Python – podstawy programowania

Marcin Pluciński

mplucinski@zut.edu.pl

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny

Opracowano w ramach projektu: „ZUT 2.0 – Nowoczesny Zintegrowany Uniwersytet”
nr POWER 03.05.00-00-Z205/17

Python – charakterystyka

- Łatwy do nauczenia.
- Treściwy kod.
- Niezależny od platformy sprzętowej.
- Język ogólnego przeznaczenia (np. obliczenia, obsługa baz danych, aplikacje internetowe, GUI, itd.).
- Język interpretowany (choć napisany program można łatwo przekształcić do postaci samodzielnej aplikacji).
- Dostarczany z pełną biblioteką standardową.
- Dostępne tysiące darmowych bibliotek opracowanych przez trzecie.
- Może być wykorzystany do programowania proceduralnego, zorientowanego obiektowo i w mniejszym stopniu do programowania funkcjonalnego.

Python – charakterystyka

Python 3 – Python 2 ?

- Aktualne wersje Python 3.12.6 (3.11.10, 3.10.15) i Python 2.7.18 dostępne na stronie: `https://www.python.org/`
- Python v.3 – ewolucyjne zmiany, nowe funkcje, poprawione błędy.
- Brak pełnej zgodności pomiędzy wersjami (np. inne działanie funkcji `print`, czy operatora dzielenia).

Python – środowisko

Po instalacji mamy do dyspozycji interpreter Pythona, umożliwiający uruchamianie programów, np.:

```
C:\Marcin\Dokumenty\Kursy\Python\Examples>python znaki.py
```

