem *match object*.

The program prints

```
reg: ([^\W\d_]+)[\W\d_]+([^\W\d_]+)
type of m: <class 're.Match'>
m: <re.Match object; span=(0, 15), match='John <_0_> Mary'>
m.group():        John <_0_> Mary
m.group(0):       John <_0_> Mary
len(m.groups()): 2
m.lastindex:      2
m.groups():       ('John', 'Mary')
m.group(1):       John
m.group(2):       Mary
m.start(2):       11
m.end(2):         15
m.span(2):        (11, 15)
```

co ilustruje metody obiektu matcher:

- **group()** i równoważnie **group(0)** zwracają całe dopasowanie;
- **groups()** zwraca krotkę zawartości wszystkich grup; w naszym przypadku dwie grupy z sekwencjami liter;
- **group(n)** zwraca zawartość n-tej grupy (pierwsza ma indeks 1, indeks 0 oznacza całe dopasowanie);
- **start(n)** zwraca pozycję (indeks) pierwszego znaku n-tej grupy w całym tekście;
- **end(n)** zwraca pozycję (indeks) pierwszego znaku za ostatnim znakiem n-tej grupy;
- **span(n)** zwraca start i end jako krotkę;

- lastindex jest indeksem grupy, która była zamknięta jako ostatnia — nie jest to równoważne liczbie grup, bo grupy mogą być zagnieżdżane (patrz następny przykład); to jest atrybut obiektu matcher, *nie* metoda.

Grupy mogą być zagnieżdżane i są numerowane od 1, bo grupa 0 to cały dopasowany podnapis. Numery (indeksy) są przypisywane grupom według kolejności w jakiej napotykane są ich nawiasy otwierające — każda grupa sięga aż do odpowiadającego nawiasu zamykającego. Na przykład w poniższym programie

Listing 40 REC-Nesting/Nesting.py

```
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

#            1                    2    3       opening
regex = r'\s*(' + LETTERs + r').*?(\d+-(\d+))'
#                              1            32  closing

print('regex: ' + str(regex))

string = "   Einstein Albert, 1879-1955"
m = re.fullmatch(regex, string)

print('m.group():        ' + m.group())
print('m.lastindex:      ' + str(m.lastindex))
print('len(m.groups()): ' + str(len(m.groups())))
print('m.groups():       ' + str(m.groups()))
print('m.group(1):       ' + m.group(1))
print('m.group(2):       ' + m.group(2))
print('m.group(3):       ' + m.group(3))
```

mamy trzy grupy:

- jedną zawierającą pierwsze tylko słowo po, być może, sekwencji wiodących białych znaków a przed sekwencją dowolnych znaków (które nie wejdą jednak do żadnej grupy);
- następnie grupę zawierającą dwie sekwencje cyfr oddzielone znakiem myślnika;
- oraz zagnieżdżoną w drugiej grupę składającą się tylko z drugiej sekwencji cyfr.

Zauważmy, że grupa *zamknięta* jako ostatnia to grupa 2, nie 3 — tak więc, lastindex jest tu 2 i nie jest równe liczbie grup.

Zwróćmy też uwagę na to, że użyliśmy tu wstrzemięźliwego regeksu `.*?` — bez znaku zapytania zachłanne `.*` skonsumowałoby pierwsze trzy cyfry pierwszej liczby, pozostawiając jedynie czwartą dla pierwszego `\d+`.
Program drukuje

```
regex: \s*([^\W\d_]+).*?(\d+-(\d+))
m.group():          Einstein Albert, 1879-1955
```

```
m.lastindex:     2
len(m.groups()): 3
m.groups():      ('Einstein', '1879-1955', '1955')
m.group(1):      Einstein
m.group(2):      1879-1955
m.group(3):      1955
```

9.3.2 Grupy nieprzechwytujące — (?: ...)

Czasami musimy użyć nawiasów, na przykład by zastosować ORowanie albo kwantyfikator do części regeksu, ale nie chcemy traktować tego jako grupy. Wtedy, za otwierającym nawiasem okrągłym stawiamy pytajnik i dwukropek: `(?: ... )`. Na przykład w programie

Listing 41 RED-NoGroup/NoGroup.py

```
#!/usr/bin/env python

import re

LETTERs    = r'[^\W\d_]+' # sequence of letters

regex = r'(?:Mr\.|Mrs\.)\s+' + LETTERs + f'\\s+({LETTERs})'
#           not group                          group

print('regex: ' + str(regex))

string1 = 'Mr. Adam  Smith'
string2 = 'Mrs. Jane Gordon'

m1 = re.fullmatch(regex, string1)
print('m1.groups(): ', m1.groups())

m2 = re.fullmatch(regex, string2)
print('m2.groups(): ', m2.groups())
```

pierwsza para nawiasów nie tworzy grupy, jest użyta tylko do ORowania dwóch opcji: `Mr.` albo `Mrs.`. Pierwsze imię musi wystąpić, ale nie jest wyłapywane do żadnej grupy; wyłapujemy do grupy tylko nazwisko, a zatem będzie tu tylko jedna grupa:

```
regex: (?:Mr\.|Mrs\.)\s+[^\W\d_]+\s+([^\W\d_]+)
m1.groups():  ('Smith',)
m2.groups():  ('Gordon',)
```

9.3.3 Grupy nazwane — (?P<name> ...)

Zamiast numerować grupy, możemy im nadawać nazwy. Składnia jest następująca: `(?P<name> ... )`, gdzie `name` jest dowolnym identyfikatorem (nawiasy kątowe są tu konieczne). Odwołujemy się do takich grup po nadanej nazwie, `matcher.group('name')` a nie po indeksie (choć równolegle możemy używać i indeksów). Oba sposoby odwoływania się do grup, po indeksie i po nazwie, zilustrowane są w poniższym programie

Listing 42 REE-NamedGroups/NamedGroups.py

```
1  #!/usr/bin/env python
2
3  import re
4
5  regex1 = r'(\d{1,2})-(\d{1,2})-(\d{4}|\d{2})';
6  regex2 = r'(?P<D>\d{1,2})-(?P<M>\d{1,2})-' \
7                       r'(?P<Y>\d{4}|\d{2})'
8
9  print('regex1: ' + str(regex1))
10 print('regex2: ' + str(regex2))
11
12 string = '12-07-21 x_x 6-6-2010 123 y! 1-11-2011'
13
14 matchiter = re.finditer(regex1, string)
15 print('type(matchiter): ' + str(type(matchiter)))
16 print('By indices:')
17 for matcher in matchiter:
18     print(f'{matcher.group(1)}/{matcher.group(2)}/'
19           f'{matcher.group(3)}', end='  ')
20
21 matchiter = re.finditer(regex2, string)
22 print('\nBy names:')
23 for matcher in matchiter:
24     print(f'{matcher.group("D")}/{matcher.group("M")}/'
25           f'{matcher.group("Y")}', end='  ')
26 print()
```

który drukuje

```
regex1: (\d{1,2})-(\d{1,2})-(\d{4}|\d{2})
regex2: (?P<D>\d{1,2})-(?P<M>\d{1,2})-(?P<Y>\d{4}|\d{2})
type(matchiter): <class 'callable_iterator'>
By indices:
12/07/21  6/6/2010  1/11/2011
By names:
12/07/21  6/6/2010  1/11/2011
```

Oba regeksy szukają dat, czyli trzech liczb oddzielonych myślnikiem: jedno- lub dwucyfrowej (dzień), jedno- lub dwucyfrowej (miesiąc) i dwu- lub czterocyfrowej (rok). Zauważmy, że w ostatnim przypadku musieliśmy użyć `(\d{4}|\d{2})`, a nie odwrotnie (linie 5 i 7). Gdybyśmy użyli `(\d{2}|\d{4})` pierwsza alternatywa, z dwoma cyframi, pasowałaby do pierwszych dwóch cyfr czterocyfrowego roku, a `\d{4}` nie byłoby w ogóle wzięte pod uwagę.

W pierwszym regeksie (linia 5) nie używamy nazw, więc odnosić się do grup możemy tylko po indeksach (linie 18 i 19). W drugim (linie 6 i 7) nadaliśmy grupom nazwy ‘D’, ‘M’ i ‘Y’ więc możemy się za ich pomocą odnosić później do poszczególnych grup w liniach 24-25 (choć indeksy też by zadziałały).

Funkcja **finditer** użyta w liniach 14 i 21 zwraca iterator, iterując po którym otrzymujemy obiekty matcher dla każdego podnapisu pasującego do regeksu (patrz rozdz. 9.5.7,

str. 191)

9.3.4 Wewnętrzne odwołania wsteczne

Do grup możemy się też odwoływać w samym regeksie. Wyrażenie `\k` odwołuje się do wcześniej znalezionej grupy o indeksie k (nazywa się to odwołaniem wstecznym, ang. *backreferencing*). Można też zamiast `\k` użyć wyrażenia `\g<k>` — to może być konieczne, jeśli w regeksie mamy cyfrę zaraz za odwołaniem wstecznym. Jeśli grupa do której chcemy się odwołać ma nazwę, można też użyć formy `(?P=name)`.
W poniższym przykładzie chcemy znaleźć fragmenty umieszczone w apostrofach albo cudzysłowach, ale musimy zadbać o to, aby znak zamykający był tego samego rodzaju co otwierający:

Listing 43 REF-BackRefs/BackRefs.py

```
#!/usr/bin/env python

import re

regex1 = r'(["\'])([^"\']*)\1';
regex2 = r'(?P<quote>["\'])([^"\']*)(?P=quote)'

print('regex1: ' + str(regex1))
print('regex2: ' + str(regex2))

string = "'abc' xx \"def\" yy \"ghi' zz"

matchiter = re.finditer(regex1, string)
print('By indices:')
for matcher in matchiter:
    print(f'{matcher.group(1)}{matcher.group(2)}'
          f'{matcher.group(1)}', end='  ')

matchiter = re.finditer(regex2, string)
print('\nBy name or index:')
for matcher in matchiter:
    print(f'{matcher.group("quote")}{matcher.group(2)}'
          f'{matcher.group(1)}', end='  ')
print()
```

Regeks zawiera grupę jednoznakową – apostrof albo cudzysłów – za którym następuje grupa jakichkolwiek znaków *nie* będących apostrofem lub cudzysłowem, za którą z kolei stoi *taki sam* znak jak w jednoznakowej pierwszej grupie (czyli apostrof lub cudzysłów). Program drukuje

```
regex1: (["\'])([^"\']*)\1
regex2: (?P<quote>["\'])([^"\']*)(?P=quote)
By indices:
'abc'  "def"
By name or index:
'abc'  "def"
```

Zauważmy, że sekwencja `"ghi'` jest otoczona niepasującymi do siebie znakami, więc, prawidłowo, nie została znaleziona.

9.3.5 Zewnętrzne odwołania wsteczne

Odwołania wsteczne mogą też być używane nie wewnątrz samego regeksu, ale też w tekstach zastępujących (ang. *replacement text*) w funkcji **sub**, którą już spotkaliśmy (str. 171). W tekście zastępującym możemy się odwołać do zawartości dopasowanych grup poprzez indeks grupy – `\k` lub `\g<k>`, albo, jeśli grupa była nazwana, poprzez nazwę `\g<name>`.

W poniższym przykładzie, dla ilustracji, nie używamy funkcji **re.sub**, jak poprzednio, ale metody **sub** klasy **Pattern** (patrz rozdz. 9.2, str. 171):

Listing 44 REG-Replace/Replace.py

```
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

regex = f'(?P<first>{LETTERs})\\s+(?P<last>{LETTERs})'
print('regex: ' + str(regex))

patt = re.compile(regex)

strold = "John  Smith, Mary   Brown"

strnew = patt.sub(r'\2 \1', strold)
print(strold + '  ==>>  ' + strnew)
  # or
strnew = patt.sub(r'\g<2> \g<1>', strold)
print(strold + '  ==>>  ' + strnew)
  # or
strnew = patt.sub(r'\g<last> \g<first>', strold)
print(strold + '  ==>>  ' + strnew)
  # or mixed
strnew = patt.sub(r'\g<last> \1', strold)
print(strold + '  ==>>  ' + strnew)
```

Regeks znaczy: dwie sekwencje liter oddzielone niepustą sekwencją sekwencją białych znaków. Sekwencje liter wychwytywane są do nazwanych grup *first* i *last*. Następnie zamieniamy znalezione dopasowania do regeksu zawartością tych właśnie grup, ale w odwróconym porządku i oddzielonych dokładnie jedną spacją. Robimy to na cztery sposoby, używając indeksów lub nazw (z tym samym wynikiem):

```
regex: (?P<first>[^\W\d_]+)\s+(?P<last>[^\W\d_]+)
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
John  Smith, Mary   Brown  ==>>  Smith John, Brown Mary
```

9.3.6 Asercje wsteczne i w przód — (?= ...), (?! ...), (?<= ...), (?<! ...)

Czasem chcemy znaleźć podnapisy pasujące do regeksu, ale tylko w pewnym kontekście: na przykład tylko jeśli przed lub za znalezionym fragmentem znajduje się fragment pasujący do innego regeksu (asercja pozytywna wsteczna lub w przód) lub, wręcz przeciwnie, tylko jeśli przed lub za znalezionym fragmentem *nie* występuje ciąg pasujący do innego regeksu (asercja negatywna wsteczna lub w przód). składnia jest następująca

- (?= ...) — pozytywna w przód;
- (?<= ...) — pozytywna wsteczna;
- (?! ...) — negatywna w przód;
- (?<! ...) — negatywna wsteczna.

Same asercje nie będą częścią dopasowanego tekstu, służą tylko do określenia kontekstu, w jakim dopasowywany tekst ma się pojawić.

W przykładzie poniżej używamy funkcji **finditer** zwracającej iterator, który przy iteracji daje obiekty matcher dla każdego dopasowanego fragmentu tekstu (patrz rozdz. 9.5.7, str. 191).

Listing 45 REH-Look/Look.py

```
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

string = 'carrot 5.7, butter 6.5, carrot 5 milk ' \
         'carrot 7.3  parsley 3.5 carrot 2.00  ' \
         'butter 2 carrot apples'
# -- 1 --
# sum of prices only of carrot with a price
regex = r'(?<=carrot) (\d+(?:\.\d+)?)'
matchiter = re.finditer(regex, string)
summ = 0
for matcher in matchiter: summ += float(matcher.group(1))
print('carrot:               ' + str(summ))

# -- 2 --
# sum of prices of all items with a price, except carrot
regex = r'(?<!carrot) (\d+(?:\.\d+)?)'
matchiter = re.finditer(regex, string)
summ = 0
for matcher in matchiter: summ += float(matcher.group(1))
print('all but carrot:       ' + str(summ))

# -- 3 --
# set of items with price indicated
regex = f'({LETTERs})\\b(?=(?: \\d+))'
matchiter = re.finditer(regex, string)
sett = set()
for matcher in matchiter: sett.add(matcher.group(1))
```

```
32 print('items with price:     ' + str(sett))
33
34 # -- 4 --
35 # set of items with price not indicated
36 regex = f'({LETTERs})\\b(?!(?: \\d+))'
37 matchiter = re.finditer(regex, string)
38 sett = set()
39 for matcher in matchiter: sett.add(matcher.group(1))
40 print('items with no price: ' + str(sett))
```

Spójrzmy na pierwszy przykład. Jest tu jedna grupa (linia 12): `(\d+(?:\.\d+)?)`. Znaczy ona *sekwencja cyfr za którą, opcjonalnie, może występować kropka jeszcze jedna sekwencja cyftr*. Ta opcjonalna część nie tworzy osobnej grupy, bo jest na początku `?:` — nawiasy tylko mówią czego dotyczy znak '?' na końcu. Przed grupą mamy spację, a przed tą spacją musi występować słowo `carrot`. Ono samo *nie* będzie częścią dopasowania, ale wskazuje pozytywną asercję wsteczną. Innymi słowami szukamy liczb, ale tylko za słowem `carrot`.

Drugi przykład jest podobny, ale tym razem mamy negatywną asercję wsteczną — szukamy zatem liczb, ale tylko jeśli *nie* są poprzedzone słowem `carrot`.
Trzeci i czwarty przykład ilustrują asercje w przód: tworzymy zbiór nazw produktów za którymi występuje lub właśnie nie występuje cena.

Program drukuje

```
carrot:              20.0
all but carrot:      12.0
items with price:    {'butter', 'parsley', 'carrot'}
items with no price: {'milk', 'apples', 'carrot'}
```

I ostatni przykład: poniższy program używa asercji w przód, aby sprawdzić, czy dane słowo może być prawidłowym hasłem. Zakładamy tu, że hasło powinno mieć przynajmniej 8 znaków, co najmniej jedną dużą literę, co najmniej jedną małą, co najmniej jedną cyfrę i co najmniej jeden znak specjalny (w programie uważamy za takie znaki napisu PC):

Listing 46 RER-PassCheck/PassCheck.py

```
1  #!/usr/bin/env python
2
3  import re
4
5  def checkPass(password):
6      PC =r'!@#$%^&*><?.,;:' # special characters
7      p1 = (fr'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d)'
8            fr'(?=.*[{PC}])[A-Za-z\d{PC}]{{8,}}$')
9        # or, equivalently with verbose mode on
10     p2 = fr'''(?x)^              # verbose; start of string
11               (?=.*[A-Z])       # capital letter somewhere
12               (?=.*[a-z])       # lower-case letter somewhere
13               (?=.*\d)          # digit somewhere
14               (?=.*[{PC}])      # special character somewhere
```

```
            [A-Za-z\d{PC}]{{8,}}$  # at least 8 chars, EOS
        '''
    return re.match(p2, password)

for pas in [r'Ab?De93',  r'A1b:A1b>',  r'Ab/Acb<1',
                r'abc123><', r'Zorro@123', r'ab<AB!1z']:
    print(f'{pas:9} is {"" if checkPass(pas) else "not "}valid')
```

Regeksy p1 i p2 są równoważne; w drugim z nich użyliśmy trybu *verbose* (zob. rozdz. 9.4.5, str. 186) aby jego znaczenie stało się łatwiejsze do uchwycenia. Program drukuje

```
Ab?De93   is not valid      # too short
A1b:A1b>  is valid
Ab/Acb<1  is not valid      # / is not among special chars
abc123><  is not valid      # no capital letter
Zorro@123 is valid
ab<AB!1z  is valid
```

(zauważmy podwójne klamry w {{8,}}; w f-stringach klamry mają specjalne znaczenie i aby dostać jedną dosłowną klamrę, należy ją podwoić).

9.3.7 Regeksy yes-no

Składnia tej rzadko używanej konstrukcji jest

```
(?(id/name)pattern_if_yes|pattern_if_no)
```

Szukamy tu podnapisów pasujących albo do pattern_if_yes albo do pattern_if_no w zależności of tego czy grupa do której odnosi się id (1, 2, ...) lub name dla grup nazwanych została dopasowana czy nie. Na przykład,

Listing 47 REI-YesNo/YesNo.py

```
#!/usr/bin/env python

import re

LETTERs = r'[^\W\d_]+' # sequence of letters

lines = ['#bananas',
         '255',
         '#tomatos',
         '4',
         '#apples',
         '30']

regex = f'(#)?((?(1){LETTERs}|\\d+))'
patt = re.compile(regex)
for line in lines:
    print( str( patt.fullmatch(line).group(2) ) )
```

będzie próbowało znaleźć dopasowanie albo dla słowa albo dla liczby, w zależności od

tego, czy na początku był znak '#' czy nie.

```
bananas
255
tomatos
4
apples
30
```

9.3.8 Comments — (?# ...)

May też w końcu grupy, które nie tylko nie są grupami, (jak te `(?: ...)`, ale są całkowicie ignorowane przez silnik wyrażeń regularnych i służą jedynie jako komentarze. Na przykład:

```
import re
LETTERs = r'[^\W\d_]+' # sequence of letters

regex = f'{LETTERs}(?# letters and then)\\.(?# a dot)'
string = 'Mississippi.'
m = re.fullmatch(regex, string)
if m: print(m.group())
```

drukuje `Mississippi.`.

9.4 Flagi opcji

Jest kilka opcji, które wpływają na proces kompilacji regeksu. Mogą one być wyspecyfikowane na dwa sposoby:

- jako dodatkowy argument dla funkcji **re.compile** lub innej funkcji z modułu *re* pobierającej nieskompilowany regeks (w postaci napisu) jako pierwszy argument — jak wiemy kompilacja i tak zajdzie „pod spodem"; lub
- mogą być ustawione bezpośrednio wewnątrz napisu reprezentującego regeks (opcje wewnętrzne).

Opcje są zdefiniowane w module *re* jako stałe typu wyliczeniowego **RegexFlag** (który jest podtypem **enum.IntFlag**) i mogą być ORowane za pomocą kreski pionowej `|` jeśli chcemy uaktywnić kilka z nich.

Jeśli chcemy użyć opcji wewnętrznych, robimy to poprzez włącznie do regeksu wyrażenia `(?letters)`, gdzie `letters` to jedna lub więcej liter odpowiadających różnym opcjom.

Na przykład przypuśćmy, że chcemy włączyć opcje kompilacji DOTALL i MULTILINE; możemy to zrobić albo tak

```
>>> import re
>>> regex = "..."
>>> regObj = re.compile(regex, re.MULTILINE | re.DOTALL)
```

albo tak

```
>>> import re
>>> regex = "..."
>>> regObj = re.compile(regex, re.M | re.S)
```

bo re.MULTILINE jest równoważne re.M a re.DOTALL jest równoważne re.S.

Ale te same opcje możemy ustawić włączając `(?letters)` bezpośrednio do regeksu

```
>>> import re
>>> regex = "(?sm)..."
>>> regObj = re.compile(regex)
```

bo litera 's' oznacza DOTALL ('s', gdyż w Perlu ta opcja nazywała się 'single-line') a 'm' oznacza MULTILINE.

Zauważmy, że włączanie opcji powoduje często pogorszenie wydajności kompilacji, więc nie należy tego robić bez potrzeby.

Jako że flagi są enumeratorami, odpowiadają **int**om i mogą być ORowane. Program poniżej drukuje wszystkie flagi, ich pełną formę, formę skróconą, odpowiadające im litery, które można wstawić jako opcje wewnętrzne, oraz liczby całkowite jakim odpowiadają:

Listing 48 REK-RegFlags/RegFlags.py

```
#!/usr/bin/env python

import re

regex  = r'\d+'
patObj = re.compile(regex, flags=0)
print('Default flags: ' + str(patObj.flags) + '\n')
print('Type of flags: ' + str(type(re.ASCII)) + '\n')

print(f'IGNORECASE I (?i) -> {int(re.I):3d}')
print(f'LOCALE     L (?L) -> {int(re.L):3d}')
print(f'MULTILINE  M (?m) -> {int(re.M):3d}')
print(f'DOTALL     S (?s) -> {int(re.S):3d}')
print(f'UNICODE    U (?u) -> {int(re.U):3d}')
print(f'VERBOSE    X (?x) -> {int(re.X):3d}')
print(f'DEBUG             -> {int(re.DEBUG):3d}')
print(f'ASCII      A (?a) -> {int(re.A):3d}')
```

Program drukuje

```
Default flags: 32

Type of flags: <enum 'RegexFlag'>

IGNORECASE I (?i) ->   2
LOCALE     L (?L) ->   4
MULTILINE  M (?m) ->   8
DOTALL     S (?s) ->  16
UNICODE    U (?u) ->  32
VERBOSE    X (?x) ->  64
DEBUG             -> 128
ASCII      A (?a) -> 256
```

Opcja UNICODE nie jest używana, bo Unicode jest domyślnie ustawiony — jest wciąż obecna ze względu na zgodność ze starszymi wersjami Pytona. Opcja DEBUG, jeśli jest ustawiona, jest używana do debugowania — program będzie drukował szczegółową informację o procesie kompilowania regeksu; informacja ta jest przeznaczona raczej dla ekspertów i nie będziemy się nią zajmować. Również LOCALE jest używana rzadko, bo może być zastosowana tylko do regeksów zadanych w postaci ciągu bajtów a nie napisów.

Pozostałe flagi są opisane poniżej.

9.4.1 *MULTILINE*

Forma skrócona opcji re.MULTILINE to re.M. W formie wewnętrznej opowiada jej litera 'm'.

Opcja ta uaktywnia tryb wieloliniowy, co oznacza, że metaznaki ^ and $ oznaczają, odpowiednio, również początki lub końce linii, a nie tylko początek i koniec całego tekstu (ale nawet w trybie wieloliniowym początek i koniec całego tekstu są dostępne za pomocą metaznaków \A and \Z).
Na przykład program

Listing 49 REL-FlagM/FlagM.py

```
#!/usr/bin/env python

import re

string = "A 123\nD 456";

matchiter1 = re.finditer(r'^\w',     string)
matchiter2 = re.finditer(r'(?m)^\w', string)

for matcher in matchiter1:
    print(f'{matcher.group()}', end=' ')
print()

for matcher in matchiter2:
    print(f'{matcher.group()}', end=' ')
print()
```

drukuje

```
A
A D
```

bo w pierwszym przypadku szukamy tylko litery na początku całego tekstu, podczas gdy w drugim — również na początku każdej linii z osobna.

9.4.2 *DOTALL*

Forma skrócona opcji re.DOTALL to re.S. W formie wewnętrznej opowiada jej litera 's'.

Włączenie opcji DOTALL zmienia interpretację metaznaku . (kropka). Normalnie znaczy on 'dowolny znak, z wyjątkiem znaku końca linii', ale z włączoną opcją DOTALL znaczy 'dowolny znak, *również* znak końca linii'.

Na przykład, następujący program

```
import re

string = "A 123\n456 B";
print('Matches' if (re.fullmatch(r'\w.*\w', string))
                 else "Doesn't match")
print('Matches' if (re.fullmatch(r'\w.*\w', string, re.S))
                 else "Doesn't match")
```

wydrukuje

```
Doesn't match
Matches
```

bo w pierwszym przypadku `.*` konsumuje wszystko, ale tylko do znaku końca linii, więc nie pasuje do całego tekstu, natomiast w drugim konsumuje wszystko łącznie ze znakiem nowej linii i znajduje na końcu literę 'B'.

9.4.3 *IGNORECASE*

Forma skrócona opcji re.IGNORECASE to re.I. W formie wewnętrznej opowiada jej litera 'i'.

Przy ustawionej opcji duże i odpowiadające im małe litery traktowane są jako nierozróżnialne.

Na przykład następujący program

```
import re

string = "PaRiS";
print('Matches' if (re.fullmatch('Paris', string))
                 else "Doesn't match")
print('Matches' if (re.fullmatch('Paris', string, re.I))
                 else "Doesn't match")
```

wypisze

```
Doesn't match
Matches
```

bo w drugim przypadku ignorowana jest wielkość liter, więc `PaRiS` i `Paris` są uważane za jednakowe.

9.4.4 *ASCII*

Forma skrócona opcji re.ASCII to re.A. W formie wewnętrznej opowiada jej litera 'a'.

Gdy ta opcja jest włączona, regeksy `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` i `\S` traktują jako litery tylko litery łacińskie (ASCII), a nie wszystkie znaki kwalifikowane jako litery w standardzie Unicode. Na przykład `\w` pasuje do podkreślnika, cyfr arabskich (ale nie innych) i liter łacińskich.

Na przykład następujący program

```
import re

string = "Żółć Bałtyk Paris";
regex  = r'[^\W\d_]+' # sequence of letters

matcher1 = re.finditer(regex, string)
matcher2 = re.finditer(regex, string, re.ASCII)

print([m.group() for m in matcher1])
print([m.group() for m in matcher2])
```

drukuje

```
['Żółć', 'Bałtyk', 'Paris']
['Ba', 'tyk', 'Paris']
```

gdyż w drugim przypadku `[^\W\d_]+` pasuje do sekwencji liter łacińskich, a litery nie-ASCII, jak polskie litery 'Ż', 'ó', 'ł', 'ć', *nie* są uważane za litery.

9.4.5 *VERBOSE*

Forma skrócona opcji re.VERBOSE to re.X. W formie wewnętrznej opowiada jej litera 'x'.

Ta opcja jest używana dla wygody. Kiedy jest ustawiona

- białe znaki (w tym znaki końca linii) w regeksach są ignorowane, chyba że występują w klasach znaków lub są poprzedzone wstecznym ukośnikiem;
- znaki po znaku # (nie będącym w definicji klasy znaków i nie poprzedzonym odwróconym ukośnikiem) aż do końca linii są również ignorowane (traktowane jako komentarz).

Pozwala to pisać bardzie czytelne regeksy i opatrywać je komentarzami, jak to jest zilustrowane w programie

Listing 50 REM-Verbose/Verbose.py

```
#!/usr/bin/env python

import re

string = '2/11/2021 xxx 13-01/1999 yyy 13-10 zzz 21.2.2022'

regex1 = r'\d{1,2}([-/.])\d{1,2}\1\d{4}'
ls1 = re.finditer(regex1, string)
print([m.group() for m in ls1])

# now the same thing with VERBOSE

regex2 = r'''(?x)\d{1,2} # verbose; day: one or two digits
             ([-/.])     # day-month separator: - or / or .
             \d{1,2}     # month: one or two digits
             \1          # the same separator as day-month
```

```
17              \d{4}       # year: four digits
18          '''
19  ls2 = re.finditer(regex2, string)
20  print([m.group() for m in ls2])
```

który drukuje

```
['2/11/2021', '21.2.2022']
['2/11/2021', '21.2.2022']
```

Inny przykład widzieliśmy już wcześniej, zob. Listing 46.

9.5 Funkcje operujące na wyrażeniach regularnych

Wiele funkcji operujących na wyrażeniach regularnych ma dwie formy:

- jako funkcje modułu *re* pobierające regeks w postaci napisu;
- jako metody klasy **re.Pattern**, wywoływane na obiektach reprezentujących już skompilowane regeksy.

W pierwszym przypadku kompilacja może być zmodyfikowana za pomocą flag kompilacji (patrz rozdz. 9.4, str. 182) podawanych jako argumenty wywołania tych funkcji; w drugim przypadku oczywiście nie, bo wywołujemy je na już skompilowanych regeksach.

Takie pary funkcji będziemy poniżej opisywać łącznie: przez **re.function(...)** rozumieć będziemy funkcję z modułu *re*, natomiast **regObj.function(...)** oznaczać będzie metodę wywoływaną na obiekcie regObj typu **Pattern** będącym skompilowaną formą regeksu. Zauważmy, że w dokumentacji napisy definiujące regeksy nazywane są *regex patterns*, a ich forma skompilowana to *regex object*.

Funkcje **search**, **match** i **fullmatch** (a również **finditer**) zwracają obiekty *match objects* (lub po prostu *matchery*) typu **re.Match**. Mają one metody do pobierania informacji o znalezionych dopasowaniach; je również omówimy poniżej, tak jak też kilka przydatnych pól klasy **Pattern**.

9.5.1 re.compile

`re.compile(regex, flags=0)` — kompiluje regeks zadany w postaci napisu i zwraca *regex object:* skompilowaną wersję regeksu, które można potem używać wielokrotnie bez konieczności rekompilacji.

```
>>> regex = '(\d+) ([^\W\d_]+)'
>>> regObj = re.compile(regex, re.I)
>>> type(regObj)
<class 're.Pattern'>
```

Przykłady: ?? (srt. ??), Listing 44 (str. 178), Listing 47 (str. 181) i Listing 48 (str. 183).

9.5.2 re.search

`re.search(regex, string, flags=0)` — wyszukuje pierwszego (i tylko pierwszego) podnapisu string pasującego do regeksu. Zwraca matcher opisujący znaleziony podnapis lub None, jeśli żaden taki fragment nie został znaleziony.

W drugiej wersji, jako metoda `regObj.search(string[,pos[,endpos]])`, funkcja ma dwa dodatkowe, opcjonalne parametry:

- pos jest indeksem w przeszukiwanym napisie od którego należy rozpocząć szukanie (domyślnie jest to 0;
- endpos jest indeksem znaku za ostatnim, który powinien być brany pod uwagę (domyślnie równy długości napisu); tylko fragment od pozycji pos do pozycji endpos-1 (włącznie) będzie przeszukiwany.

Na przykład

```
import re
regex = r'\d+'
string = 'abc 123456789'
regObj = re.compile(regex)
m = regObj.search(string, 0, 7)
print(type(m))
print(m.group())
print(m.span())
```

drukuje

```
<class 're.Match'>
123
(4, 7)
```

bo tylko fragment napisu złożony ze znaków pod indeksami w zakresie $[0, 6]$ (czyli abc 123) jest przeszukiwany. Zmienna m jest tutaj obiektem-matcherem. Użyliśmy tu jego metody **group()**, która zwraca, jako napis, podnapis pasujący do regeksu. Metoda **span** z kolei zwraca krotkę indeksów w całym napisie tego podnapisu; jak zwykle pierwszy należy rozumieć jako *inclusive*, a drugi jako *exclusive*.

9.5.3 re.match

`re.match(regex, string, flags=0)` — zachowuje się jak **search**, ale szuka fragmentu pasującego do regeksu *tylko* na samym początku całego napisu.

Druga weresja funkcji, jako metoda `regObj.match(string[,pos[,endpos]])`, szuka dopasowania *tylko* na początku fragmentu odpowiadającego indeksom pos i pos-1 ignorując część rozpoczynającą się od indeksu endpos. Na przykład

```
import re

regex = r'\d+'
string = 'abc 123456789'

regObj = re.compile(regex)
m = regObj.match(string)
print(m)
```

```
m = regObj.match(string, 6)
print(m.group())
print(m.span())
```

drukuje

```
None
3456789
(6, 13)
```

W pierwszym przypadku dopasowanie nie zostało znalezione, bo choć sekwencja cyfr występuje, to nie od indeksu 0. W drugim przypadku rozpoczynamy od indeksu 6 (czyli od znaku '3') i tam rzeczywiście od razu mamy ciąg cyfr.

9.5.4 re.fullmatch

`re.fullmatch(regex, string, flags=0)` — jak **match**, ale tym razem to *cały* tekst musi pasować do regeksu, od początku do końca, tylko pasujący fragment nie wystarczy. Zwraca obiekt matchera jeśli wynik jest pozytywny, a **None** w przeciwnym przypadku.

Druga wersja funkcji, jako metoda `regObj.fullmatch(string[,pos[,endpos]])`, jeśli podane są dodatkowe argumenty, działa tylko na podnapisie określonym indeksami pos i endpos.

Przykłady można znaleźć w Listing 39 (str. 172), Listing 40 (str. 174), Listing 41 (str. 175) i Listing 47 (str. 181).

9.5.5 re.split

`re.split(regex, string, maxsplit=0, flags=0)` — podobnie jak **String::split** z Javy, rozdziela tekst na „słowa" traktując fragmenty pasujące do regeksu jako separatory; zwraca listę znalezionych „słów". Jeden przykład już pokazywaliśmy, Listing 38 (str. 170).

Uwaga: używanie grup w regeksie definiującym separatory nie jest wskazane. Wyniki tego mogą być nieintuicyjne: fragmenty „wyłapane" przez grupy będą dodane do listy „słów". Na przykład

```
import re

regex = r'(:|\||;)'  #  ':' '|' and ';' are separators
string = ':ab|c;|3:'

lst = re.split(regex, string)
print(lst)
```

drukuje

```
['', ':', 'ab', '|', 'c', ';', '', '|', '3', ':', '']
```

Bez grup

```
import re

regex = r':|\||;'  #  ':' '|' and ';' are separators
string = ':ab|c;|3:'

lst = re.split(regex, string)
print(lst)
```

program drukuje

```
['', 'ab', 'c', '', '3', '']
```

Jak widzimy, jeśli separator występuje na początku lub końcu tekstu, otrzymana lista będzie zawierać na początku i końcu elementy puste, podobnie, jeśli gdzieś w środku występują dwa separatory obok siebie (jak między 'c' and '3').

Dodatkowy argument maxsplit określa maksymalną liczbę „przecięć" — lista będzie zatem zawierać najwyżej maxsplit+1 elementów, przy czym ostatni będzie zawierać już cały pozostały fragment tekstu:

```
import re

regex = r'[\W\d_]+' # sequence of non-letters
string = 'John, Mary Adam,  Kitty Sue   Bill'

lst = re.split(regex, string, maxsplit=3)
print(lst)
```

drukuje

```
['John', 'Mary', 'Adam', 'Kitty Sue   Bill']
```

Jest też druga wersja, jako metoda `regObj.split(string, maxsplit=0)`.

9.5.6 re.findall

`re.findall(regex, string, flags=0)` — zwraca listę wszystkich podnapisów tekstu string pasujących do regeksu. Jednakże, rezultat zależy od liczby grup w regeksie:

- jeśli nie ma grup, dostaniemy listę podnapisów pasujących do regeksu;
- jeśli jest jedna grupa, listę fragmentów przechwyconych do tych grup;
- jeśli jest więcej grup, listę krotek, z których każda będzie zawierać zawartość poszczególnych grup dla kolejnych pasujących podnapisów.

Ilustruje to następujący przykład:

```
import re

string = 'John weight =  75, height= 182  Kitty 23'

# 1
regex = r'[^\W\d_]+\s*=\s*\d+'
print( re.findall(regex, string) )

# 2
regex = r'[^\W\d_]+\s*=\s*(\d+)'
print( re.findall(regex, string) )

# 3
regex = r'([^\W\d_]+)\s*=\s*(\d+)'
print( re.findall(regex, string) )
```

which prints

```
['weight =  75', 'height= 182']
['75', '182']
[('weight', '75'), ('height', '182')]
```

Regeks znaczy *sekwencja liter za którą występuje znak równości, być może otoczony białymi znakami, a potem sekwencja cyfr.*

W pierwszym przypadku nie ma grup, więc dostajemy listę podnapisów dopasowanych do całego regeksu.
W drugim przypadku jest jedna grupa (sekwencja cyfr), więc dostajemy tylko listę dopasowań do tej grupy.
W trzecim przypadku są dwie grupy (sekwencja liter i sekwencja cyfr), więc dostajemy listę krotek zawartości poszczególnych grup dla każdego dopasowania.

W wersji jako metoda `regObj.findall(string[,pos[,endpos]])`, tylko fragment tekstu string odpowiadający indeksom pos i endpos jest brany pod uwagę.

9.5.7 re.finditer

`re.finditer(regex, string, flags=0)` — zwraca iterator, który przy iteracji zwraca matchery dla kolejnych dopasowań do regeksu. Przykłady spotkaliśmy w Listing 42 (str. 176), Listing 43 (str. 177) i Listing 45 (str. 179).
W wersji jako metoda `regObj.finditer(string[,pos[,endpos]])`, tylko fragment tekstu string odpowiadający indeksom pos i endpos jest brany pod uwagę.

9.5.8 re.sub

`re.sub(regex, repl, string, count=0, flags=0)` — przypomina metodę **String::replaceAll** z Javy — zwraca napis otrzymany przez zastąpienie w tekście string wszystkich fragmentów pasujących do regeksu napisem repl. Jeśli żadnych dopasowań nie ma, zwracany jest niezmodyfikowany napis string. Opcjonalny argument count mówi jaka jest maksymalna liczba podstawień; domyślna wartość jest 0, co znaczy *wszystkie*.