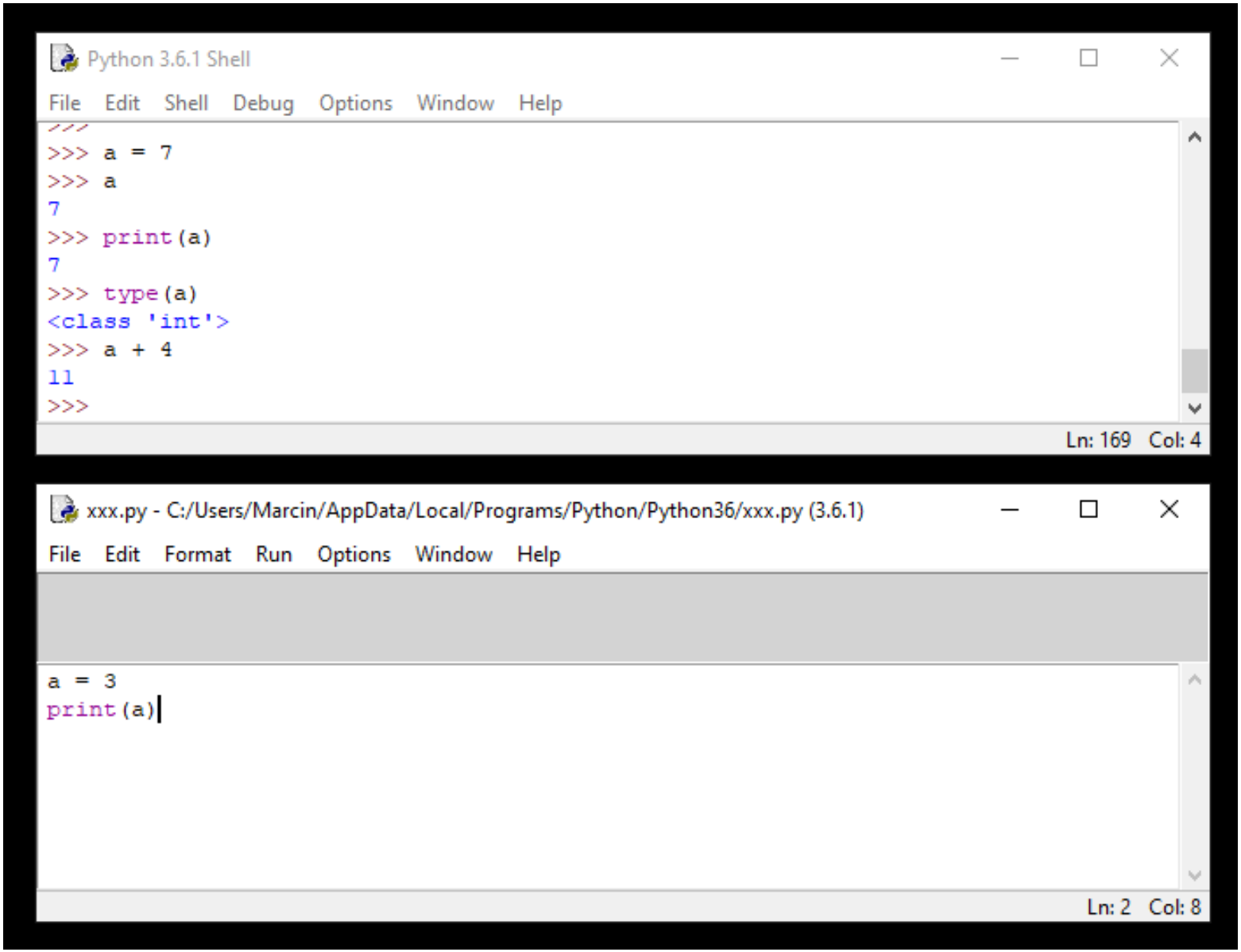
Python – środowisko

Po instalacji mamy do dyspozycji interpreter Pythona, umożliwiający uruchamianie programów, np.:

```
C:\Marcin\Dokumenty\Kursy\Python\Examples>python znaki.py
```

Mamy też do dyspozycji środowisko IDLE (Interactive Developement Environment), oferujące prosty edytor tekstu i tzw. powłokę interaktywną (Shell). Edytor umożliwia uruchamianie i debugowanie programów. Powłoka umożliwia wykonywanie dowolnych poleceń Pythona. Może być wykorzystywana przykładowo jako bardzo zaawansowany kalkulator.

Python – środowisko



Python – środowisko

Powłoka udostępnia także pomoc do języka.

```
>>>
>>>
>>> help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at http://docs.python.org/3.6/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".

help>
```

Python – środowisko

Popularne narzędzia:

- Eclipse + PyDev
- JetBrains PyCharm – darmowy w wersji Community Edition
- Visual Studio Code
- Spyder

Typy danych

Python udostępnia kilka wbudowanych typów danych.
Najważniejsze to:

- `int`
- `str`
- `float`

Dane tego typu są niezmienne!

Typy danych

Python udostępnia kilka wbudowanych typów danych.
Najważniejsze to:

- `int`
- `str`
- `float`

Dane tego typu są niezmienne!

Typ `int` reprezentuje liczby całkowite (dodatnie i ujemne). Wielkość liczb całkowitych w Pythonie jest ograniczona jedynie przez ilość pamięci zainstalowanej w komputerze, a nie przez odgórnie ustaloną liczbę bajtów.

```
>>> 132
132
>>> -567
-567
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376
```

Typy danych

Typ `float` reprezentuje wartości rzeczywiste (zmiennoprzecinkowe o podwójnej precyzji). Zakres zależy tu od kompilatora C użytego do kompilacji Pythona. Liczby tego typu mają ograniczoną precyzję (problemy z wiarygodnym porównywaniem!).

```
>>> 0.5, 0.1, -27.3, 9.2e-4, -3.6e3
(0.5, 0.1, -27.3, 0.00092, -3600.0)
```

Typy danych

Typ `float` reprezentuje wartości rzeczywiste (zmiennoprzecinkowe o podwójnej precyzji). Zakres zależy tu od kompilatora C użytego do kompilacji Pythona. Liczby tego typu mają ograniczoną precyzję (problemy z wiarygodnym porównywaniem!).

```
>>> 0.5, 0.1, -27.3, 9.2e-4, -3.6e3
(0.5, 0.1, -27.3, 0.00092, -3600.0)
```

Typ `str` reprezentuje ciągi tekstowe (sekwencje znaków Unicode).

```
>>> 'Przykładowy tekst'
'Przykładowy tekst'
>>> "Źdźbło trawy"
'Źdźbło trawy'
>>> ''
''
```

Typy danych

W celu uzyskania dostępu do elementu sekwencji (np. w tekście), w Pythonie używa się nawiasów kwadratowych []. Przykładowo:

```
>>> 'Przykładowy tekst'[5]
'ł'
>>> 'Przykładowy tekst'[0]
'P'
>>> 'Przykładowy tekst'[-1]
't'
```