W postawiany napisie repl można użyć odwołań wstecznych do grup znalezionych w dopasowaniu: poprzez indeks (np. \2 lub \g<2>) albo, dla grup nazwanych, poprzez nazwę (\g<name>) — zob. rozdz. 9.3.5, str. 178.

Argument repl jest zwykle napisem — być może z odwołaniami wstecznymi — ale może też być funkcją. Jeśli tak jest, ta funkcja będzie wywołana dla każdego znalezionego dopasowania. Argumentem będzie matcher odpowiadający temu dopasowaniu, zwracany powinien być napis, który zostanie użyty jako napis zastępujący dopasowanie w wynikowym napisie.

Przykład widzieliśmy w Listing 44, str. 178. Inny przykład, pokazujący jak użyć funkcji zamiast tekstu zastępującego, podajemy poniżej:

Listing 51 REJ-SubFun/SubFun.py

```
1 #!/usr/bin/env python
2
3 import re
4
5 def classify(matcher):
6     g = matcher.group()
7     num = int(g)
8     if   num > 10: g += ' (too high)\n'
```

```
    elif num <  0: g += ' (too low)\n'
    else:          g += ' (OK)\n'
    return g

regex = r'-?\d+'
strold = '-7 11 5 2 12 9 -1'

strnew = re.sub(regex, classify, strold)
print(strnew)
```

Regeks jest tu prosty: *sekwencja cyfr opcjonalnie poprzedzona znakiem minus*. Funkcja **classify** będzie wywołana dla każdego dopasowanego podnapisu; odpowiedni matcher będzie do niej przesłany jako argument a dopasowany podnapis, ale zmodyfikowany, będzie zwrócony i użyty do właściwego podstawienia:

```
-7 (too low)
11 (too high)
5 (OK)
2 (OK)
12 (too high)
9 (OK)
-1 (too low)
```

Druga wersja funkcji to metoda `regObj.sub(repl, string, count=0)`.

9.5.9 re.subn

`re.sub(regex, repl, string, count=0, flags=0)` — zachowuje się jak **sub**, ale zwraca dwuelementową krotkę, której pierwszym elementem jest napis taki jaki zostałby zwrócony przez **sub**, a drugim liczba wykonanych podstawień. Alternatywnie, możemy użyć metody `patObj.sunb(repl, string, count=0)`.

Na przykład

```
import re

string = 'cat, caterpilar,bulldog - dog, cat'
regex = r'\b(cat|dog)\b'

res = re.subn(regex, 'pet', string)
print(type(res))
print(res)
```

drukuje

```
<class 'tuple'>
('pet, caterpilar,bulldog - pet, pet', 3)
```

9.5.10 re.escape

`re.escape(regex)` — zwraca napis ze wszystkimi metaznakami w regex zmienionym tak, aby mogły ozaczać same siebie: wynikowy napis może zatem być użyty

jako regeks, gdy szukamy podnapisów zawierających dosłowne metaznaki. Przypuśćmy na przykład, że mamy plik w formacie TEX lub LATEX i szukamy w nim napisu `\index{Sherlock}`. Zamiast myśleć czym zastąpić metaznaki (tutaj będą to `\`, `{` i `}`), możemy użyć funkcji **escape**, która zrobi to za nas:

```
>>> import re
>>> regex = r'\index{Sherlock}'
>>> re.escape(regex)
'\\\\index\\{Sherlock\\}'
```

lub

```
>>> import re
>>> operators = r'Operators: *, /, +, -'
>>> re.escape(operators)
'Operators:\\ \\*,\\ /,\\ \\+,\\ \\-'
```

Inny przykład na stronie 167.

9.5.11 re.purge

`re.purge()` — czyści pamięć podręczną (cache) silnika wyrażeń regularnych. Pyton utrzymuje specjalny cache do pamiętania ostatnio skompilowanych regeksów; gdy interpreter zorientuje się, że pewien regeks już był kompilowany, zamiast rekompilacji użyje zapamiętanego obiektu **Pattern**. Ten mechanizm jest bardzo efektywny, więc wywoływanie tej funkcji zwykle jest bezcelowe.

9.6 Metody obiektów typu Matcher

Metody matchera wywoływane są na obiekcie typu **Matcher** *uzyskanego w wyniku wywołania funkcji* **search**, **match**, **fullmatch** *i* **finditer**. *Będą one zilustrowane następującym przykładem:*

Listing 52 REN-Matcher/Matcher.py

```
#!/usr/bin/env python

import re

string = 's23453 (Jane Brown)'

regex = r'''(?x)            # verbose
            (?P<snum>\d+)   # group 'snum': student number
            \s+\(           # spaces and (
            (?P<full>       # group 'full': full name
             [^\W\d_]+\s+
             (?P<last>      # group 'last': last name
              [^\W\d_]+
             )              # end of group 'last'
            )               # end of group 'full'
            \)
```

```
        '''

matchiter = re.finditer(regex, string)
for matcher in matchiter:
    print(' 0. ' + str(matcher.groups()))  # tuple of groups
    print(' 1. ' + matcher.group())
    print(' 2. ' + matcher.group(0))       # group(0) = group()
    print(' 3. ' + matcher.group(1))
    print(' 4. ' + matcher.group(2))
    print(' 5. ' + matcher.group(3))
    print(' 6. ' + matcher[3])             # group(i) = matcher[i]
    print(' 7. ' + matcher.group('snum'))  # group by name
    print(' 8. ' + matcher.group('last'))
    print(' 9. ' + str(matcher.group(1,3)))  # yields a tuple
    print('10. ' + str(matcher.groupdict())) # dict of named groups
    print('11. ' + str(matcher.start(1)))    # where group 1 starts
    print('12. ' + str(matcher.end(1)))      # and where it ends
    print('13. ' + str(matcher.start('last'))) # now by name
    print('14. ' + str(matcher.end('last')))
    print('15. ' + str(matcher.span(2)))       # (start, end) tuple
    print('16. ' + str(matcher.span('full')))
    print('17. ' + str(matcher.lastindex))     # last group closed
    print('18. ' + matcher.lastgroup)          # name of it
    print('19. ' + matcher.string)             # original text
    print('20. ' + str(type(matcher.re)))      # pattern object
```

Mamy tu trzy (nazwane) grupy odpowiadające numerowi studenta, jej/jego pełnemu imieniu i nazwisku, oraz, jako część drugiej grupy, samemu nazwisku. Program drukuje

```
 0. ('23453', 'Jane Brown', 'Brown')
 1. 23453 (Jane Brown)
 2. 23453 (Jane Brown)
 3. 23453
 4. Jane Brown
 5. Brown
 6. Brown
 7. 23453
 8. Brown
 9. ('23453', 'Brown')
10. {'snum': '23453', 'full': 'Jane Brown', 'last': 'Brown'}
11. 1
12. 6
13. 13
14. 18
15. (8, 18)
16. (8, 18)
17. 2
18. full
19. s23453 (Jane Brown)
20. <class 're.Pattern'>
```

i będziemy się odnosić do numerów linii z programu i z wydruku w wyjaśnieniach poniżej.

9.6.1 matcher.groups, matcher.group

- `matcher.groups()` — zwraca krotkę napisów pasujących do kolejnych grup zdefiniowanych w regeksie; patrz linia 0. Liczba grup jest zatem `len(matcher.groups())`.
- `matcher.group()` — zwraca część tekstu pasującą do całego regeksu (nawet jeśli nie było tam zdefiniowanych żadnych grup); patrz linia 1.
- `matcher.group(index)` — zwraca, jako napis, dopasowanie grupy numer index z fragmentu tekstu dopasowanego do całego regeksu. Oczywiście taka grupa musi istnieć. Z indeksem 0 jest to równoważne wywołaniu **group()** bez argumentu. Patrz linie 2-5.
- `matcher.__getitem__(index)` — jest równoważne wywołaniu `matcher.group(index)`. Zauważmy, że `__getitem__` jest wywoływane gdy używamy indeksu (w nawiasach kwadratowych), więc równoważnie można po prostu napisać `matcher[index]`, jak to jest pokazane w linii 6.
- `matcher.group('name')` — zwraca, jako napis, dopasowanie do grupy nazwanej name z dopasowanego do całego regeksu podnapisu; oczywiście taka grupa musi istnieć. Patrz linie 7-8.
- `matcher.group(index1, index2, ...)` — zwraca w postaci krotki napisów dopasowanych do grup o podanych indeksach — patrz linia 9.

9.6.2 matcher.groupdict

`matcher.groupdict()` — zwraca słownik z nazwami grup jako kluczami i odpowiadającymi im dopasowanymi fragmentami tekstu jako wartościami (tylko nazwane grupy są brane pod uwagę — patrz linia 10.

9.6.3 matcher.start, matcher.end, matcher.span

- `matcher.start(index)` — zwraca indeks w całym tekście gdzie dopasowanie do grupy o danym indeksie się zaczyna; patrz linia 11.
- `matcher.end(index)` — zwraca indeks w całym tekście za końcem dopasowania do grupy o danym indeksie; patrz linia 12.
- `matcher.start('name')` — zwraca indeks w całym tekście gdzie dopasowanie do grupy o danej nazwie się zaczyna; patrz linia 13.
- `matcher.end('name')` — zwraca indeks w całym tekście za końcem dopasowania do grupy o danej nazwie; patrz linia 14.
- `matcher.span(index)` — zwraca krotkę zawierającą indeksy `matcher.start(index)` i `matcher.end(index)`; see line no 15.
- `matcher.span('name')` — to samo, ale dla grupy nazwanej; patrz linia 16.

9.6.4 matcher.lastindex, matcher.lastgroup, matcher.re, matcher.string

Nie są to funkcje/metody, ale *pola* (atrybuty) obiektu matchera:

- matcher.lastindex — indeks grupy, która została zamknięta jako ostatnia. W naszym przykładzie jest to 2, bo co prawda są trzy grupy, ale druga zawiera trzecią, więc kończy się później; patrz linia 17.

- matcher.lastgroup — nazwa grupy, która została zamknięta jako ostatnia. W naszym przykładzie jest to full, bo co prawda są trzy grupy, ale druga zawiera trzecią, więc kończy się później; patrz linia 18.
- matcher.string — tekst przekazany do funkcji **search**, **match**, **fullmatch** lub **finditer** z której zwrócony został ten obiekt matchera; patrz linia 19.
- matcher.re — obiekt **Pattern** (skompilowany regeks) użyty do utworzenia tego obiektu matchera; patrz linia 20.

9.7 Atrybuty obiektu Pattern

Opisaliśmy już metody obiektów **Pattern**, *(zob. rozdz. 9.5, str. 187) ale mają one też użyteczne atrybuty. Zilustrujemy je na następującym przykładzie, gdzie sam regeks jest taki sam jak w poprzednim przykładzie (Listing 52).*

Listing 53 REO-PattAttr/PattAttr.py

```
1  #!/usr/bin/env python
2
3  import re
4
5  regex = r'''(?P<snum>\d+)   # group 'snum': student number
6              \s+\(           # spaces and (
7              (?P<full>       # group 'full': full name
8               [^\W\d_]+\s+
9               (?P<last>      # group 'last': last name
10               [^\W\d_]+
11              )               # end of group 'last'
12             )                # end of group 'full'
13             \)
14          '''
15 patObj = re.compile(regex, re.VERBOSE)
16
17 print(' 1 -> ' + str(patObj.flags))        # flags (as an int)
18 print(' 2 -> ' + str(patObj.groups))       # number of groups
19 print(' 3 -> ' + str(patObj.groupindex))   # dict of (name, index)
20 print(' 4 -> ' + patObj.pattern)           # regex (as string)
```

Program drukuje

```
1 -> 96
2 -> 3
3 -> {'snum': 1, 'full': 2, 'last': 3}
4 -> (?P<snum>\d+)   # group 'snum': student number
           \s+\(           # spaces and (
           (?P<full>       # group 'full': full name
            [^\W\d_]+\s+
            (?P<last>      # group 'last': last name
             [^\W\d_]+
```

```
 )              # end of group 'last'
)               # end of group 'full'
\)
```

i będziemy się odnosić do numerów linii z programu i z wydruku w wyjaśnieniach poniżej.

- patObj.flags — flaga opcji kompilacji przy kompilowaniu regeksu; liczba całkowita równa zORowanym ustawionym flagom. Tu jest to 96, czyli 32 (odpowiadająca re.UNICODE, która jest ustawiona domyślnie) oraz 64 (odpowiadająca re.VERBOSE), co można zobaczyć z wydruku Listing 48. Patrz linia 1.
- patObj.groups — liczba grup w regeksie; patrz linia 2.
- patObj.groupindex — słownik z nazwami wszystkich nazwanych grup jako kluczami i odpowiadającymi im indeksami jako wartościami; patrz linia 3.
- patObj.pattern — regeks użyty do utworzenia tego obiektu **Pattern**; patrz linia 4.

Rozdział 10

Wyjątki

10.1 Podnoszenie i obsługa wyjątków

Wyjątki i ich obsługa pozwalają na zmianę przebiegu programu gdy zajdzie jakieś niespodziewane zdarzenie i dalsze jego kontynuowanie staje się niemożliwe. Może to być, na przykład, próba dzielenia przez zero lub czytania pliku, który nie istnieje. Wyjątki są w Pythonie wszechobecne, więc jest niezwykle ważne by je rozumieć i umieć je obsługiwać.[1]

W sytuacjach, gdy interpreter nie może normalnie kontynuować programu, podnosi (eng. *raises*) wyjątek, który, jak wszystko w Pythonie, reprezentowany jest przez pewien obiekt (co znamy też z Javy czy C++). Jeśli nie podpowiemy interpretorowi, co robić gdy wyjątek się zdarzył, program zostanie przerwany, ale przynajmniej wyświetli jakąś informację na temat przyczyny i lokalizacji błędu. Spróbujmy:

```
1   def g(num, d):
2       return num // d
3
4   def f(num, a, b):
5       for d in range(a, b+1):
6           x = g(num, d)
7           print(f'{num}//{d}={x}')
8
9   def main():
10      f(5, -3, 2)
11
12  main()
```

Wykonanie tego małego programu drukuje

```
5//-3=-2
5//-2=-3
5//-1=-5
Traceback (most recent call last):
File "/usr/lib/python3.10/idlelib/run.py", line 578,
    in runcode exec(code, self.locals)
File "/home/werner/p.py", line 12, in <module>
    main()
File "/home/werner/p.py", line 10, in main
    f(5, -3, 2)
File "/home/werner/p.py", line 6, in f
    x = g(num, d)
File "/home/werner/p.py", line 2, in g
    return num // d
ZeroDivisionError: integer division or modulo by zero
```

Czytamy to od dołu do góry

[1] Szczęśliwie wyjątki w Pythonie są podobne do tych znanych nam już z Javy lub C++. Jednakże, nie ma tu pojęcia wyjątków sprawdzanych i niesprawdzanych, jak w Javie — w tym sensie wszystkie wyjątki w Pythonie są niesprawdzane: możemy, ale nie musimy ich obsługiwać.

- Wyjątek, który się zdarzył, reprezentowany jest obiektem typu **ZeroDivisionError** z opisem *integer division or modulo by zero.*
- Został podniesiony przy wykonywaniu linii 2, w ciele funkcji **g**: linia w której wystąpił to `return num // d`
- Ponieważ wyjątek nie został obsłużony, idziemy w górę[1] stosu do następnej (głębszej) ramki: funkcja **g** była wywołana z funkcji **f** w linii 6 (`x = g(num, d)`).
- Ta z kolei funkcja była wywołana z funkcji **main** w linii 10 (`f(5, -3, 2)`)
- A funkcja **main** została wywołana bezpośrednio z modułu w linii 12 (`main()`)...
- Który został wywołany z interpretera Pythona.

Wyjątki nie muszą jednak powodować załamania programu. Możemy poinformować interpreter co ma robić w razie ich wystąpienia poprzez umieszczenie kodu, gdzie potencjalnie możliwe jest podniesienie (zgłoszenie) wyjątku wewnątrz bloku **try** i dostarczenie odpowiedniego bloku **except**: jeśli wyjątek rzeczywiście się zdarzy podczas wykonywania bloku **try**,[2] to jego wykonywanie jest przerywane i sterowanie przechodzi bezpośrednio do bloku **except**: w tym momencie wyjątek uznaje się za „obsłużony" i program może kontynuować działanie

```
a, b = 1, 0
try:
    z = a // b
    print(z)      # will not be executed
except:
    print('Exception handled')

print('Continuing...')
```

drukuje

```
Exception handled
Continuing..
```

W powyższym przykładzie użyliśmy po prostu **except**, bez specyfikowania jaki typ wyjątku chcemy tu obsługiwać: wszystkie możliwe wyjątki zostaną tu „wyłapane".

Lepiej jednak być bardziej specyficznym. Za blokiem **try** możemy umieścić kilka bloków **except** wyłapujących wyjątki różnych typów: pierwszy blok pasujący do typu wyjątku będzie wykonany, a pozostałe – zignorowane. Program

```
from math import sqrt

a, b = -1, 0
try:
    x = sqrt(a)
    y = a // b
except ValueError:
    print('ValueError occurred')
except ZeroDivisionError:
    print('ZeroDivisionError occurred')
```

[1] Dlaczego *w górę*, a nie *w dół*? W większości architektur stos rośnie w stronę mniejszych adresów, więc jeśli wizualizujemy stos z większymi adresami u góry, to stos rośnie w dół, a zwijany jest w górę.

[2] ...co może też nastąpić w funkcji wywoływanej w tym bloku, lub funkcji wywoływanej przez tę funkcję, itd.

```
a, b = 1, 0
try:
    x = sqrt(a)
    y = a // b
except ValueError:
    print('ValueError occurred')
except ZeroDivisionError:
    print('ZeroDivisionError occurred')
```

drukuje

```
ValueError occurred
ZeroDivisionError occurred
```

Musimy oczywiście pamiętać o hierarchii klas: blok **except** odpowiadający pewnemu typowi będzie pasował nie tylko do tego konkretnego typu, ale również jego podtypów. Coś takiego

```
try:
    # ...
except LookupError:
    # ...
except IndexError:
    # ...
```

nie ma sensu, bo **IndexError** jest podtypem **LookupError**, więc wyjątki tego typu będą przechwycone przez pierwszy blok **except**, czyniąc ten drugi nieosiągalnym. Jednak

```
try:
    # ...
except IndexError:
    # ...
except LookupError:
    # ...
```

może być rozsądne: pierwszy **except** wychwyci wyjątki typu **IndexError**, a drugi typu **LookupError** oraz jego podtypów niebędących typu **IndexError** (w naszym przypadku byłby to **IndexError** oraz ewentualnie jego podtypy utworzone przez użytkownika).

W ten sposób jednak nie mamy dostępu do obiektu reprezentującego wyjątek, który może zawierać użyteczną informację. Możemy taki dostęp uzyskać poprzez użycie `as name`, co wiąże nazwę (dowolną) `name` z tym obiektem i daje dostęp do jego metod i atrybutów:

```
import traceback

a, b = 1, 0
try:
    z = a // b
except ArithmeticError as ex:
    print('*** 1. Info:\n', type(ex), str(ex))
    print('*** 2. Attributes:\n', str(dir(ex)))
    print('*** 3. Doc:\n', ex.__doc__)
```

```
        print('*** 4. Args:\n', type(ex.args), str(ex.args))
        print('*** 5. Traceback:\n', type(ex.__traceback__),
                                     str(ex.__traceback__))
        print('*** 6. Trace:\n' + '-'*35)
        traceback.print_exception(ex)
        print('-'*35)
```

drukuje

```
*** 1. Info:
<class 'ZeroDivisionError'> integer division or modulo by zero
*** 2. Attributes:
['__cause__', '__class__', '__context__', '__delattr__',
 '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__getstate__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__le__',
 '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__setstate__', '__sizeof__',
 '__str__', '__subclasshook__', '__suppress_context__',
 '__traceback__', 'add_note', 'args', 'with_traceback']
*** 3. Doc:
Second argument to a division or modulo operation was zero.
*** 4. Args:
<class 'tuple'> ('integer division or modulo by zero',)
*** 5. Traceback:
<class 'traceback'> <traceback object at 0x7fa29a45ac00>
*** 6. Trace:
-----------------------------------
Traceback (most recent call last):
File "/home/werner/python/exceptions.py", line 5, in <module>
    z = a // b
        ~~~~~~
ZeroDivisionError: integer division or modulo by zero
```

Przypatrzmy się kolejnym wydrukom z funkcji **print** w blokach **except**

*** **1.** Prawdziwym typem wyjątku był **ZeroDivisionError**, który jest podtypem **ArithmeticError**. Funkcja **str** drukuje informacje z atrybutu args obiektu ex (patrz niżej).

*** **2.** Jak widzimy, obiekt ex jest całkiem bogaty i ma wiele atrybutów.

*** **3.** Atrybut __doc__ zawiera napis dokumentujący (. *docstring.*)

*** **4.** Atrybut args jest krotką argumentów przesłanych do konstruktora gdy ex był tworzony; zwykle jest to pojedynczy napis z opisem błędu. Zawartość krotki args będzie wypisywana przez metodę **__str__**, w szczególności, gdy drukowany jest ślad stosu (ang. *stack trace.*) Możemy zdefiniować tę krotkę poprzez przesłanie dowolnych argumentów w momencie tworzenia obiektu wyjątku, czy to z biblioteki standardowej czy naszego własnego, „ręcznie” (patrz niżej).

*** **5.** Atrybut __traceback__ zawiera informacje o stanie stosu.

*** **6.** Atrybut traceback będzie użyty przez funkcję **print_exception** przy wypisywaniu informacji o stanie stosu (tzw. *ślad stosu.*)[1]

Możemy również w pojedynczym bloku **except** obsługiwać kilka niezwiązanych ze sobą wyjątków, umieszczając je w krotce (z nawiasami). Na przykład

```
from math import sqrt

a, b = -1, 0
try:
    x = sqrt(a)
    y = a // b
except (ValueError, ZeroDivisionError) as ex:
    print(type(ex).__name__, ex)
print('Exception handled')

print()

a, b = 1, 0
try:
    x = sqrt(a)
    y = a // b
except (ValueError, ZeroDivisionError) as ex:
    print(type(ex).__name__, ex)
print('Exception handled')
```

drukuje

```
ValueError math domain error
Exception handled

ZeroDivisionError integer division or modulo by zero
Exception handled
```

Możemy podnieść wyjątek „ręcznie” stosując instrukcję **raise**. Jako argument podajemy obiekt istniejącego typu wyjątku: z biblioteki lub naszego własnego typu dziedziczącego z **Exception**. Podanie samej nazwy tego typu jest równoważne utworzeniu obiektu bez przesyłania czegokolwiek do konstruktora. Zatem następujące formy podniesienia wyjątku

```
e = Exception()
raise e

raise Exception()

raise Exception
```

są równoważne.

Można też użyć pop prostu **raise** bez specyfikowania typu wyjątku. Taka instrukcja ponawia aktualnie obsługiwany wyjątek, a zatem może być użyta tylko wewnątrz bloku

[1] Podobnej do **printStackTrace** w Javie.

except. Może to być użyteczne, gdy chcemy na przykład wypisać, albo wpisać to pliku logowania, informację o wyjątku, ale przekazać jego obsługę w górę do wywołującego.

W przykładzie poniżej, jeśli wyjątek wystąpił, drukujemy o nim informację i go ponawiamy (line ❷). Jego obsługę przekazujemy tym samym do funkcji wywołującej. Jeśli wyjątku nie było, zwracamy rezultat w bloku else (linia ❸, patrz dalej). Funkcja wywołująca (linia ❹) drukuje wynik lub informację o wystąpieniu błędu, ale nawet wtedy pozwala programowi kontynuować:

Listing 54 CAH-Reraise/Reraising.py

```
#!/usr/bin/env python

import math

def fun(a, b):
    try:
        sq = math.sqrt(a)
        lo = math.log(b)
        res = sq/lo
    except Exception as e:                          ❶
        print(f'{type(e).__name__} in fun')
        raise                                       ❷
    else:                                           ❸
        return res

args = [(4, math.e), (4, 1), (-4, 1), (4, -1), (4, math.e**2)]
for a, b in args:
    print(f'\n*** Calling fun with {a=}, {b=}')
    try:                                            ❹
        r = fun(a, b)
        print(f'Result is {r}')
    except:                                         ❺
        print('This call failed: continuing')
```

Program drukuje

```
*** Calling fun with a=4, b=2.718281828459045
Result is 2.0

*** Calling fun with a=4, b=1
ZeroDivisionError in fun
This call failed: continuing

*** Calling fun with a=-4, b=1
ValueError in fun
This call failed: continuing

*** Calling fun with a=4, b=-1
ValueError in fun
This call failed: continuing
```

```
*** Calling fun with a=4, b=7.3890560989306495
Result is 1.0
```

Powyższy przykład ilustruje też tak zwany *„catch all”* blok **except**. Użyty jest tu w linii ❶, gdzie jako typu wychwytywanego wyjątku podany jest **Exception**. To wychwyci (prawie) wszystkie wyjątki, gdyż wszystkie dziedziczą, pośrednio lub bezpośrednio, z **Exception** (jest kilka typów stojących wyżej w hierarchii klas wyjątków, ale one prawie nigdy w ogóle nie powinny być obsługiwane). Jeśli nie interesuje nas obiekt reprezentujący wyjątek, możemy po prostu nie wiązać go z żadną nazwą, pisząc po prostu **except:**, jak w linii ❺. Trzeba jednak pamiętać, że **except:** (bez specyfikowania żadnego typu) nie jest zalecane, gdyż wyłapie również wyjątki, które obsługiwane być *nie* powinny, jak **SystemExit**, **KeyboardInterrupt** lub **GeneratorExit**.

Podobnie jak w Javie, po wszystkich blokach **except** możemy (opcjonalnie) dodać blok **finally** zawierający kod, który ma być wykonany zawsze, niezależnie od tego, czy wyjątek wystąpił , czy nie. Nawet jeśli w bloku **try** lub **except** zawarta jest instrukcja **return**, tuż przed jej wykonaniem wykonany zostanie blok **finally**:

```
>>> def f(arg):
...     try:
...         return len(arg)
...     finally:
...         print('finally executed')
...
>>> res = f('a string')
finally executed
>>> print(res)
8
>>> res = f(7.5)
finally executed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in f
TypeError: object of type 'float' has no len()
```

Zauważmy że, jak w powyższym przykładzie, bloku **finally** możemy użyć nawet jeśli nie było żadnego bloku **except**!

Jeśli wchodząc do bloku **finally** jakiś wyjątek pozostaje nieobsłużony, jak widzieliśmy w powyższym przykładzie, to będzie po wykonaniu tego bloku ponowiony. Natomiast jeśli kod takiego bloku sam zawiera **return** (albo **break** lub **continue**, jeśli jesteśmy w jakiejś pętli), to takiego ponowienia *nie* będzie:

```
>>> def f(arg):
...     try:
...         return len(arg)
...     finally:
...         print('finally executed')
...         return 'ERROR'
...
>>> res = f(7.5)
finally executed
```

```
>>> print(res)
ERROR
```

Inaczej niż w Javie, w Pythonie można też użyć bloku **else**.. Jest używany rzadko, ale jeśli jest użyty, to musi występować za wszystkimi blokami **except**, a przed blokiem **finally** (jeśli występuje). Blok **else** będzie wykonany **tylko** jeśli wyjątku w bloku **try** *nie* było. Zobaczmy to w poniższym małym przykładzie:

Listing 55 CAC-Clauses/Clauses.py

```
#!/usr/bin/env python

def fun(num, denom):
    try:
        print('In try')
        z = num // denom
    except ArithmeticError as zero:
        print('Exception occurred')
        print(zero)
        return None
    else:
        print('returning result:', z)
        return z
    finally:
        print('finally executed just before returning!')

print('*** This will go smoothly')
print(fun(27, 4))
print('*** This will raise an exception')
print(fun(31, 0))
```

który drukuje

```
*** This will go smoothly
In try
returning result: 6
finally executed just before returning!
6
*** This will raise an exception
In try
Exception occurred
integer division or modulo by zero
finally executed just before returning!
None
```

Jeśli nie ma wyjątku w bloku **try**, program przechodzi do bloku **else** i zwraca z (ale blok **finally** i tak jest wykonywany). Jeśli wyjątek wystąpi, przejdziemy do bloku **except** i tam zwrócimy **None** (ale **finally** też będzie wykonane).

Inną użyteczną cechą wyjątków w Pythonie jest to, że tworząc obiekt wyjątku możemy do konstruktora przesłać dowolną liczbę dowolnych argumentów. Wszystkie

one, w postaci krotki, będą zapamiętane jako atrybut args tworzonego obiektu. Atrybut ten będzie wypisywany przez standardową metodę __str__, oraz, w szczególności, podczas drukowania śladu stosu:

```
>>> ex = Exception('Raising exception just for fun', 42, '!!!!')
>>> ex.args
('Raising exception just for fun', 42, '!!!!')
>>> raise ex
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: ('Raising exception just for fun', 42, '!!!!')
```

Najczęściej do konstruktora posyłamy tylko jeden argument: jakiś komunikat wyjaśniający naturę błędu – będzie on widoczny w razie wystąpienia tego wyjątku.[1] Ten mechanizm działa, gdy sami tworzymy obiekt wyjątku. Ale można też dodać informację do już istniejącego wyjątku: wszystkie klasy wyjątków mają metodę add_note, która dodaje przesłany przez argument napisowy do atrybutu __notes__ – jest to lista, która również zostanie wyświetlona przy drukowaniu śladu stosu.

Może to być przydatne, gdy wyjątek jest ponawiany i przechodzi przez kilka ramek przy zwijaniu stosu, aż do napotkania odpowiedniego bloku except lub załamania programu. Dodawać wtedy można informacje do istniejącego obiektu wyjątku „po drodze", jak w przykładzie poniżej

```
def funb(a, b):
    try:
        return a // b
    except Exception as e:
        e.add_note(f'In funb with args {a=}, {b=}')
        raise

def funa(x, y):
    try:
        return funb(x+y, x-y)
    except Exception as e:
        e.add_note(f'In funa with args {x=}, {y=}')
        raise

r = funa(5, 5)
print(f'{r=}')
```

który drukuje

```
Traceback (most recent call last):
  File "/home/werner/python/Notes.py", line 17, in <module>
    r = funa(5, 5)
        ^^^^^^^^^^
  File "/home/werner/python/Notes.py", line 12, in funa
    return funb(x+y, x-y)
           ^^^^^^^^^^^^^^
  File "/home/werner/python/Notes.py", line 5, in funb
```

[1] Istnieje konwencja, że komunikat ten powinien zaczynać się z małej litery i nie zawierać na końcu kropki.

```
        return a // b
               ~~~~~~
    ZeroDivisionError: integer division or modulo by zero
    In funb with args a=10, b=0
    In funa with args x=5, y=5
```

i gdzie informacja o argumentach przesłanych do funkcji jest dodawana do atrybutu __notes__ obiektu reprezentującego wyjątek.

W Javie/C++ wyjątki są traktowane jako zło konieczne; coś, co w idealnym świecie nie powinno się zdarzyć, ale musimy to zaakceptować wiedząc, że świat doskonały nie jest. W Pythonie podejście jest nieco bardziej liberalne. Jak widzieliśmy, w pewnych sytuacjach wyjątki są stosowane intencjonalnie, bez żadnych nieszczęśliwych zdarzeń w tle – na przykład za każdym razem kiedy kończy się iteracja (zob. rozdz. 2.5, str. 24 i rozdz. 6, str. 93).

Wyjątki mogą też być stosowane, na przykład, do przerwania głęboko zagnieżdżonych pętli. Zamiast stosowania boolowskich zmiennych do wyjścia poprzez kilka poziomów pętli,[1] czytelniejsze i prostsze może być zastosowanie wyjątku.

W przykładzie poniżej poszukujemy indeksów (w rosnącym porządku) trzech elementów listy lst, które sumują się do zadanej wartości sumOf3. Gdy je znajdziemy, nie ma sensu kontynuować pętli, więc podnosimy wyjątek (❶) przekazując znalezione indeksy (czyli aktualne wartości a, b i c) do obiektu wyjątku — ich krotka stanie się atrybutem args tego obiektu, który w bloku except "wyciągamy" z ex (❷). Jeśli wyjątku nie było, wykonywany jest blok else (❸), który ustawia rozwiązanie na (-1, -1, -1), co sygnalizuje brak rozwiązania:

Listing 56 CAF-ExcBreak/StrangeExc.py

```
1  #!/usr/bin/env python
2
3  lst = [9, 1, 5, 7, 11, 10, 4]
4  sumOf3 = int(input('Expected sum: '))
5
6  try:
7      for a in range(len(lst)-2):
8          for b in range(a+1, len(lst)-1):
9              for c in range(b+1, len(lst)):
10                 if lst[a] + lst[b] + lst[c] == sumOf3:
11                     raise Exception(a, b, c)  ❶
12 except Exception as ex:
13     res = ex.args                             ❷
14 else:
15     res = (-1, -1, -1)                        ❸
16
17 print(res)
```

Wykonanie programu mogłoby wyglądać tak:

```
python> ./StrangeExc.py
```

[1] Nie ma w Pythonie pętli etykietowanych, jak w Javie.

```
Expected sum: 11
(-1, -1, -1)
python> ./StrangeExc.py
Expected sum: 10
(1, 2, 6)
```

10.2 Built-in and custom exceptions

Standard Pythona dostarcza wiele typów wyjątków w formie hierarchii klas, której korzeniem jest klasa **BaseException**. Te wyjątki są podnoszone przez standardowe funkcje, ale możemy definiować własne typy poprzez dziedziczenie z jednego ze standardowych wyjątków: **Exception** lub jednego z jego podklas — *nie* jest zalecane dziedziczenie bezpośrednio z **BaseException** lub jego dzieci **SystemExit**, **KeyboardInterrupt**, lub **GeneratorExit**). Nie jest również dobrym pomysłem dziedziczenie z kilku klas wyjątków, bo prowadzić to może do niespodziewanego zachowania.

Listę standardowych wyjątków można znaleźć w oficjalnej dokumentacji[1] i zamieszamy ją poniżej (pochodzi z wersji 3.10; nowe typy są dodawane w kolejnych wersjach Pythona):

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
     +-- StopIteration
     +-- StopAsyncIteration
     +-- ArithmeticError
     |    +-- FloatingPointError
     |    +-- OverflowError
     |    +-- ZeroDivisionError
     +-- AssertionError
     +-- AttributeError
     +-- BufferError
     +-- EOFError
     +-- ImportError
     |    +-- ModuleNotFoundError
     +-- LookupError
     |    +-- IndexError
     |    +-- KeyError
     +-- MemoryError
     +-- NameError
     |    +-- UnboundLocalError
     +-- OSError
     |    +-- BlockingIOError
     |    +-- ChildProcessError
     |    +-- ConnectionError
     |    |    +-- BrokenPipeError
     |    |    +-- ConnectionAbortedError
     |    |    +-- ConnectionRefusedError
```

[1] https://docs.python.org/3/library/exceptions.html

```
|     |     +-- ConnectionResetError
|     +-- FileExistsError
|     +-- FileNotFoundError
|     +-- InterruptedError
|     +-- IsADirectoryError
|     +-- NotADirectoryError
|     +-- PermissionError
|     +-- ProcessLookupError
|     +-- TimeoutError
+-- ReferenceError
+-- RuntimeError
|     +-- NotImplementedError
|     +-- RecursionError
+-- SyntaxError
|     +-- IndentationError
|          +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|     +-- UnicodeError
|          +-- UnicodeDecodeError
|          +-- UnicodeEncodeError
|          +-- UnicodeTranslateError
+-- Warning
     +-- DeprecationWarning
     +-- PendingDeprecationWarning
     +-- RuntimeWarning
     +-- SyntaxWarning
     +-- UserWarning
     +-- FutureWarning
     +-- ImportWarning
     +-- UnicodeWarning
     +-- BytesWarning
     +-- EncodingWarning
     +-- ResourceWarning
```

W większości przypadków zaleca się stosowanie wbudowanych klas wyjątków. Zauważmy, że zwykle najważniejsza informacja zawarta jest w samej nazwie wyjątku; na przykład

ModuleNotFoundError lub ImportError gdy są kłopoty ze znalezieniem lub wczytaniem modułu;

NameError gdy nazwa nie jest zdefiniowana;

AttributeError gdy odwołujemy się do nieistniejącego atrybutu;

IndexError gdy użyjemy w sekwencji indeksu o wartości poza dozwolonym zakresem;

KeyError gdy użyjemy w słowniku nieistniejącego klucza;

TypeError gdy funkcja lub operator ma być zastosowany do obiektu o nieprawidłowym type;

ValueError gdy funkcja lub operator ma być zastosowany do obiektu o właściwym typie, ale niedozwolonej wartości;

ZeroDivisionError gdy drugi operand dzielenia lub operatora modulo ma wartość 0;

UnicodeError gdy wystąpi błąd przy kodowaniu lub odkodowywaniu tekstu;;

RecursionError gdy osiągnięta została maksymalna głębokość rekursji.

Jeśli żadna ze standardowych klas wyjątków nie odpowiada naszym potrzebom, możemy utworzyć własny typ. Powinien on dziedziczyć z klasy **Exception** lub jednej z jej podklas. [1] Oczywiście, jest to zwykła klasa, więc można w niej definiować atrybuty, metody czy pola w zwykły sposób. Zwykle jednak pozostawiamy ją jak najprostszą; nawet definicja

```
class MyException(Exception):
    pass
```

może być wystarczająca, jeśli sama nazwa jest tym czego potrzebujemy.

W przykładzie poniżej definiujemy klasę **OutOfRange**. Jest przewidziana na sytuacje, gdy jakaś wartość nie należy do przedziału, do którego należeć powinna. Naturalne jest zatem, że dziedziczy z **ValueError**. Konstruktor pobiera tę nieprawidłową wartość (val) oraz dolną i górną granicę przedziału, do którego powinna należeć. Następnie wywołuje konstruktor klasy nadrzędnej, przekazując te wszystkie informacje w postaci jednego napisu:

Listing 57 CAJ-RangeError/Range.py

```
#!/usr/bin/env python

class OutOfRange(ValueError):
    def __init__(self, val, lower, upper):          ❶
        super().__init__(f'Got {val} which is out'
                         f' of [{lower}, {upper}]')


def getMonthName(n):
    names = ['January', 'February', 'March', 'April',
             'May', 'June', 'July', 'August',
             'September', 'October', 'December']
    if (n < 1 or n > 12): raise OutOfRange(n, 1, 12)
    return names[n-1];

m = getMonthName(13)
print(m)
```

Program drukuje

```
File "/home/werner/python/Range.py", line 16, in <module>
  m = getMonthName(13)
```

[1] Zob. rozdz. ??, str. ?? o dziedziczeniu.