Typy danych

W celu uzyskania dostępu do elementu sekwencji (np. w tekście), w Pythonie używa się nawiasów kwadratowych []. Przykładowo:

```
>>> 'Przykładowy tekst'[5]
'ł'
>>> 'Przykładowy tekst'[0]
'P'
>>> 'Przykładowy tekst'[-1]
't'
```

Omawiane typy danych są niezmienne! Czyli próba zmiany jakiegoś znaku spowoduje błąd:

```
>>> 'Przykładowy tekst'[5] = 'l'

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    'Przykładowy tekst'[5] = 'l'
TypeError: 'str' object does not support item assignment
```

Typy danych

Aby przekonwertować jeden typ danych na inny można użyć składni:
`typ_danych(element)`

```
>>> int(4.79)
4
>>> str(2.71)
'2.71'
>>> int('132')
132
```

```
>>> float(20)
20.0
>>> float('2.54')
2.54
>>> float('  2.765   ')
2.765
```

```
>>> int('a')
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    int('a')
ValueError: invalid literal for int() with base 10: 'a'

>>> int('2.54')
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    int('2.54')
ValueError: invalid literal for int() with base 10: '2.54'
```

Zmienne – odniesienia do obiektów

Python stosuje dynamiczną kontrolę typu zmiennej – typ jest ustalany w momencie przypisania wartości do zmiennej. Nie ma potrzeby deklaracji i określania typu.
Python nie posiada zmiennych jako takich – stosuje odniesienia do obiektów. W przypadku danych niezmiennych nie jest to istotne. W przypadku obiektów zmiennych może to mieć znaczenie.

```
a = 'jeden'
b = 'dwa'
c = a
```

Wykonując polecenia, Python tworzy obiekt typu `str` z tekstem 'jeden' i dalej tworzy odniesienie do obiektu, nazwane `a`. Trzecie polecenie tworzy nowe odniesienie `c`, wskazujące na ten sam obiekt co `a`.

Zmienne – odniesienia do obiektów

```
a = 'jeden'
b = 'dwa'
c = a
b = a
```

Po wykonaniu poleceń wszystkie trzy zmienne będą odnosiły się do obiektu 'jeden'. Ponieważ do obiektu 'dwa' nie ma więcej odniesień, Python może go usunąć (użyty zostanie mechanizm *garbage collection*).

Zmienne – nazwy

Nazwy zmiennych:

- nie mogą być takie same jak słowa kluczowe języka Python,
- muszą zaczynać się od litery lub znaku podkreślenia,
- składają się z liter, cyfr, znaku podkreślenia (dopuszczalne są dowolne znaki Unicode – bez znaków odstępu),
- nie mają ograniczenia długości,
- są wrażliwe na wielkość liter.

```
>>> a = 3
>>> A = 7
>>> print(a,A)
3 7

>>> a_b = 7
>>> żółć = 8
>>> żółć
8
>>> x = y = z = 'tekst'
>>> x,y,z
('tekst', 'tekst', 'tekst')
```

```
>>> while = 6
SyntaxError: invalid syntax
>>> a b = 3
SyntaxError: invalid syntax

>>> print = 5
```

Zmienne – nazwy

Polecenie `del` powoduje odłączenie odniesienia do obiektu (zmiennej) od danych i usunięcie zmiennej. Polecenie nie usuwa danych z pamięci. Tym zajmuje się mechanizm *garbage collection*.

```
>>> a = 7
>>> print(a)
7

>>> print = 7

>>> print(a)
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    print(a)
TypeError: 'int' object is not callable

>>> del print

>>> print(a)
7
```

Zmienne – typ

Typ zmiennych można dynamicznie zmieniać w trakcie wykonywania kodu. W przykładzie poniżej, każde kolejne przypisanie wiąże zmienną (odniesienie do obiektu) z kolejnymi obiektami typu `str`, `float` i na koniec `int`.

```
>>> x = 'Tekst'
>>> print(x, type(x))
Tekst <class 'str'>

>>> x = 5.7
>>> print(x, type(x))
5.7 <class 'float'>

>>> x = -20
>>> print(x, type(x))
-20 <class 'int'>
```

Podstawowe kolekcje

Podstawowe typy kolekcji to: `list` – lista i `tuple` – krotka. Obie służą do przechowywania dowolnej liczby elementów dowolnego typu w formie uporządkowanej sekwencji.