```
        ~~~~~~~~~~~~~~~~
  File "/home/werner/python/Range.py", line 13, in getMonthName
    if (n < 1 or n > 12): raise OutOfRange(n, 1, 12)
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^
OutOfRange: Got 13 which is out of [1, 12]
```

10.3 Assercje

Asercje znamy z Javy i C++. Podobny mechanizm można też zastosować w Pythonie. Instrukcja **assert** jest prosta:

```
assert expression
```

lub

```
assert expression, message
```

Tutaj expression jest dowolnym wyrażeniem, które będzie zinterpretowane jako **True** lub **False** według znanych nam zasad: jeśli będzie to **True**, nic się nie stanie, jeśli **False**, podniesiony zostanie wyjątek **AssertionError**. Ten wyjątek nie powinien być obsługiwany: spodziewamy się, że expression będzie zawsze prawdziwe; jeśli nie jest, to sygnalizuje jakiś poważny błąd w logice programu. Opcjonalny argument message będzie przekazany do konstruktora nadklasy i wypisany gdy wyjątek się zdarzy.

Instrukcja **assert** jest równoważna

```
if __debug__:
    if not expression: raise AssertionError
```

lub

```
if __debug__:
    if not expression: raise AssertionError(message)
```

Ale co to jest __debug__? Jest to globalna niemodyfikowalna zmienna zainicjowana podczas uruchamianiu programu. Normalnie jest ustawiana na **True**. Gdy program jest już sprawdzony i gotowy do działania produkcyjnego, możemy uruchomić go ze zmienną __debug__ ustawioną na **False** poprzez użycie opcji optymalizacji (opcja '-O' – to jest litera, nie cyfra 0) przy wywołaniu interpretera. Wtedy asercje są całkowicie ignorowane, tak że program jest nieco szybszy, jak to pokazuje poniższy przykład:

Listing 58 CAL-Assert/FiboAssert.py

```
1 def fibo(n):
2     assert n >= 0, 'Argumnet of fibo must be non-negtive'
3     if n <= 1: return n
4     return fibo(n-2) + fibo(n-1)
5
6 print('fibo(40) =', fibo(40), end=' ')
```

Mierzymy tu czas wykonania z asercjami lub bez nich:

```
> time python FiboAssert.py
fibo(40) = 102334155 14.312u 0.000s 0:14.31 100.0% 0+0k 0+0io 0pf+0w
> time python -O FiboAssert.py
fibo(40) = 102334155 11.949u 0.003s 0:11.95 99.9%  0+0k 0+0io 0pf+0w
```

Widzimy, że nawet w tak ekstremalnie sztucznym przykładzie[1] oszczędność czasu nie jest bardzo znacząca: zyskaliśmy ok. 16 %.

[1]Przy rekurencyjnym obliczaniu F_{40} funkcja **fibo** jest wywoływana ponad 300 milionów razy, za każdym razem sprawdzając asercję.

Rozdział 11

Klasy

11.1 Wstęp

Tak jak inne zorientowane obiektowo języki, Python pozwala na tworzenie własnego typu danych poprzez zdefiniowanie klasy — obiektu specjalnego typu, który jest „producentem" („fabryką") obiektów tego nowego typu. Ale jakiego typu jest ten obiekt? Odpowiedź jest: klasy są same *obiektami* typu **type**!

```
>>> class AClass():
...     pass
>>> a = AClass()
>>> type(AClass)
<class 'type'>
>>> type(a)
<class '__main__.AClass'>
```

W zasadzie obiekty — a zatem i klasy — są słownikami.

Zwykle klasy definiują zestaw *atrybutów* (składowych), które będą dzielone przez wszystkie obiekty tejże klasy. W Pytonie są to

- *Metody niestatyczne*, czyli funkcje, których pierwszym argumentem będzie zawsze referencja do obiektu danej klasy. Konwencjonalnie, odpowiadający mu parametr ma nazwę self.
- *Pola klasowe*, które „należą" do samej klasy, a nie do jej obiektów — odpowiadają one polom statycznym w C++/Javie.
- *Metody statyczne*, które zachowują się jak „normalne" funkcje, ale należą do przestrzeni nazw klasy — odpowiadają one metodom statycznym w C++/Javie.
- *Metody klasowe*, czyli funkcje, których pierwszym argumentem będzie zawsze referencja do obiektu reprezentującego *klasę* (niekoniecznie *tę* klasę, w której je definiujemy, często jest to klasa pochodna *tej* klasy). Konwencjonalnie, odpowiadający mu parametr ma nazwę cls. Nie ma odpowiadającego metodom klasowym konstruktu w C++/Javie.

Co może tu być zaskakujące, to brak „normalnych", niestatycznych pól, które istniałyby w każdym obiekcie klasy z osobna (jak w C++/Javie) — więcej o tym powiemy za chwilę.

Definicja najprostszej możliwej klasy w Pythonie wygląda tak:

```
class Klazz:
    pass
```

Tak jak definicja funkcji, również definicja klasy jest *instrukcją wykonywalną:* tworzony jest obiekt reprezentujący klasę i związywany z nazwą Klazz. Wydawałoby się, że taka klasa jest kompletnie pusta. Tak jednak nie jest:

```
>>> class Klazz:
...     pass
...
>>> dir(Klazz)
```

```
['__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__',
 '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__']
>>>
>>> callable(Klazz)
True
>>> k = Klazz()            # tworzenie obiektu
>>> type(k)
<class '__main__.Klazz'>
>>> k.__hash__()           # __hash__ działa
8776042262012
>>>
>>> AnotherName = Klazz    # referencja to tego samego obiektu
>>> n = AnotherName()      # tworzenie obiektu
>>> type(n)
<class '__main__.Klazz'>
>>> n.__sizeof__()
32
```

Klasy, jak wszystko w Pythonie, są reprezentowane przez obiekty. Obiekty te są wywoływalne (*callable*), jak to widać z powyższego fragmentu. Wywołanie klasy zwraca obiekt (instancję) tej klasy,
Obiekty reprezentujące klasy *nie* są niezmienne: możemy, na przykład, dodawać do nich dynamicznie nowe atrybuty, tak zresztą jak i do tej klasy obiektów.
Prosty przykład (za chwilę omówimy szczegóły):

Listing 59 ABC-Members/Members.py

```
1  #!/usr/bin/env python
2
3  class Person:
4      def __init__(self, name):
5          self.name = name
6      def showName(self):
7          print(self.name)
8
9  mary = Person("Mary")
10 mary.showName()         # these two invocations
11 Person.showName(mary)   # are equivalent
12
13 try:
14     print(mary.age)
15 except AttributeError:
16     print('Mary has no age')
17
18 mary.age = 23                                    (1)
19 print('And now she has one!')
```

```
20 print(mary.age)
21
22 Person.showAge = lambda p: print(p.age)          ❷
23 print('And now the class has a new method!')
24 mary.showAge()
```

W linii ❶ dodajemy atrybut age (którego nie było w definicji klasy **Person**) do pojedynczego obiekty klasy; w linii ❷ natomiast, do istniejącej klasy dodajemy nową metodę. Program drukuje

```
Mary
Mary
Mary has no age
And now she has one!
23
And now the class has a new method!
23
```

Rozpatrzmy teraz poniższy przykład:

Listing 60 BBB-Classes/Classes.py

```
1  #!/usr/bin/env python
2
3  class Klazz:
4      pass
5
6  def hello(self, s):                # just a function
7      print(s, 'self =', self)
8
9  Klazz.hello = hello                # will be attribute of Klazz
10
11 print('* 1*', hello)
12 print('* 2*', Klazz.hello)
13
14 k = Klazz()                         # creating object
15 Klazz.hello(k, '* 3* HelloK,')
16 k.hello('* 4* HelloK as method,') # k will be first argument
17
18
19 class Clazz:
20     def hello(self, s):            # will be attribute of Clazz
21         print(s, 'self =', self)
22
23 print('* 5*', type(Clazz.hello))
24
25 c = Clazz()
26 Clazz.hello(c, '* 6* HelloC,')
27 c.hello('* 7* HelloC as method,') # c will be first argument
28
```

```
29 Klazz.att = 'Attribute of Klazz'
30 Clazz.att = 'Attribute of Clazz'
31
32 cc = Clazz()
33 print('* 8*', Klazz.att)
34 print('* 9*', k.att)              # k created before
35 print('*10*', Clazz.att)
36 print('*11*', c.att)              # c created before
37 print('*12*', Clazz.__dict__)
38 print('*13*', cc.__dict__)
39
40 Clazz.hello('John', '*14* Mary,')
```

Program drukuje

```
* 1* <function hello at 0x7fd6aa503d90>
* 2* <function hello at 0x7fd6aa503d90>
* 3* HelloK, self = <__main__.Klazz object at 0x7fd6aa396da0>
* 4* HelloK as method, self = <__main__.Klazz object at 0x7fd6aa396da0>
* 5* <class 'function'>
* 6* HelloC, self = <__main__.Clazz object at 0x7fd6aa396f50>
* 7* HelloC as method, self = <__main__.Clazz object at 0x7fd6aa396f50>
* 8* Attribute of Klazz
* 9* Attribute of Klazz
*10* Attribute of Clazz
*11* Attribute of Clazz
*12* {'__module__': '__main__',
      'hello': <function Clazz.hello at 0x7fd6aa3c6440>,
      '__dict__': <attribute '__dict__' of 'Clazz' objects>,
      '__weakref__': <attribute '__weakref__' of 'Clazz' objects>,
      '__doc__': None, 'att': 'Attribute of Clazz'}
*13* {}
*14* Mary, self = John
```

Tworzymy „pustą” klasę **Klazz** i zwykłą dwuparametrową funkcję **hello**. Nazwą pierwszego parametru jest tu self, ale nie oznacza to niczego specjalnego. Następnie (linia 9), dodajemy do obiektu reprezentującego klasę **Klazz** nowy atrybut: jego nazwą jest zmiennahello i odnosi się do obiektu reprezentującego funkcję **hello**, co widzimy z wydruku linii 11 and 12

```
* 1* <function hello at 0x7fd6aa503d90>
* 2* <function hello at 0x7fd6aa503d90>
```

W linii 14 wywołujemy obiekt (wywoływalny) **Klazz** i dostajemy k — obiekt tej klasy. Spójrzmy teraz na linie 15-16. Wywołujemy tu bezpośrednio funkcję **hello** klasy **Klazz** przesyłając k jako pierwszy argument (ale mogłoby to być cokolwiek innego).

W linii 16 wywołanie jest inne – wywołujemy tę samą funkcję, ale „na rzecz” obiektu k stosując notację z kropką (znaną nam z C++/Javy). Interpretacja jest taka: wywołaj funkcję **hello** z klasy obiektu k (czyli **Klazz**) i *prześlij* k *jako pierwszy argument.* Jest to więc podobnie jak w C++/Javie, choć składnia użyta w linii 15 nie zostałaby zaakceptowana przez kompilatory tych języków. Z wydruku linii 15-16

```
* 3* HelloK, self = <__main__.Klazz object at 0x7fd6aa396da0>
* 4* HelloK as method, self = <__main__.Klazz object at 0x7fd6aa396da0>
```

widzimy, że w obu przypadkach pierwszym argumentem funkcji **hello** jest dokładnie ten sam obiekt.

W pierwszym przypadku funkcja **hello** jest traktowana jak normalna funkcja, tyle że referencja do tej funkcji jest akurat atrybutem klasy. Argument self może być dowolnego typu dla którego ciało funkcji ma sens. Jednak w drugim przypadku, gdy używamy notacji z kropką, funkcja **hello** jest traktowana jako *metoda* klasy. Może być wywołana tylko na rzecz obiektu klasy i referencja do tego obiektu będzie przesłana jako pierwszy argument (odpowiadający mu parametr ma tradycyjnie nazwę self).[1]

W linii 19 definiujemy klasę **Clazz**. Tym razem definicja funkcji **hello** jest zawarta wewnątrz definicji klasy czyniąc ją metodą klasy (jej atrybutem). Będzie ona wywoływana zawsze na rzecz obiektu klasy i referencja do tego obiektu będzie przekazywana do funkcji jako pierwszy argument.

[Uwaga: Nawet wtedy jednak jest możliwe wywołanie tej metody jako niezwiązanej funkcji (linie 26 i 40) przekazując do niej jako pierwszy argument referencję do dowolnego obiektu, jak to widzimy w linii 40. Takie użycie metody jest jednak mylące i powinno być unikane.]

W liniach 29-30 tworzymy dodatkowy atrybut att w obu klasach. Zauważmy, że do tego atrybutu możemy się odwoływać poprzez nazwę klasy, ale również poprzez obiekty klasy, nawet jeśli zostały utworzone *przed* przed modyfikacją klas (linie 33-36).

W liniach 37-38 drukujemy atrybut __dict__ zarówno klasy jak i obiektu tej klasy.[2] Atrybut ten to słownik zawierający nazwy i wartości atrybutów klasy. Jak widzimy z wydruku, metoda **hello** i att są są atrybutami klasy, ale nie obiektu tej klasy. Tym niemniej, będą znalezione, gdyż są wyszukiwane w atrybutach klasy, jeśli nie zostały znalezione wśród atrybutów obiektu.

Klasa może zawierać dowolną liczbę metod. Ale co z polami? Tutaj jest inaczej, niż w takich językach jak C++/Java. W tych językach deklarujemy je w samej definicji klasy i potem będą one obecne w każdym obiekcie klasy (oczywiście z różnymi wartościami). Wszystkie obiekty klasy mają ten sam zestaw pól, ten sam wymiar i taką samą wewnętrzną strukturę. Ale nie w Pythonie. W zasadzie każdy obiekt klasy mógłby mieć zupełnie inne, niezwiązane ze sobą pola (atrybuty), choć w praktyce prowadziłoby to do niemożliwego do opanowanie chaosu.

Aby zdefiniować pola, które mają być obecne we wszystkich obiektach danej klasy, używamy *konstruktora.* W Pytonie jego rolę pełni „magiczna" metoda **__init__**. Zauważmy, że nie ma w Pythonie przeciążania funkcji, więc i konstruktor może być tylko jeden. Tak jak w C++/Javie konstruktor jest wywoływany na samym końcu procesu tworzenia obiektu, kiedy obiekt już istnieje, i referencja do niego jest przekazywana jako pierwszy argument (odpowiadający parametrowi self). Wewnątrz konstruktora możemy *utworzyć* atrybuty (pola) tego obiektu i przypisać im wartości (które zwykle zależą od innych argumentów konstruktora). Ponieważ konstruktor, jeśli został zdefiniowany, będzie wywoływany przy tworzeniu wszystkich obiektów, więc wszystkie będą zawierać tworzone tam pola.

Zauważmy, że nie ma tu żadnej informacji o dostępności składowych klas: wszystkie są w Pythonie publiczne *par naissance.* Oznacza to, że mamy do nich bezpośredni dostęp poprzez notację „z kropką" *(dot notation)*:

[1] W C++/Javie parametr ten nie jest w ogóle wymieniony na liście parametrów, więc nie ma jak nadać mu nazwy; jest ona zatem raz na zawsze ustalona: this.

[2] Moglibyśmy też użyć do tego celu wbudowanej funkcji **vars**.

```
class Person:
    def __init__(self, n, y):
        self.name = n
        self.year = y

john = Person("John", 2001)
mary = Person("Mary", 2005)

print(f'{john.name}({john.year})')
print(f'{mary.name}({mary.year})')
```

drukuje

```
John(2001)
Mary(2005)
```

W tym przykładzie nie ma konfliktu nazw między nazwami pól a nazwami parametrów. W takiej sytuacji w C++/Javie nie musielibyśmy dodawać this przy nazwach pól — kompilator dodałby to automatycznie. Mógłby to zrobić, bo już zna nazwy pól podane w definicji klasy. W Pythonie jednak nie zostały one nigdzie zadeklarowane. Zatem referencja self w `self.name = n` jest zawsze konieczna. Ogólnie, gdziekolwiek w C++/Javie this jest obecne z założenia, ale może być opuszczone, w Pythonie self musi być podane jawnie.

11.2 Tworzenie obiektów

Nie mówiliśmy jeszcze o dziedziczeniu, ale pamiętamy, że w C++/Javie konstruktor klasy bazowej jest zawsze wywoływany podczas tworzenia obiektu klasy pochodnej (i to zanim przepływ sterowania wejdzie do konstruktora klasy pochodnej). Jak to jest w przypadku Pythona?[1] Tu sytuacja jest inna:

- Jeśli nasza klasa *nie* definiuje metody `__init__`, to konstruktor klasy bazowej *będzie* wywołany;
- Jeśli nasza klasa *definiuje* metodę `__init__`, to konstruktor klasy bazowej *nie* będzie wywoływany automatycznie, choć możemy go wywołać „ręcznie” z konstruktora klasy pochodnej poprzez użycie super, jak w przykładzie poniżej.[2]

Listing 61 BBD-Ctors/Ctors.py

```
#!/usr/bin/env python

class Base:
    def __init__(self):
        print('init of Base')

class DerivNoInit(Base):
    pass

```

[1] W Pythonie, tak jak w Javie (ale nie w C++), każda klasa dziedziczy, pośrednio lub bezpośrednio, z klasy object.

[2] W Pythonie klasę bazową podajemy w nawiasach zaraz za nazwą definiowanej klasy pochodnej.

```
10 class DerivWithInit(Base):
11     def __init__(self):
12         print('init of DerivWithInit')
13
14 class DerivCallingBase(Base):
15     def __init__(self):
16         super().__init__()
17         print('init of DerivCallingBase')
18
19
20 print('Creating DerivNoInit object')
21 dn = DerivNoInit()
22
23 print('\nCreating DerivWithInit object')
24 dw = DerivWithInit()
25
26 print('\nCreating DerivCallingBase object')
27 dc = DerivCallingBase()
```

Program drukuje

```
Creating DerivNoInit object
init of Base

Creating DerivWithInit object
init of DerivWithInit

Creating DerivCallingBase object
init of Base
init of DerivCallingBase
```

Zauważmy, że wywołując **super** w konstruktorze klasy pochodnej nie musimy tego robić koniecznie w pierwszej linii, równie dobrze możemy to zrobić na samym końcu.

Tak jak w C++/Javie, tworzenie obiektu jest procesem dwustopniowym. Najpierw musi być zaalokowana pamięć i utworzony obiekt. Następnie, gdy obiekt już istnieje, na jego rzecz jest wywoływany konstruktor, którego zadaniem jest „skonfigurowanie" obiektu. Zazwyczaj krok pierwszy jest realizowany przez funkcję **__new__** z klasy **object**, po czym **__init__** jest wywoływana na rzecz właśnie skonstruowanego obiektu. Możemy jednak oddzielić te dwa kroki wywołując **__new__** i **__init__** „ręcznie":

```
class Person:
    def __init__(self, n):
        self.name = n
        print('init called')

print('Calling new')
p = object.__new__(Person)
print('dict of p:', p.__dict__)   # empty
```

```
print('Calling init')
p.__init__('John')
print('dict of p:', p.__dict__)    # now has name

print(p.name)
```

co drukuje

```
Calling new
dict of p: {}
Calling init
init called
dict of p: {'name': 'John'}
John
```

Zauważmy, że funkcja **__new__** pobiera jako argument referencję do klasy, obiekt której ma stworzyć (odpowiedni parametr ma konwencjonalnie nazwę cls).

Możemy też sami zdefiniować metodę **__new__** w naszej klasie. Wtedy metoda **__new__** z klasy **object** nie będzie wywołana automatycznie — musimy to zrobić sami i zwrócić utworzony obiekt; na rzecz tego obiektu będzie automatycznie wywołany konstruktor **__init__**:

```
class Person:
    def __new__(cls, *args, **kwargs):
        print('Creating object')
        ob = object.__new__(cls)
        return ob

    def __init__(self, name):
        print('init called')
        self.name = name

p = Person('John')
print(p.name)
```

prints

```
Creating object
init called
John
```

Zauważmy sygnaturę funkcji **__new__**: jej pierwszym argumentem jest klasa (referencja do obiektu reprezentującego klasę), ale potem następuje `*args, **kwargs`, co w zasadzie znaczy „cokolwiek”. Zwykle funkcja **__new__** nie używa tych argumentów, ale są one przekazywane do **__init__** jako pozostałe argumenty (po referencji do utworzonego obiektu, jako self), co ilustruje poniższy przykład:

Listing 62 BBG-NewInit/NewInit.py

```
1 #!/usr/bin/env python
2
3 class AClass:
4     def __new__(cls, *args, **kwargs):
```

```
        print('__new__')
        ob = object.__new__(cls)
        return ob

    def __init__(self, *args, **kwargs):
        print('__init__', end=': ')
        if len(args) == 0:
            print('No var args;', end=' ')
        else:
            print('Var args:', end=' ')
            for i, v in enumerate(args):
                print(f'{i+1} -> {v};', end=' ')
        if len(kwargs) == 0:
            print('No kwargs;')
        else:
            print('KWargs:', end=' ')
            for k, v in kwargs.items():
                print(f'{k} : {v}; ', end = ' ')
            print()

o1 = AClass()
o2 = AClass(1, 2)
o3 = AClass(1, 2, name='John', age=27)
o4 = AClass(name='John', age=27)
```

Program drukuje

```
__new__
__init__: No var args; No kwargs;
__new__
__init__: Var args: 1 -> 1; 2 -> 2; No kwargs;
__new__
__init__: Var args: 1 -> 1; 2 -> 2; KWargs: name : John;  age : 27;
__new__
__init__: No var args; KWargs: name : John;  age : 27;
```

W zasadzie, jeśli już definiujemy sami metodę **___new___**, to i tak tworzymy w niej obiekt, więc możemy go od razu skonfigurować używając argumentów przesłanych do **___new___** i obyć się bez funkcji **___init___**:

```
class Person:
    def __new__(cls, name):
        ob = object.__new__(cls)
            # lub: ob = super().__new__(cls)
        ob.name = name
        return ob

p = Person('John')
print(p.name)
```

co drukuje

```
John
```

11.3 Pola statyczne i niestatyczne

W przeciwieństwie do C++/Javy, pola niestatyczne (a więc należące do poszczególnych obiektów) *nie* są częścią definicji klasy. Zazwyczaj są one tworzone w konstruktorze (metodzie __init__), choć, jak już wiemy mogą też być dodawane do już istniejących obiektów:

```
>>> class AClass:
...     def __init__(self, attr1):
...         self.attr1 = attr1
...
>>> ob1 = AClass('a')
>>> ob2 = AClass(1)
>>> ob1.attr1, ob2.attr1
('a', 1)
>>> ob1.attr2 = 'b'  # ob1 ma dodatkowe pole
>>>
>>> ob1.__dict__
{'attr1': 'a', 'attr2': 'b'}
>>> ob2.__dict__
{'attr1': 1}
>>> ob1.attr2
'b'
>>> ob2.attr2        # ale nie ob2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'AClass' object has no attribute 'attr2'.
```

Jak widzimy, obiekty (dowolnej klasy) mają specjalny atrybut __dict__, który jest słownikiem zawierającym nazwy i wartości jego atrybutów (pól) dodanych do obiektu. Te słowniki mogą być różne dla różnych obiektów tej samej klasy, choć oczywiście lepiej jest, gdy wszystkie obiekty klasy mają te same atrybuty tych samych typów (z różnymi wartościami).

Powyższy przykład ilustruje też inną cechę pól niestatycznych klasy — w obiekcie ob1 pole attr1 jest typu **str**, natomiast w obiekcie ob2 jest on typu **int**. Taka sytuacja, choć dopuszczalna, powinna być unikana.

Klasy też są reprezentowane przez obiekty, a zatem te obiekty mogą mieć swoje atrybuty, nienależące do poszczególnych instancji tej klasy ale raczej do samej klasy. Takie pola (jak w C++/Javie), nazywamy polami statycznymi lub klasowymi. Te pola *są* częścią definicji klasy[1] i jak w C++/Javie, są dostępne za pomocą notacji z kropką zarówno poprzez klasę jak i obiekty tej klasy:

```
>>> class AClass:
...     statAttr = 42
...
>>> a = AClass()
>>> AClass.statAttr
42
>>> a.statAttr
42
```

[1] Znowu, jest możliwe dodawanie takich pól dynamicznie w czasie wykonania, ale nie jest zazwyczaj dobry pomysł.

Statyczne pola są zainicjowane od razu w definicji klasy; inaczej się zresztą nie da, bo w Pythonie nie podaje się typów, więc jedynym sposobem na „powiedzenie" interpreterowi jaki jest ich typ jest podanie inicjatora.
Jako że obiekty reprezentujące klasy są modyfikowalne, można dodawać do nich atrybuty dynamicznie podczas wykonania. Gdy się do nich odnosimy, aktualna wersja klasy będzie użyta:

```
>>> class AClass:
...     statAttr1 = 1
...
>>> a = AClass()
>>> a.statAttr1
1
>>> AClass.statAttr2 = 2
>>> a.statAttr2
2
```

Przykład ten pokazuje, że obiekt a „widzi" pole statyczne statAttr2 choć dodane ono zostało do klasy już *po* utworzeniu samego obiektu a. To jest zrozumiałe, bo statAttr1 i statAttr1 „należą" do klasy, nie do jej obiektów, a klasa została zmodyfikowana. Możemy sprawdzić, co należy do czego za pomocą funkcji **vars**, która zwraca atrybut __dict__:

```
>>> class AClass:
...     def __init__(self, a):
...         self.a = a
...     statAttr = 7
...
>>> a = AClass(3)
>>> vars(a)
{'a': 3}
>>> vars(AClass)
mappingproxy({'__module__': '__main__',
              '__init__': <function AClass.__init__ at 0x7fca27909b40>,
              'statAttr': 7, etc. .... })
```

Do pól klasowych (statycznych) możemy się odwoływać poprzez nazwę klasy lub obiektu. Jednak jeśli chcemy zmodyfikować taki atrybut, tylko pierwsza forma może być zastosowana, ponieważ a.statAttr=3 utworzyłoby raczej dodatkowe pole niestatyczne w obiekcie, a nie zmieniłoby atrybutu klasy.

Zobaczmy to na jeszcze jednym przykładzie. Tworzymy tu (❶) prostą klasę z dwoma atrybutami (polami statycznymi), a i b. W linii ❷ tworzymy też obiekt tej klasy, x.

Listing 63 BBC-Attrs/Attrs.py

```
1 #!/usr/bin/env python
2
3 class Klass:                    ❶
4     a = 'a'
5     b = 'b'
6
7 x = Klass()                     ❷
```

```
print('**1**', vars(Klass))
print('**2**', vars(x))
print('**3**', x.a, x.b)
Klass.a = 'aa'                    ❸
print('**4**', vars(x))
print('**5**', x.a, x.b)
y = Klass()
y.b = 'bb'                        ❹
print('**6**', x.a, x.b)
print('**7**', y.a, y.b)
print('**8**', vars(x))
print('**9**', vars(y))
```

Z wydruku następnych trzech linii

```
**1** {'__module__': '__main__', 'a': 'a', 'b': 'b',
       '__dict__': <attribute '__dict__' of 'Klass' objects>,
       '__weakref__': <attribute '__weakref__' of 'Klass' objects>,
       '__doc__': None}
**2** {}
**3** a b
```

widzimy, że klasa ma atrybuty a i b, ale obiekt x ich nie ma; ponieważ ich nie ma, x.a i x.b znalezione zostanie w obiekcie klasy. W linii ❸ modyfikujemy atrybut a klasy. Ponieważ obiekt x nie ma „własnego” a, x.a w dalszym ciągu odwołuje się do (zmienionego) atrybutu klasy

```
**4** {}
**5** aa b
```

Teraz tworzymy jeszcze jeden obiekt klasy, y, i przypisujemy wartość 'bb' do y.b (❹). Atrybut b w tym obiekcie nie istnieje, ale jest to przypisanie, więc zostanie utworzony! Teraz x.a i x.b w dalszym ciągu odwołują się do atrybutów klasy. Tak też jest dla y.a, ale y.b teraz jest atrybutem obiektu y, niezależnym od Klass.b:

```
**6** aa b
**7** aa bb
**8** {}
**9** {'b': 'bb'}
```

11.4 Metody niestatyczne, statyczne i klasowe

11.4.1 Metody niestatyczne

Metody niestatyczne klasy są definiowane wewnątrz jej definicji jako normalne funkcje. Wszystkie muszą mieć przynajmniej jeden parametr, odpowiadający referencji obiektu na rzecz którego metoda będzie wywołana (ten parametr ma konwencjonalnie nazwę self) i jest analogiem tego, co w C++/Javie nazywa się this (w tych językach parametru tego w ogóle nie wymienia się na liście parametrów, dlatego jego nazwa jest ustalona). Zdefiniowana metoda staje się atrybutem klasy i możemy się do niej odwołać poprzez notację „z kropką” na rzecz klasy, podając jako pierwszy argument referencję do jakiegoś jej obiektu, która będzie dostępna w ciele metody jako self. Zwykle

jednak używamy tradycyjnej składni — wywołujemy metodę „na rzecz" obiektu (nie klasy) i wtedy referencja do tego obiektu jest przekazywana do funkcji jako pierwszy argument automatycznie:

```
>>> class AClass:
...     def __init__(self, at):
...         self.attr = at                          # 1
...     def info(self):
...         print('Object with attr =', self.attr) # 2
...
>>> a = AClass('hello')
>>> AClass.info(a)          # 3
Object with attr = hello
>>> a.info()                # 4
Object with attr = hello
```

W linii oznaczonej `'# 3'` wywołujemy funkcję **info** z klasy **AClass**, przekazując jako pierwszy argument obiekt tej klasy. W następnej linii, oznaczonej `'# 4'` wywołujemy tę samą metodę na rzecz obiektu `a` bez argumentów; referencja `a` będzie niejawnie przekazana jako pierwszy argument. Te dwie formy wywołania są w Pythonie równoważne, choć druga jest oczywiście stosowana częściej (w C++/Javie tylko druga byłaby składniowo poprawna).

Zauważmy, że wewnątrz **__init__**, (linia `'# 1'`) a także w funkcji **info** (linia `'# 2'`), self jest wymagane: w C++/Javie moglibyśmy w tym kontekście opuścić this.

11.4.2 Metody klasowe

Metody klasowe (nie mylić ze statycznymi!) nie mają odpowiednika w C++/Javie. Mają one, tak jak „zwykłe", niestatyczne metody, pierwszy argument szczególny. Dla zwykłych metod jest to referencja do obiektu na rzecz którego metodę wywołujemy (i jest „odbierana" jako self); dla metod klasowych jest to referencja do *klasy* i jest „odbierana" jako pierwszy parametr nazwany, na zasadzie konwencji, cls.

Definiujemy taką metodę za pomocą dekoratora **classmethod**. Pierwszy parametr będzie odpowiadał *klasie* na rzecz której metoda jest wywoływana. W zasadzie można ją wywołać na rzecz obiektu, ale, tak jak dla metod statycznych w C++/Javie, liczy się wtedy tylko typ (klasa) tego obiektu, a nie sam obiekt:

```
class AClass:
    @classmethod
    def fun(cls, arg):
        print(cls, 'arg =', arg)

a = AClass()
AClass.fun(5)
a.fun('Hello')
```

drukuje

```
<class '__main__.AClass'> arg = 5
<class '__main__.AClass'> arg = Hello
```

Jak widzimy, w obu przypadkach pierwszy argument jest taki sam: referencja do klasy, a nie do obiektu tej klasy.

Metody klasowe są często używane jako metody fabrykujące, które, w pewnym sensie, dostarczają dodatkowych konstruktorów. Może to być użyteczne, gdyż „właściwy” konstruktor (__init__) nie może być w Pythonie przeciążany, tak jak może być w C++/Javie.

Rozpatrzmy następujący program:

Listing 64 BBJ-ClassMeth/ClassMeth.py

```
#!/usr/bin/env python

from math import pi, sqrt, sin, cos, isclose

class Point:
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    @classmethod
    def from_polar(cls, r, phi):                    ❶
        return cls(r*cos(phi), r*sin(phi))

    @classmethod
    def from_complex(cls, z):                       ❷
        return cls(z.real, z.imag)

    def __eq__(self, other):                        ❸
        return (isclose(self.x, other.x)
            and isclose(self.y, other.y))

aux = 1/sqrt(3)
z = complex(1, aux)

p1 = Point(1, aux)
p2 = Point.from_polar(2*aux, pi/6)
p3 = p1.from_complex(z)                             ❹

print(p1 == p2, p2 == p3, p1 == p3) # output: True True True
```

Klasa Point opisuje punkt na płaszczyźnie o dwóch współrzędnych. Naturalny konstruktor pobiera zatem dwie liczby i tworzy dwa niestatyczne pola x i y. Chcielibyśmy mieć inny konstruktor, który pobierałby współrzędne biegunowe punktu, r i φ, ale mamy problem: to są też dwie liczby, więc jak konstruktor __init__ miałby rozpoznać, czy są to x i y, czy r i φ? Definiujemy więc metodę klasową, from_polar (❶), która, prócz cls, pobiera dwie liczby, interpretuje je jako r i φ punktu, tworzy i zwraca obiekt klasy Point odpowiadający tym współrzędnym.

Inna metoda klasowa, from_complex (❷), robi to samo, ale na podstawie podanej liczby zespolonej.

W linii ❸ definiujemy „magiczną” metodę __eq__, aby działało porównywanie z ostatniej linii (więcej o „magicznych” metodach w rozdz. 11.8, str. 236).

Linia ❹ ilustruje fakt, że metoda klasowa może też być wywołana na rzecz obiektu, choć wtedy znaczenia ma tylko jego typ.

Tworząc obiekty w obu metodach klasowych używaliśmy cls zamiast jawnie **Point**. To jest zamierzone. Przypuśćmy, że jest klasa dziedzicząca z **Point**. Będziemy wtedy mogli wywoływać te odziedziczone metody na rzecz klasy dziedziczącej (lub jej obiektów) i automatycznie będą one zwracać obiekty tejże klasy dziedziczącej, a nie klasy **Point**.

11.4.3 Metody statyczne

Metody statyczne oznaczane są dekoratorem **staticmethod** i zachowują się podobnie do statycznych metod w C++/Javie. Gdy je wywołujemy, nie ma żadnych „ukrytych" argumentów: wszystkie argumenty, łącznie z pierwszym, odpowiadają kolejnym parametrom metody i żaden z nich nie ma jakiegoś szczególnego znaczenia. Można je wywoływać poprzez nazwę klasy, jak również na rzecz obiektu: w obu przypadkach nie ma żadnego dodatkowego argumentu prócz tych jawnie wymienionych.

W przykładzie poniżej mamy w klasie **Point** dwie metody statyczne. Jedna to **heronArea**, która, korzystając ze wzoru Herona, oblicza pole trójkąta mając współrzędne trzech wierzchołków. Aby to zrobić, potrzebne są jednak długości boków: aby je znaleźć, używamy innej funkcji statycznej **dist**.

Listing 65 BBM-StatMeth/StatMeth.py

```
#!/usr/bin/env python

import math

class Point:
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    @staticmethod
    def heronArea(p1, p2, p3):
        a = Point.dist(p1, p2)          ❶
        b = Point.dist(p2, p3)
        c = Point.dist(p3, p1)
        s = (a + b + c)/2
        return math.sqrt(s*(s-a)*(s-b)*(s-c))

    @staticmethod
    def dist(p1, p2):
        return math.hypot(p1.x-p2.x, p1.y-p2.y)

p1 = Point(1, 1)
p2 = Point(4, 1)
p3 = Point(4, 5)

print(Point.heronArea(p1, p2, p3)) # output: 6.0
```

Zauważmy, że w linii ❶ musimy kwalifikować nazwę funkcji **dist** nazwą klasy do której

należy — byłoby to niepotrzebne w C++/Javie. W Pythonie jednak jest niezbędne. Pamiętamy bowiem o regule LEGB (patrz rozdz. 5.1, str. 63): nazwa **dist** będzie szukana (i nie zostanie znaleziona) w zakresie lokalnym, potem w zakresie otaczającej funkcji, której tu jednak nie ma, a następnie w zakresie globalnym (zakresie modułu). Funkcja **dist** *nie* jest w zakresie globalnym — co jednak w nim jest to sama klasa **Point**, której atrybutem jest w końcu nasza szukana **dist**.

Jak z tego widać, podobny efekt uzyskalibyśmy definiując **dist** jako „zwykłą" funkcję w zakresie globalnym. Byłoby to nawet łatwiejsze, bo nie musielibyśmy kwalifikować jej nazwy w linii ❶. Z tego też powodu metody statyczne są w Pythonie używane raczej rzadko.

W przykładzie poniżej definiujemy klasę **MyDate**, której obiekty reprezentują daty. Statyczne metody **today**, **yesterday** i **tomorrow** nie musiały tu być statyczne, ale możemy je takimi uczynić, chcąc podkreślić ich ścisły związek z klasą **MyDate**:

Listing 66 ABJ-Static/Dates.py

```
#!/usr/bin/env python

import time

class MyDate:
    def __init__(self, year, month, day):       # constructor
        self.year = year
        self.month = month
        self.day = day

    def __repr__(self):
        return f'{self.day:02}/{self.month:02}/{self.year:4}'

    @staticmethod
    def today():
        t = time.localtime()
        return MyDate(t.tm_year, t.tm_mon, t.tm_mday)

    @staticmethod
    def tomorrow():
        t = time.localtime(time.time() + 86400)
        return MyDate(t.tm_year, t.tm_mon, t.tm_mday)

    @staticmethod
    def yesterday():
        t = time.localtime(time.time() - 86400)
        return MyDate(t.tm_year, t.tm_mon, t.tm_mday)


coronation = MyDate(1953, 6, 2) # calling constructor
yesterday  = MyDate.yesterday() # static factory methods
today      = MyDate.today()     #        ./.
tomorrow   = MyDate.tomorrow()  #        ./.
print("The Queen's coronation:", coronation)
```

```
print("Yesterday:              ", yesterday)
print("Today:                  ", today)
print("Tomorrow:               ", tomorrow)
```

Program drukuje

```
The Queen's coronation: 02/06/1953
Yesterday:              20/02/2024
Today:                  21/02/2024
Tomorrow:               22/02/2024
```

11.5 „Kacze” typowanie

Python wspiera tak zwane „kacze typowanie” *(duck typing)*. Interpreter nie sprawdza typu argumentów metod/funkcji, zakładając, że programista wie co robi. Kiedy wywołujemy metodę na rzecz obiektu, to to co się liczy, to fakt, czy w klasie tego obiektu jest taka metoda, czy nie. Ilustruje tę cechę poniższy przykład.

Listing 67 ABD-Duck/Duck.py

```
#!/usr/bin/env python

class A:
    def __init__(self, data):
        self.data = data
    def getData(self):                  ❶
        return self.data

class B:
    def __init__(self, something):
        self.something = something
    def getData(self) :                 ❷
        return self.something

def fun(ob):
    return ob.getData()                 ❸

print(fun( A('Lucy') ))                 ❹
print(fun( B(28)     ))                 ❺
```

Funkcja **fun** pobiera obiekt (tu jest to ob) i „na ślepo” wywołuje na jego rzecz metodę **getData** (❸). Funkcji nie obchodzi czy ob jest typu **A** (❹) czy zupełnie niezwiązanego z nim typu **B** (❺). Jeśli się tak złożyło, że oba te typy rzeczywiście mają metodę **getData**, program działa bez żadnych skarg i drukuje