Krotki (podobnie jak podstawowe typy danych) pozostają niezmienne. Listy są zmienne: można dodawać i usuwać ich elementy, a także zmieniać ich wartość.

Krotki definiujemy w nawiasach ().

```
>>> a = ('abc', 'def', 'ijk')
>>> a
('abc', 'def', 'ijk')
>>> b = (1,3,5,7)
>>> b
(1, 3, 5, 7)
>>> c = ('abc', 1, 3.987, 'x')
>>> c
('abc', 1, 3.987, 'x')
```

```
>>> d = ()
>>> d
()
>>> d = (5)
>>> d
5
>>> d = (5,)
>>> d
(5,)
```

Podstawowe kolekcje

Listy definiujemy w nawiasach [].

```
>>> a = ['abc', 'def', 'ijk']
>>> a
['abc', 'def', 'ijk']
>>> b = [1,3,5,7]
>>> b
[1, 3, 5, 7]
>>> c = ['abc', 1, 3.987, 'x']
>>> c
['abc', 1, 3.987, 'x']
```

```
>>> d = []
>>> d
[]
>>> d = [5]
>>> d
[5]
>>> d = [5,]
>>> d
[5]
```

Listy i krotki

Do określania rozmiaru listy, krotki (i innych typów, dla których ma to sens) służy funkcja `len`.

```
>>> a = ('aaa','bbb','ccc')
>>> len(a)
3
>>> b = ['123',1,2,3]
>>> len(b)
4
>>> c = []
>>> len(c)
0
>>> d = 'To jest tekst'
>>> len(d)
13
```

Listy i krotki

Wewnętrznie listy i krotki nie przechowują elementów danych, a jedynie odniesienia do obiektów – w trakcie tworzenia listy, są one do niej kopiowane.

Podobnie jak inne typy danych w Pythonie (np. `int`, `str`, `float`), listy i krotki są obiektami – egzemplarzami określonego typu danych (nazywanego też klasą). Obiekty mogą mieć metody – funkcje wywoływane dla określonych obiektów.

Przykładowo typ `list` ma metodę `append()`, która umożliwia dodawanie elementu na koniec listy.

```
>>> a = ['123',1,-22,'xyz',0.2]
>>> print(a)
['123', 1, -22, 'xyz', 0.2]
>>> a
['123', 1, -22, 'xyz', 0.2]
>>> a.append('nowy')
>>> a
['123', 1, -22, 'xyz', 0.2, 'nowy']
```

Listy i krotki

Obiekt a „wie", że jest typu `list` – w Pythonie wszystkie obiekty „znają" swój typ. W praktycznej implementacji metody append, pierwszym argumentem jest zawsze sam obiekt a – przekazanie tego obiektu jest przeprowadzane automatycznie (jako część syntaktycznej obsługi metody).

Każdą metodę można także wykorzystać inaczej – przekazując do niej obiekt w sposób jawny.

```
>>> a = ['123',1,-22,'xyz',0.2]
>>> a
['123', 1, -22, 'xyz', 0.2]

>>> a.append('nowy')
>>> a
['123', 1, -22, 'xyz', 0.2, 'nowy']

>>> list.append(a,'najnowszy')
>>> a
['123', 1, -22, 'xyz', 0.2, 'nowy', 'najnowszy']
```

Listy i krotki

Operator kropki jest używany w celu uzyskania dostępu do atrybutów i metod obiektu. Zarówno listy, jak i krotki posiadają takich metod wiele.

Podobnie jak dla typu tekstowego, za pomocą nawiasów kwadratowych możemy odwoływać się do dowolnych elementów listy i krotki.

```
>>> a = [1,3,5,7]
>>> a[0]
1
>>> a[1:3]
[3, 5]
>>> a[1] = 'trzy'
>>> a
[1, 'trzy', 5, 7]
```

```
>>> b = (1,3,5,7)
>>> b[0]
1

>>> b[1] = 'trzy'
Traceback (most recent call last):
  File "<pyshell#89>", line 1, in <module>
    b[1] = 'trzy'
TypeError: 'tuple' object does not support item assignment

>>> b.append(9)
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    b.append(9)
AttributeError: 'tuple' object has no attribute 'append'
```