```
Lucy
28
```

Nie jest to możliwe w C++/Javie, choć w C++ zbliżony efekt można osiągnąć za pomocą szablonów.

11.6 Właściwości (properties)

Nie ma w Pythonie modyfikatorów dostępu, jak **private** czy **public** w C++/Javie — wszystko jest w zasadzie publiczne. To stwarza pewne problemy: wszystkie składowe klas, w szczególności pola, są dostępne dla kodu klienta bez żadnych ograniczeń — mogą być dowolnie modyfikowane, czy nawet usuwane. W językach takich ja C++/Java pola (dane) są zwykle prywatne, a udostępniamy tylko publiczne metody, które operują na tych danych w sposób kontrolowany. Jest to bardzo cenna cecha tych języków i chcielibyśmy móc przynajmniej jakość symulować podobne zachowanie w Pythonie.

Jedna z metod polega na „dżentelmeńskiej umowie", że wszystkie referencje, których nazwa zaczyna się pojedynczym podkreślnikiem traktujemy jako chronione (lub prywatne). Kod klienta, na zasadzie konwencji, „udaje", że nic nie wie nawet o ich istnieniu. Stosując tę konwencję skłaniamy użytkownika do stosowania metod, których nazwa *nie* zaczyna się od podkreślnika (a więc są „naprawdę" publiczne). W ten sposób możemy mieć składowe (atrybuty) chronione przez odpowiednie gettery/settery, jak w innych językach.

Listing 68 BBT-GetSet/GetSet.py

```
1  #!/usr/bin/env python
2
3  class Point:
4      def __init__(self, x, y):
5          self._x = x
6          self._y = y
7
8      def getX(self):
9          return self._x
10
11     def setX(self, v):
12           # validation in setter
13         if v < 0: raise ValueError('x < 0 !!!')  ❶
14         self._x = v
15
16     def getY(self):
17         return self._y
18
19     def setY(self, v):
20         self._y = v
21
22 p = Point(2, 5)
23 p.setX(7)
24 print(p.getX(), p.getY())    # output: 7 5
25
26   # but if you are not a gentleman...
27 p._x = -9                                  ❷
28 print(p._x, p._y)            # output: -9 5
```

Settery mogą być użyte do jakiejś kontroli, jak walidacja danych (jak w linii ❶). Oczy-

wiście tak zaimplementowaną „prywatność” łatwo obejść, jak to widzimy w linii ❷.

Bardzie „pytonicznym” sposobem na tego rodzaju problemy jest użycie *właściwości*. Właściwości pozwalają na tworzenie metod, które zachowują się jak („publiczne”) atrybuty (pola). Na przykład w powyższej klasie **Point** moglibyśmy zamienić pola x i y na właściwości i użytkownik tej klasy wciąż mógłby się odnosić do nich w ten sam sposób, nie wiedząc, że teraz są to wywołania metod. Zaletą właściwości jest właśnie to, że pozwala udostępniać publiczne API klasy, ale zachować możliwość dowolnych zmian w implementacji.

Rozpatrzmy następujący przykład. Klasa **Square** ma jedno pole, _a, które jest prywatne. Definiujemy dla niego getter (❶) i setter (❷), który implementuje prostą walidację. Jest nawet deleter (do usuwania atrybutu) — dla ilustracji, bo zwykle go nie potrzebujemy. Następnie definiujemy getter (❹) i setter (❺) dla pola area. W zasadzie takie pole nie istnieje — obie te metody dostępowe korzystają tak naprawdę z pola _a. Obie też są prywatne, jako że ich nazwa zaczyna się podkreślnikiem

Listing 69 BBP-Props/Props.py

```
1  #!/usr/bin/env python
2
3  from math import sqrt
4
5  class Square:
6      def __init__(self, a):
7          self._a = float(a)
8
9      def _getA(self):                                    ❶
10         return self._a
11
12     def _setA(self, new_a):                             ❷
13         if new_a < 0:
14             raise ValueError('No negative sides')
15         self._a = float(new_a)
16
17     def _delA(self):                                    ❸
18         del self._a
19
20     def _getArea(self):                                 ❹
21         return self._a**2
22
23     def _setArea(self, new_area):                       ❺
24         if new_area < 0:
25             raise ValueError('No negative areas!!!')
26         self._a = sqrt(new_area)
27
28     a    = property(_getA, _setA,                       ❻
29                     _delA, "Side of the square.")
30     area = property(_getArea, _setArea,                 ❼
31                     None, "Area of the square.")
32
```

```
33
34 print(type(Square.a), type(Square.area))          ❽
35 sq = Square(5)
36 print(f'Square: a={sq.a}, area={sq.area}')       ❾
37 sq.area = 16                                     ❿
38 print(f'Square: a={sq.a}, area={sq.area}')
39 sq.a = 7
40 print(f'Square: a={sq.a}, area={sq.area}')
41 print(f'Square\'s area doc-string: {Square.area.__doc__}')
```

Jak na razie nie ma w klasie żadnego publicznego API. Ale teraz następuje coś interesującego. Tworzymy dwa *publiczne,* statyczne atrybuty, `_a` (❻) i `area` (❼) typu **property**. Z punktu widzenia użytkownika, będą się one zachowywać jak publiczne pola *niestatyczne.*

Jak widzimy z wydruku

```
<class 'property'> <class 'property'>
Square: a=5.0, area=25.0
Square: a=4.0, area=16.0
Square: a=7.0, area=49.0
Square's area doc-string: Area of the square.
```

ich typem jest klasa **property**, której konstruktor ma postać

```
property(fget=None, fset=None, fdel=None, doc=None)
```

i pobiera getter, setter, deleter i napis dokumentujący dany atrybut (setter, deleter i napis dokumentujący mogą być opuszczone, lub **None**, co jest wartością domyślną). Ale te pola (nazywane *właściwościami zarządzanymi*) są polami statycznymi klasy **Square**, jak więc mogą odpowiadać niestatycznym atrybutom obiektów?

Właściwości używamy zawsze na rzecz konkretnych obiektów, jak w liniach ❾ i ❿. Referencja do tych obiektów (w przykładzie jest to `sq`) jest przekazywana jako pierwszy argument do metody obiektu typu **property** i na rzecz tego właśnie obiektu będzie wywołany getter lub setter, który przesłaliśmy do konstruktora właściwości — setter, jeśli jest to przypisanie do właściwości, a getter jeśli pobieramy wartość tej właściwości.[1]

Jest też inny sposób na tworzenie właściwości zarządzanych. Można użyć dekoratorów — i to jest sposób rekomendowany. Zilustrujemy go przepisując inaczej przykład z Listing 69.

Listing 70 BBW-PropsDec/PropsDec.py

```
1 #!/usr/bin/env python
2
3 from math import sqrt
4
5 class Square:
6     def __init__(self, a):
7         self._a = float(a)
8
```

[1] Lub deleter, jeśli jest argumentem operatora **del**.

```
9       @property                                    ❶
10      def a(self):
11          """Side of the square."""
12          return self._a
13
14      @a.setter                                    ❷
15      def a(self, new_a):
16          if new_a < 0:
17              raise ValueError('No neg. sides')
18          self._a = float(new_a)
19
20      @a.deleter                                   ❸
21      def a(self):
22          del self._a
23
24      @property
25      def area(self):                              ❹
26          """Area of the square."""
27          return self._a**2
28
29      @area.setter
30      def area(self, new_area):                    ❺
31          if new_area < 0:
32              raise ValueError('No neg. areas!!!')
33          self._a = sqrt(new_area)
34
35  print(type(Square.a), type(Square.area))
36  sq = Square(5)
37  print(f'Square: a={sq.a}, area={sq.area}')
38  sq.area = 16
39  print(f'Square: a={sq.a}, area={sq.area}')
40  sq.a = 7
41  print(f'Square: a={sq.a}, area={sq.area}')
42  print(f'Square\'s area doc-string: {Square.area.__doc__}')
```

Tworzymy właściwość nazwaną `a` w linii ❶. Zauważmy, że `a` będzie nazwą *właściwości*, a nie funkcji. Implementacja musi odpowiadać getterowi, a napis dokumentujący będzie napisem dokumentującym tę właściwość (a nie sam getter). Mamy już teraz właściwość `a` z samym getterem, co odpowiada polu tylko do odczytu. Przypuśćmy, że chcemy mieć również setter i deleter.[1]

Aby dodać setter do już istniejącej właściwości `a`, dekorujemy go dekoratorem **`@a.setter`** (❷), gdzie `a` jest nazwą tej właściwości. Zauważmy, że nazwa za słowem kluczowym **`def`** musi być dokładnie taka sama jak nazwa dekorowanej właściwości, w naszym przypadku `a`. To samo stosuje się do deletera (❸).

W podobny sposób tworzymy właściwość `area`, tym razem bez deletera – ❹ i ❺.

Wydruk programu jest dokładnie taki sam jak programu Listing 69:

```
<class 'property'> <class 'property'>
```

[1] Ale najpierw zadajmy sobie pytanie, czy rzeczywiście ich potrzebujemy...

```
Square: a=5.0, area=25.0
Square: a=4.0, area=16.0
Square: a=7.0, area=49.0
Square's area doc-string: Area of the square.
```

11.7 Reprezentacja napisowa obiektów

Bardzo często chcemy wydrukować informacje o obiekcie w postaci napisu (i ogólnie, otrzymać napisową reprezentację obiektu). Ten problem rozwiązuje w Javie nadpisanie metody **toString**, a w C++ przeciążenie funkcji **operator<<**. W Pytonie są nawet dwie metody, które możemy nadpisać w podobnym celu: **__str__** and **__repr__**.

Tak jak w przypadku innych metod *dunder*, prawie nigdy nie wywołujemy ich „po nazwie". Jeśli naszym obiektem jest ob, wywołanie `str(ob)` wywoła automatycznie na rzecz ob metodę **__str__**, a wywołanie `repr(ob)` metodę **__repr__**.

W zasadzie możemy wywoływać te funkcje bez definiowania żadnych metod w naszej klasie

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

john = Person("John", 2001)

print(str(john))
print(repr(john))
```

i dostać

```
<__main__.Person object at 0x7f8ba1e07a00>
<__main__.Person object at 0x7f8ba1e07a00>
```

Taka jest domyślna implementacja tych metod odziedziczona z klasy **object** — rzadko użyteczna, tak zresztą jak domyślna implementacja **toString** w Javie. Aby dostać coś bardziej przydatnego, możemy zdefiniować w naszej klasie jedną lub obie metody **__str__** i **__repr__**. Jeśli zdefiniujemy tylko **__repr__**, to będzie ona użyta również dla `str(ob)`:

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

    def __repr__(self):
        return 'from repr'

john = Person("John", 2001)

print(str(john))
print(repr(john))
```

drukuje

```
from repr
from repr
```

Jednakże, jeśli zdefiniujemy tylko metodę **__str__**

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

    def __str__(self):
        return 'from str'

john = Person("John", 2001)

print(str(john))
print(repr(john))
```

to dostaniemy

```
from str
<__main__.Person object at 0x7f1242a13a90>
```

czyli *nie* będzie ona użyta przy wywołaniu `repr(ob)`. Wynika z tego, że we własnych klasach powinniśmy raczej nadpisywać metodę **__repr__**, a jeśli chcemy mieć inną reprezentację dla wywołań **str**, to dodatkowo metodę **__str__**.

Dlaczego mamy dwie metody dające napisową reprezentację obiektów? Chodzi o to, że przeznaczone są dla różnych adresatów. Rezultat metody **__str__** jest z założenia wygodny dla czytelnika programu: powinien to być napis niekoniecznie bardzo precyzyjny, ale zrozumiały i łatwy do zinterpretowania. Natomiast to co zwraca **__repr__** jest bardziej przeznaczone dla autora kodu — ma być precyzyjne i użyteczne przy debugowaniu. Jeśli to możliwe, zwracany napis powinien pozwalać na dokładne zreprodukowanie obiektu:

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

    def __repr__(self):
        return f'Person("{self.name}", {self.year})'

john = Person("John", 2001)

print('*1*', john)      # repr will be used

p = eval(repr(john))
print('*2*', type(p))
print('*3*', p)
```

drukuje

```
*1* Person("John", 2001)
*2* <class '__main__.Person'>
*3* Person("John", 2001)
```

(We used here the built-in **eval** function which evaluates its argument as a Python code.)

Obiekty są też konwertowane na napisy podczas formatowania. Na przykład **print** używa **__str__**, jak to czyni również `{object}` w f-stringu. Jeśli wolelibyśmy raczej reprezentację otrzymywaną przez **__repr__**, to trzeba dodać flagę konwersji 'r': `{object!r}`. Dla formy ze znakiem równości, `{object=}`, jest na odwrót: domyślnie będzie użyte **__repr__**, a jeśli chcemy żeby to było **__str__**, musimy dodać flagę 's': `{object=!s}`:

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

    def __repr__(self):
        return f'Person("{self.name}", {self.year})'

    def __str__(self):
        return f'{self.name}({self.year})'

john = Person("John", 2001)

print('*1*', john)
print('*2*', repr(john))
print('*3*', f'{john}')
print('*4*', f'{john!r}')
print('*5*', f'{john=}')
print('*6*', f'{john=!s}')
```

drukuje

```
*1* John(2001)
*2* Person("John", 2001)
*3* John(2001)
*4* Person("John", 2001)
*5* john=Person("John", 2001)
*6* john=John(2001)
```

11.8 Magic methods

Istnieje w Pytonie wiele tak zwanych metod „magicznych". Ich nazwy zaczynają się i kończą dwoma podkreślnikami, więc są kolokwialnie nazywane metodami „dunder" *(double underscore)*. Są o tyle specjalne, że prawie nigdy nie są wywoływane bezpośrednio, ale raczej pośrednio w pewnych kontekstach: na przykład gdy operator binarny pojawia się pomiędzy dwoma obiektami lub operator unarny stoi przed obiektem, lub przez najrozmaitsze funkcje wbudowane, jak **iter**, **next**, **len**, i wiele innych, a także gdy obiekt jest indeksowany lub traktowany jak funkcja (z listą argumentów w nawiasach okrągłych).

Niektóre z tych metod już napotkaliśmy: **__iter__**, **__next__**, **__getitem__** (rozdz. 2.5, str.24), **str** i **__repr__** (rozdz. 11.7, str.234), **new** i **__init__** (rozdz. 11.2, str.218), i kilka innych.

11.8.1 Przeciążanie operatorów

Jak pamiętamy, w C++ możemy przeciążać operatory poprzez zdefiniowanie specjalnych metod (lub funkcji wolnych) o nazwach **operator+**, **operator−** itd. Aby podobny efekt uzyskać w Pytonie, definiujemy „magiczne” metody (ale nie funkcje globalne), takie jak **__add__**, **__sub__**, **__mul__**, etc. Tak jak w C++, są pewne ograniczenia

- Nie można przeciążać operatorów dla typów wbudowanych.
- Nie można wprowadzać nowych symboli operatorów.
- Niektóre operatory nie mogą być przeciążane: **is**, **and**, **or** i **not**.

Zacznijmy od operatorów arytmetycznych.

Przypuśćmy że mamy dwa obiekty, `a` i `b`, klasy **AClass** i chcemy nadać znaczenie wyrażeniu `a+b`, tak by jego wartością był nowy obiekt tej samej klasy odpowiadający, w jakimś sensie, sumie argumentów.[1] Aby osiągnąć ten cel, wystarczy zdefiniować w naszej klasie metodę **__add__**. Wtedy wyrażenie `a+b` będzie równoważne `a.__add__(b)`, lub, bardziej precyzyjnie, `type(a).__add__(a, b)`. Zauważmy, że metoda **__add__** (i tak samo dla innych operatorów binarnych) będzie zawsze wywołana na rzecz *lewego* argumentu.

Podobnie dla operatorów jednoargumentowych (unarnych). Teraz jest tylko jeden operand i specjalna metoda będzie wywołana na jego rzecz bez przesyłania żadnych dodatkowych argumentów. Na przykład operator negacji bitowej, ~, może być przeciążony metodą **__invert__**; wyrażenie `~a` będzie równoważne `a.__invert__()`, czyli `type(a).__invert__(a)`.

Zobaczmy przykład:

Listing 71 BBZ-Over/Resistor.py

```
#!/usr/bin/env python

class Resistor:
    """Describes resistor."""
    def __init__(self, r = 0):
        self.res = r                                    ❶

    @property
    def res(self):
        """Resistance of the resistor."""
        return self._res

    @res.setter
    def res(self, r):
        if r < 0: raise ValueError('Negative resistance')
        self._res = float(r)                            ❷

    def __add__(self, other):                           ❸
```

[1] W zasadzie mógłby to być dowolny obiekt dowolnej klasy, ale użytkownik spodziewać się takiego zachowania przez analogię do innych typów, dla których dodawanie jest określone; lepiej zatem stosować się do *zasady najmniejszego zaskoczenia* (*principle of least astonishment.*)

```
19          return Resistor(self.res + other.res)
20
21      def __mul__(self, other):                          ❹
22          return Resistor(self.res*other.res/(self.res+other.res))
23
24      def __repr__(self):                                ❺
25          return f'Resistor({self.res:.2f})'
26
27
28  r1, r2 = Resistor(), Resistor(6)
29  r1.res = 3
30  print(r1+r2, r1*r2)                                    ❻
```

Klasa **Resistor** opisuje oporniki. Ma jedno prywatne pole, _res, i właściwość res odpowiadającą oporowi opornika.Zauważmy, że w konstruktorze nie definiujemy self._res — używamy właściwości res (❶), która wywoła odpowiedni setter ❷. W ten sposób przy tworzeniu obiektu zostanie przeprowadzona ewentualne logowanie lub walidacja danych (w naszym przypadku będzie to zapewnienie, że opór jest dodatni i typu **float**). W linii ❸ definiujemy dodawanie oporników w ten sposób, aby dawało nowy opornik o oporze zastępczym równym sumie oporów, co odpowiada połączeniu szeregowemu. „Mnożenie" oporników również daje nowy opornik, ale z oporem zastępczym odpowiadającym połączeniu równoległemu (❹). Dodatkowo definiujemy metodę **__repr__** (❺), aby mieć napisową reprezentację naszych oporników. Wszystkie te specjalne metody są użyte w linii ❻. Program drukuje

```
Resistor(9.00) Resistor(2.00)
```

Wymieńmy operatory tego typu:

Tablica 4: Przeciążanie operatorów numerycznych

Operacja	Operator	Metoda
Dodawanie	`a + b`	`a.__add__(b)`
Odejmowanie	`a - b`	`a.__sub__(b)`
Mnożenie	`a * b`	`a.__mul__(b)`
Dzielenie	`a / b`	`a.__truediv__(b)`
Dzielenie całk.	`a // b`	`a.__floordiv__(b)`
Reszta	`a % b`	`a.__mod__(b)`
Potęgowanie	`a ** b`	`a.__pow__(b)`
Przesunięcie w lewo	`a << b`	`a.__lshift__(b)`
Przesunięcie w prawo	`a >> b`	`a.__rshift__(b)`
Bitowe AND	`a & b`	`a.__and__(b)`
Bitowe OR	`a | b`	`a.__or__(b)`

Tablica 4: (ciąg dalszy)

Operacja	Operator	Metoda
Bitowe XOR	`a ^ b`	`a.__xor__(b)`
Bitowe NOT	`~a`	`a.__not__()`
Unarny PLUS	`+a`	`a.__pos__()`
Unarny MINUS	`-a`	`a.__neg__()`

Większość z tych operatorów ma również formę złożonego przypisania (jak `a += b`). Ta forma też może być przeciążona (do nazwy odpowiedniej metody trzeba dodać 'i'). Od tych operatorów oczekujemy, że będą modyfikować `self` i zwracać `self`:

Tablica 5: Przeciążanie złożonego przypisania

Przypisanie z...	Operator	Metoda
dodawaniem	`a += b`	`a.__iadd__(b)`
odejmowaniem	`a -= b`	`a.__isub__(b)`
mnożeniem	`a *= b`	`a.__imul__(b)`
dzieleniem	`a /= b`	`a.__itruediv__(b)`
dzieleniem całk.	`a //= b`	`a.__ifloordiv__(b)`
resztą	`a %= b`	`a.__imod__(b)`
potęgowaniem	`a **= b`	`a.__ipow__(b)`
przesunięciem w lewo	`a <<= b`	`a.__ilshift__(b)`
przesunięciem w prawo	`a >>= b`	`a.__irshift__(b)`
bitowym AND	`a &= b`	`a.__iand__(b)`
bitowym OR	`a |= b`	`a.__ior__(b)`
bitowym XOR	`a ^= b`	`a.__ixor__(b)`

Zauważmy, że mamy problem z operatorami, które „z natury" powinny być symetryczne. Zdefiniujmy klasę **Modulo7**, która reprezentuje liczbę na której operacje arytmetyczne przeprowadzane są modulo 7 (❶). Ilustrujemy tu tylko dodawanie, ale inne operacje mogą być zaimplementowane podobnie. Definiujemy zatem metodę **__add__** (❸) dodającą do obiektu klasy liczbę, w oczywisty sposób. Wyrażenie `n4 + 5` będzie zinterpretowane jako `n4.__add__(5)` i otrzymamy obiekt odpowiadający 2 (bo 9 $\mod 7$ jest 2).