Listy i krotki

```
>>> a = (1,2,3)
>>> b = [4,5,6]
>>> print(a,b)
(1, 2, 3) [4, 5, 6]
>>> b.append(9)
>>> b
[4, 5, 6, 9]
>>> b.append(a)
>>> b
[4, 5, 6, 9, (1, 2, 3)]

>>> c = [1,1,1]
>>> b.append(c)
>>> b
[4, 5, 6, 9, (1, 2, 3), [1, 1, 1]]

>>> c[1]=9
>>> c
[1, 9, 1]
>>> b
[4, 5, 6, 9, (1, 2, 3), [1, 9, 1]]
```

Operatory logiczne – operator tożsamości

Operator tożsamości `is` sprawdza, czy dwa odniesienia od obiektu wskazują na ten sam obiekt. Operator **nie porównuje wartości**, porównuje jedynie adresy w pamięci dla wskazanych obiektów.

```
>>> a = [1,'dwa',3]
>>> b = [1,'dwa',3]
>>> a is b
False
>>> c = a
>>> a is c
True
```

Operatory logiczne – operator tożsamości

Często spotykanym przykładem użycia operatora `is` jest porównanie obiektu z wbudowanym w język obiektem `None`, używanym do wskazania na obiekt nieistniejący.

W celu odwrócenia testu tożsamości używamy operatora `is not`.

```
>>> a = [1,'dwa',3]
>>> a is None
False
>>> a is not None
True
>>> b = None
>>> b is None
True
```

Operatory logiczne – operator porównania

Python oferuje standardowy zestaw binarnych operatorów porównania. Operatory porównują **wartości** obiektów.

```
>>> a = 5
>>> b = 8
>>> a == b, a != b, a < b, a <= b, a > b, a >= b
(False, True, True, True, False, False)

>>> x = 'abc'
>>> y = 'def'
>>> z = 'abc'
>>> x is z
True
>>> x == y, x == z, x != y, x > y
(False, True, True, False)

>>> a = [1,'dwa',3]
>>> b = [2,'trzy',4]
>>> c = [1,'dwa',3]
>>> a == b, a == c, a != b, a < b
(False, True, True, True)
```

Operatory logiczne – operator porównania

Jedną ze szczególnie użytecznych cech operatorów porównania w Pythonie jest możliwość ich łączenia.

```
>>> a = 5
>>> 0 <= a < 9
True
>>> 0 > a >=9
False
```

Operatory logiczne – operator porównania

Jedną ze szczególnie użytecznych cech operatorów porównania w Pythonie jest możliwość ich łączenia.

```
>>> a = 5
>>> 0 <= a < 9
True
>>> 0 > a >=9
False
```

Przy porównywaniu stosowana jest kontrola typu.

```
>>> 1 > 'zero'
Traceback (most recent call last):
  File "<pyshell#158>", line 1, in <module>
    1 > 'zero'
TypeError: '>' not supported between instances of 'int' and 'str'
```

Op. logiczne – operator przynależności

W przypadku typów danych będących sekwencjami lub kolekcjami (listy, krotki, tekst) można sprawdzać przynależność elementu za pomocą operatora `in`, a brak przynależności za pomocą `not in`.

```
>>> a = [1,2,3,'cztery']
>>> 3 in a
True
>>> 'cztery' in a
True
>>> 'trzy' in a
False
>>> 'dwa' not in a
True

>>> zdanie = 'To jest przykładowe zdanie'
>>> 'T' in zdanie
True
>>> 'ą' not in zdanie
True
>>> 'jest' in zdanie
True
>>> 'zdanie ' in zdanie
False
```

Operatory logiczne

Język Python oferuje trzy operatory logiczne: `and`, `or`, `not`.

```
>>> a = 8
>>> b = 3
>>> c = 0

>>> a > b or b < c
True
>>> a > b and b < c
False
>>> not (a > b)
False

>>> not a
False
>>> not c
True
>>> not -0.01
False
>>> not 0.0
True
```

Kontrola pracy programu – polecenie if

Polecenia znajdujące się w pliku `*.py` są wykonywane po kolei od pierwszego wiersza. Zmienić to można wywołując funkcję (metodę), używając poleceń warunkowych lub tworząc pętle w programach. Przebieg wykonywania programu jest też zmieniany po zgłoszeniu wyjątku.

Składnia polecenia `if` jest następująca.

```
if wyrażenie_logiczne_1:
    blok_kodu_1
elif wyrażenie_logiczne_2:
    blok_kodu_2
    ....
elif wyrażenie_logiczne_N:
    blok_kodu_N
else:
    blok_else
```

Kontrola pracy programu – polecenie if

Wyrażenie logiczne – to dowolne wyrażenie, które w wyniku obliczenia da nam wartość logiczną: `True`, `False`.
W języku Python wyrażenie będzie fałszywe gdy:

- jawnie będzie równe `False`,
- jest obiektem `None`,
- jest pustą sekwencją bądź kolekcją (np. listą, krotką, tekstem),
- liczbowym typem danych równym 0.

W każdym innym przypadku wyrażenie będzie traktowane jako prawdziwe.

Blok kodu – sekwencja jednego lub większej liczby poleceń. Jeśli blok taki jest wymagany, a nie chcemy wykonywać żadnych działań, Python udostępnia nam polecenie `pass`, które nie wykonuje żadnego działania.

Kontrola pracy programu – polecenie if

Liczba klauzul `elif` może być dowolna (także 0), a klauzula `else` jest opcjonalna.

Charakterystyczne cechy – brak nawiasów oddzielających blok kodu i obecność dwukropka przed blokiem kodu.

Do wyróżnienia bloku kodu stosujemy wcięcia – standardowo 4 spacje na każdy poziom wcięcia. Python działa także z dowolną liczbą spacji zakładając, że użyte wcięcia zachowają spójność.

Kontrola pracy programu – polecenie if

```
x = 1
if x:
    print('x nie jest zerem')

############################################################

liczba = 17
if 0 <= liczba <= 10:
    print('Liczba z przedziału 0-10')
elif liczba > 10:
    print('Liczba większa od 10')
else:
    print('Liczba mniejsza od 0')

############################################################

a = 'm'
zdanie = 'To jest tekst'
if a in zdanie:
    print('Znak',a,'występuje w zdaniu:',zdanie)
else:
    print('Znak',a,'nie występuje w zdaniu:',zdanie)
```

Kontrola pracy programu – polecenie while

Polecenie `while` jest używane w celu wykonania bloku kodu wiele razy. Ilość powtórzeń zależy od stanu wyrażenia logicznego, znajdującego się w nagłówku pętli.

Składnia polecenia `while` jest następująca.

```
while wyrażenie_logiczne:
    blok_kodu
```

Kontrola pracy programu – polecenie while

Polecenie `while` jest używane w celu wykonania bloku kodu wiele razy. Ilość powtórzeń zależy od stanu wyrażenia logicznego, znajdującego się w nagłówku pętli.

Składnia polecenia `while` jest następująca.

```
while wyrażenie_logiczne:
    blok_kodu
```

W składni polecenia można jeszcze użyć słowa kluczowego `else`. Jego znaczenie omówione będzie dalej.

W pętli można używać poleceń:

- `break` – powoduje przerwanie działania pętli i przekazanie kontroli nad programem do pierwszego polecenia za nią,
- `continue` – powoduje powrót do nagłówka pętli.

Kontrola pracy programu – polecenie while

```
x = 10
while x > 0:
    print(x)
    x = x - 2
    if x == 0:
        break
```

```
10
8
6
4
2
```

```
##########################################################
```

```
a = 0
while 0 <= a <= 9:
    a = a + 1
    if a == 3 or a == 7 or a == 8:
        continue
    print(a)
```

```
1
2
4
5
6
9
10
```

Kontrola pracy programu – polecenie for

Polecenie `for` jest używane w celu wykonania bloku kodu określoną ilość razy. Blok jest wykonywany dla każdej wartości występującej w sekwencji z nagłówka pętli. Sekwencją jest przykładowo: lista, krotka i tekst.

Składnia polecenia `for` jest następująca.

```
for zmienna in sekwencja:
    blok_kodu
```

W składni polecenia można jeszcze użyć słowa kluczowego `else`. Jego znaczenie omówione będzie dalej.

W pętli można używać poleceń `break` i `continue`.