Listing 72 BDB-Reflect/ReflectBad.py

```
1 #!/usr/bin/env python
2
3 class Modulo7:
4     modulus = 7                                    ❶
5     def __init__(self, num):
6         self.num = num % Modulo7.modulus
7
8     def __add__(self, val):                        ❷
```

```
9           return Modulo7(self.num + val)
10
11      def __repr__(self):
12          return f'Modulo7({self.num})'
13
14
15  n4 = Modulo7(4)
16  n9 = n4 + 5                                    ❸
17  print(n4, n9)
18  n8 = 4 + n4                                    ❹
19  print(n8)
```

Program jednak załamuje się, bo `__add__` jest wywoływana na rzecz lewego operandu, który jest obiektem naszej klasy w linii ❸, ale nie ❹, gdzie występuje po stronie prawej — po lewej mamy po prostu liczbę:[1]

```
Modulo7(4) Modulo7(2)
Traceback (most recent call last):
  File "/home/werner/python/ReflectBad.py", line 18, in <module>
    n8 = 4 + n4                                  ##+4
TypeError: unsupported operand type(s) for +: 'int' and 'Modulo7'
```

Symetrię jednak można uratować. Załóżmy, że hipotetyczny operator ma symbol `@` odpowiadający magicznej metodzie `__oper__` w klasie AClass. Niech `a` będzie obiektem tej klasy. Wtedy, jak wiemy, `a @ b` będzie równoważne `a.__oper(b)`. Ale co się stanie, jeśli `a` stoi po prawej stronie, `b @ a`? Wtedy, jeśli

- lewy operand nie wspiera danej operacji (lub implementacja zwraca NotImplemented),
- i jest innego typu niż `a`,
- i klasa AClass implementuje metodę `__roper__` (zauważ 'r'),

to nastąpi „odbicie" operacji (*reflected operation*): `b @ a` będzie zamienione na `a.__roper__(b)`.

W naszym przykładzie dodamy zatem metodę `__radd__` (❶), która „obsłuży" wywołanie z odwróconą kolejnością argumentów (❷)

Listing 73 BDB-Reflect/ReflectGood.py

```
1   #!/usr/bin/env python
2
3   class Modulo7:
4       modulus = 7
5       def __init__(self, num):
6           self.num = num % Modulo7.modulus
7
8       def __add__(self, val):
9           return Modulo7(self.num + val)
10
```

[1] W C++ moglibyśmy zdefiniować funkcję operator+ jako globalną, w Pythonie to musi być metoda klasy.

```
11      def __radd__(self, other):                    ❶
12          return Modulo7(self.num + other)
13
14      def __repr__(self):
15          return f'Modulo7({self.num})'
16
17
18  n4 = Modulo7(4)
19  n9 = n4 + 5
20  print(n4, n9)
21  n8 = 4 + n4                                       ❷
22  print(n8)
```

i teraz `4+n4` w linii ❷ zadziała zgodnie z przewidywaniami

```
Modulo7(4) Modulo7(2)
Modulo7(1)
```

i wydrukuje

```
Modulo7(4) Modulo7(2)
Modulo7(1)
```

Użyliśmy tu dodawania, ale oczywiście ten sam mechanizm działa i dla innych operatorów:

Tablica 6: Operatory odbite

Operacja	Operator	Metoda
Dodawanie	`b + a`	`a.__radd__(b)`
Odejmowanie	`b - a`	`a.__rsub__(b)`
Mnożenie	`b * a`	`a.__rmul__(b)`
Dzielenie	`b / a`	`a.__rtruediv__(b)`
Dzielenie całk.	`b // a`	`a.__rfloordiv__(b)`
Reszta	`b % a`	`a.__rmod__(b)`
Potęgowanie	`b ** a`	`a.__rpow__(b)`
Przesunięcie w lewo	`b << a`	`a.__rlshift__(b)`
Przesunięcie w prawo	`b >> a`	`a.__rrshift__(b)`
Bitowe AND	`b & a`	`a.__rand__(b)`
Bitowe OR	`b | a`	`a.__ror__(b)`
Bitowe XOR	`b ^ a`	`a.__rxor__(b)`

Szczególnie ważny jest „pełen” zestaw sześciu operatorów porównania: `==`, `!=`, `<`, `<=`, `>`, `>=`. One też odpowiadają metodom magicznym, na przykład `a < b` odpowiada `a.__lt__(b)`.

Tablica 7: Operatory porównania

Porównanie	Operator	Metoda
mniejszy niż	`a < b`	`a.__lt__(b)`
mniejszy lub równy	`a <= b`	`a.__le__(b)`
większy niż	`a > b`	`a.__gt__(b)`
większy lub równy	`a >= b`	`a.__ge__(b)`
równy	`a == b`	`a.__eq__(b)`
różny od	`a != b`	`a.__ne__(b)`

Metody te możemy zdefiniować, gdy chcemy aby obiekty nasze klasy były porównywalne. Na przykład w programie poniżej definiujemy prostą klasę **AClass**, która ma jedno niestatyczne pole typu **str**. Przypuśćmy, że chcemy aby obiekty tej klasy były porównywane według następujących reguł:

- Krótsze napisy są „mniejsze” od dłuższych;
- Dla równych długości „mniejszy” jest napis wcześniejszy leksykograficznie, ale bez uwzględniania wielkości liter.

Możemy zatem zdefiniować wszystkie sześć operatory porównania (zauważ, że implementacja wielu z nich wykorzystuje te już zaimplementowane):

Listing 74 BDE-Cmp/Cmp.py

```
#!/usr/bin/env python

class AClass:
    def __init__(self, string):
        self.string = string

    def __lt__(self, other):
        if len(self.string) != len(other.string):
            return len(self.string) < len(other.string)
        else:
            return self.string.lower() < other.string.lower()

    def __gt__(self, other):
        return other < self

    def __ge__(self, other):
        return not (self < other)

    def __le__(self, other):
        return not (other < self)

    def __eq__(self, other):
        return self.string.lower() == other.string.lower()

    def __ne__(self, other):
        return not (self == other)
```

```
27
28      def __repr__(self):
29          return f"{self.string}"
30
31
32  def info(a, b):
33      print(f'a<b:{a<b!s:<5}, a<=b:{a<=b!s:5}, a==b:{a==b!s:5} '
34            f'a>=b:{a>=b!s:5}, a>b:{a>b!s:5}, a!=b:{a!=b!s:5}')
35
36  a, b, c, d, e, f = (AClass('Abc'),  AClass('aBC'),
37                      AClass('Abcd'), AClass('z'),
38                      AClass('XY'),   AClass('bcda'))
39  info(a, b); info(c, d); info(b, e)
40  lst = [a, b, c, d, e, f]
41  print(lst)
42  lst.sort()
43  print(lst)
```

Program drukuje

```
a<b:False, a<=b:True , a==b:True  a>=b:True , a>b:False, a!=b:False
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
[Abc, aBC, Abcd, z, XY, bcda]
[z, XY, Abc, aBC, Abcd, bcda]
```

czyli to, czego byśmy się spodziewali. Jest jednak prostszy sposób. W module ***functools*** zdefiniowany jest dekorator *klas* **total_ordering**. Zastosowany do klasy zdefiniuje on metody dla czterech z sześciu operatorów porównania; my musimy dostarczyć implementacje tylko dwóch z nich: **__eq__** i *jednej* z metod **__lt__**, **__le__**, **__gt__** i **__ge__**.

W ten sposób nasza klasa **AClass** może być przepisana w znacznie prostszej formie

Listing 75 BDE-Cmp/CmpDec.py

```
1   #!/usr/bin/env python
2
3   import functools
4
5   @functools.total_ordering
6   class AClass:
7       def __init__(self, string):
8           self.string = string
9
10      def __eq__(self, other):
11          return self.string.lower() == other.string.lower()
12
13      def __lt__(self, other):
14          if len(self.string) != len(other.string):
15              return len(self.string) < len(other.string)
```

```
16          else:
17              return self.string.lower() < other.string.lower()
18
19      def __repr__(self):
20          return f"{self.string}"
21
22  def info(a, b):
23      print(f'a<b:{a<b!s:<5}, a<=b:{a<=b!s:5}, a==b:{a==b!s:5} '
24            f'a>=b:{a>=b!s:5}, a>b:{a>b!s:5}, a!=b:{a!=b!s:5}')
25
26
27  a, b, c, d, e, f = (AClass('Abc'),  AClass('aBC'),
28                      AClass('Abcd'), AClass('z'),
29                      AClass('XY'),   AClass('bcda'))
30  info(a, b); info(c, d); info(b, e)
31  lst = [a, b, c, d, e, f]
32  print(lst)
33  lst.sort()
34  print(lst)
```

dając te same wyniki

```
a<b:False, a<=b:True , a==b:True  a>=b:True , a>b:False, a!=b:False
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
a<b:False, a<=b:False, a==b:False a>=b:True , a>b:True , a!=b:True
[Abc, aBC, Abcd, z, XY, bcda]
[z, XY, Abc, aBC, Abcd, bcda]
```

Zauważmy tu jednak, że metody wytworzone przez dekorator **total_ordering** mogą nie być optymalne. Jeśli zależy nam na wydajności, lepszym zwykle rozwiązaniem jest samodzielna staranna implementacja wszystkich sześciu metod „ręcznie”.

11.8.2 Inne metody magiczne

Metody magiczne są też używane w tworzonych przez nas klasach aby ich obiekty „odpowiadały” właściwie jako argumenty niektórych funkcji wbudowanych, i ogólnie aby zachowywały się sensownie w pewnych kontekstach. Na przykład metoda **__str__** będzie użyta przez funkcję (konstruktor) **str** i powinna dostarczyć tekstową reprezentację obiektu, a **__len__** będzie wywołana przez wbudowaną funkcję **len**. Niektóre z tych metod już spotkaliśmy, ale są inne.

Wymieńmy niektóre z nich.

- `__new__` and `__init__` — tę funkcję już znamy; zob. 11.2, str. 218.

- `__del__(self)` — będzie wywołana, gdy obiekt właśnie ma być usunięty z pamięci. Wbudowana funkcja **del** obniża tylko licznik referencji — **__del__** jest wywoływane tylko jeśli licznik staje się 0. W większości przypadków nie ma potrzeby definiowania tej metody.

- `__str__(self)`, `__repr__(self)` and `__bytes__(self)` — o dwóch pierwszych już mówiliśmy, zob. 11.7, str. 234. Trzecia jest wykorzystywana przez wbudowaną funkcję **bytes** i powinna zwracać reprezentację obiektu w formie obiektu typy **bytes**.

- `__format__(self, spec)` — jest wywoływana przez wbudowaną funkcję **format** i kilku innych kontekstach; zwykle jej nie definiujemy.

- `__hash__(self)` — jest wywoływana przez wbudowaną funkcje **hash**. Powinna zwracać liczbę typu **int**. Jeśli obiekty naszej klasy mają być elementami zbiorów lub kluczami w słownikach, powinniśmy zdefiniować *obie* metody, **__eq__** i **__hash__**, i to konsystentnie, czyli zapewnić, że jeśli dla dwóch obiektów **__eq__** (czyli ==) zwraca **True**, to ich hasze muszą być takie same. W przykładzie poniżej funkcje **__eq__** (❶) i **__hash__** (❷) obie zależą od pól fname i lname, zapewniając, że dla obiektów uważanych za równe hasz będzie obliczony z takich samych wartości. Implementując **__hash__** możemy użyć wbudowanej funkcji **hash** dla obiektów dla których jest ona już określona, tak jak w linii ❸, gdzie **hash** oblicza hasz krotki złożonej z fname i lname. Pamiętać też trzeba, że implementacja funkcji **hash** jest taka, że jej rezultat może być inny (a zatem inna może być kolejność elementów w czasie iteracji po zbiorze) przy każdym wykonaniu programu, nawet na tym samym komputerze dla dwóch kolejnych wykonań programu.

Listing 76 BDH-Hash/Hash.py

```
#!/usr/bin/env python

class A:
    def __init__(self, fname, lname):
        self.fname = fname
        self.lname = lname

    def __eq__(self, other):                              ❶
        return (self.fname,  self.lname ==
                other.fname, other.lname)

    def __hash__(self):                                   ❷
        return hash( (self.fname, self.lname) )           ❸

    def __repr__(self):
        return f'A("{self.fname}","{self.lname}")'


p1, p2, p3 = (A('Don', 'Ash'), A('Sue', 'Doe'),           ❹
                               A('Ada', 'Moo'))

d  = {A('Don', 'Ash'): 2000, A('Sue', 'Doe'): 1997,       ❺
                             A('Ada', 'Moo'): 2005}

print(f'{p1}->{d[p1]} ', f'{p2}->{d[p2]} ', f'{p3}->{d[p3]}')
```

Program drukuje:

```
A("Don","Ash")->2000  A("Sue","Doe")->1997  A("Ada","Moo")->2005
```

Zauważmy, że referencje p1, p2 i p3 odnoszą się do obiektów utworzonych w linii ❹, ale obiekty utworzone w linii ❺ jako klucze w słowniku d są innymi obiektami, o różnych

adresach, choć odnoszą się do tych samych osób, co p1, p2 i p3, bo mają te same imiona i nazwiska. To działa, bo nasza implementacja __hash__ i __eq__ jest oparta na imieniu i nazwisku, a nie adresie, jak to jest w domyślnej implementacji w klasie object. Spróbuj zakomentować jedną lub obie te funkcje i zobaczysz, że program przestanie działać.

- `__bool__(self)`, `__int__(self)`, `__complex__(self)` and `__float__(self)` — są definiowane, gdy konwersja wymagana jest konwersja obiektów klasy do tych typów, w szczególności gdy dla tych obiektów wywoływane są funkcje **bool**, **int**, **complex**, **float**. Spójrzmy na przykład:

Listing 77 BDN-Bool/Bool.py

```
#!/usr/bin/env python

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

    def __repr__(self):
        return f'Person(name={self.name},age={self.age})'

    def __bool__(self):           # conversion to bool
        return self.age >= 18


lst = [Person("A", 13), Person("B", 19),
       Person("C", 27), Person("D", 10)]
adults = [p for p in lst if p]    ❶
print(adults) # [Person(name=B,age=19), Person(name=C,age=27)]
```

W wyrażeniu w linii ❶ `[p for p in lst if p]` ostatnie p jest pod **if**em, więc musi być przekonwertowane do **bool**. Normalnie wszystko co nie jest zerem, nie jest **None** i nie jest pustą kolekcją jest rozumiane jako **True**. Jednakże w klasie **Person** nadpisaliśmy metodę __bool__, a zatem będzie ona użyta dając **True** tylko dla osób dorosłych (o wieku nie mniejszym od 18)

```
[Person(name=B,age=19), Person(name=C,age=27)]
```

- `__call__(self)`, — obiekty klas implementujących tę metodę stają się *obiektami wywoływalnymi* (*callabe object*), czyli zachowują się jak funkcje (w C++ jest podobnie dla klas implementujących metodę **operator()**). „Wywołanie" takiego obiektu, `obj(args)`, jest równoważne wywołaniu `type(obj).__call__(obj, args)` czyli `obj.__call__(a`
Taki mechanizm może być użyteczny, gdy chcemy mieć funkcję, ale która ma stan, który zachowuje się pomiędzy wywołaniami i można go modyfikować.

For example, in the following program

Listing 78 BDR-CallAve/Ave.py

```
#!/usr/bin/env python

class Average:
    def __init__(self):
        self.sum = 0
        self.num = 0

    def __call__(self, val):
        self.sum += val
        self.num += 1
        return self.sum / self.num

av1 = Average()
av2 = Average()

for n in range(1, 11):
    print(f'av1: adding {n:2d} -> {av1(n):5.2f}', ' ',   ❶
          f'av2: adding {n**2:3d} -> {av2(n*n):5.2f}')   ❷
```

klasa **Average** ma dwa pola odpowiadające sumie liczb i liczbie tych liczb. Każde wywołanie metody **__call__** zwiększa pole **sum** o przesłaną jako argument liczbę, inkrementuje **num** o jeden i zwraca nową wartość średniej arytmetycznej wszystkich do tej pory przesłanych liczb. W liniach ❶ i ❷przesyłamy liczby z zakresu $[1, 10]$ i ich kwadraty jako argumenty do dwóch wywoływalnych obiektów **av1** i **av2** naszej klasy. Program wypisuje

```
av1: adding  1 ->  1.00   av2: adding   1 ->  1.00
av1: adding  2 ->  1.50   av2: adding   4 ->  2.50
av1: adding  3 ->  2.00   av2: adding   9 ->  4.67
av1: adding  4 ->  2.50   av2: adding  16 ->  7.50
av1: adding  5 ->  3.00   av2: adding  25 -> 11.00
av1: adding  6 ->  3.50   av2: adding  36 -> 15.17
av1: adding  7 ->  4.00   av2: adding  49 -> 20.00
av1: adding  8 ->  4.50   av2: adding  64 -> 25.50
av1: adding  9 ->  5.00   av2: adding  81 -> 31.67
av1: adding 10 ->  5.50   av2: adding 100 -> 38.50
```

Rozpatrzmy teraz inny przykład: obliczamy tu liczby Fibonacciego za pomocą rekurencyjnej funkcji **fiboNoCache** oraz używając obiektu wywoływalnego klasy **FiboCached**. W tym drugim przypadku zachowujemy wszystkie do tej pory policzone wartości liczb Fibonacciego w słowniku, aby nie obliczać ich wciąż od nowa.

Listing 79 BDU-CallCache/Cache.py

```
#!/usr/bin/env python

class Average:
    def __init__(self):
        self.sum = 0
```

```
6           self.num = 0
7
8       def __call__(self, val):
9           self.sum += val
10          self.num += 1
11          return self.sum / self.num
12
13  av1 = Average()
14  av2 = Average()
15
16  for n in range(1, 11):
17      print(f'av1: adding {n:2d} -> {av1(n):5.2f}', ' ',  ❶
18            f'av2: adding {n**2:3d} -> {av2(n*n):5.2f}')  ❷
```

Jak widzimy z wydruku

```
102334155 1.517100099590607e-05 sec
102334155 12.199706612002046 sec
ratio: 804146
```

czas wykonania różni się o sześć rzędów wielkości!

Istnieje jeszcze więcej magicznych metod, które czasami są użyteczne — powiemy o nich, gdy będą potrzebne.

Listingi

Lista listingów

Index