Kontrola pracy programu – polecenie for

```
for element in [1, 2.5, 'trzy', 2<4]:
    print(element, type(element))

########################################

zdanie = 'To jest zdanie'
for znak in zdanie:
    if znak in 'AEIOUYaeiouy':
        print(znak, 'jest samogłoską')
    elif znak == ' ':
        print('spacja')
    else:
        print(znak, 'jest spółgłoską')
```

```
1 <class 'int'>
2.5 <class 'float'>
trzy <class 'str'>
True <class 'bool'>


T jest spółgłoską
o jest samogłoską
spacja
j jest spółgłoską
e jest samogłoską
s jest spółgłoską
t jest spółgłoską
spacja
z jest spółgłoską
d jest spółgłoską
a jest samogłoską
n jest spółgłoską
i jest samogłoską
e jest samogłoską
```

Polecenie for + funkcja range()

Funkcja range generuje obiekt (sekwencję) przechowującą ciąg arytmetyczny wartości.

```
for x in range(5):
    print(x)

# [0, 1, 2, 3, 4]

for x in range(2,8):
    print(x)

# [2, 3, 4, 5, 6, 7]

for x in range(0,20,4):
    print(x)

#[0, 4, 8, 12, 16]
```

```
zdanie = 'To jest zdanie'

for i in range(len(zdanie)):
    print(zdanie[i])

for znak in zdanie:
    print(znak)
```

Podstawy obsługi wyjątków

Wiele funkcji i metod Pythona generuje w pewnych sytuacjach błędy i zdarzenia poprzez zgłaszanie wyjątku. Wyjątek jest obiektem.

Składnia obsługi wyjątków jest następująca.

```
try:
    blok_kodu
except wyjątek_1 as zmienna_1:
    blok_kodu_1
...
except wyjątek_N as zmienna_N:
    blok_kodu_N
```

Użycie zmiennych jest opcjonalne. Zmienne przydają się przy wyświetlaniu informacji o zaistniałym wyjątku. Pełna składnia obsługi wyjątków jest w rzeczywistości bardziej skomplikowana i będzie omówiona dalej.

Podstawy obsługi wyjątków

```
try:
    blok_kodu
except wyjątek_1 as zmienna_1:
    blok_kodu_1
...
except wyjątek_N as zmienna_N:
    blok_kodu_N
```

Jeśli wszystkie polecenia bloku `try` zostaną wykonane bez zgłoszenia wyjątku, bloki `except` będą pominięte. Jeśli wyjątek wystąpi, program natychmiast przeskoczy do bloku kodu powiązanego z pierwszym z kolei dopasowanym typem wyjątku. Pozostałe polecenia w bloku `try` zostaną pominięte. Jeśli zdefiniowano zmienną, wówczas będzie ona odniesieniem do obiektu wyjątku.

Jeśli wyjątek zostanie zgłoszony w bloku `except` albo nie uda się znaleźć dopasowania, zwykle dochodzi do przerwania wykonywania programu z nie obsłużonym wyjątkiem. Python wyświetla wtedy komunikat dotyczący ostatnich poleceń i komunikat tekstowy wygenerowany przez wyjątek.

Podstawy obsługi wyjątków

```
a = 'trzy'
b = int(a)

------------------------------------------------------------

>>>
Traceback (most recent call last):
  File "D:\Marcin\Kursy\Python\Examples\proby.py", line 2, in <module>
    b = int(a)
ValueError: invalid literal for int() with base 10: 'trzy'
>>>
```

Podstawy obsługi wyjątków

```
try:
    a = 'trzy'
    b = int(a)
except ValueError:
    print('Nieprawidłowa wartość!')
```

```
-------------------------------------------------------
>>>
Nieprawidłowa wartość!
>>>
```

```
try:
    a = 'trzy'
    b = int(a)
except ValueError as zm_err:
    print('Nieprawidłowa wartość!')
    print(zm_err)
```

```
-------------------------------------------------------
>>>
Nieprawidłowa wartość!
invalid literal for int() with base 10: 'trzy'
>>>
```

Operatory arytmetyczne

Mamy do dyspozycji 4 podstawowe operatory arytmetyczne: $+ - * /$

```
x = 4
y = 2
z = x / y
print(x,type(x))
print(y,type(y))
print(z,type(z))
```

```
-------------------------------------------------
```

```
4 <class 'int'>
2 <class 'int'>
2.0 <class 'float'>
```

Operator dzielenia zawsze zwraca w wyniku liczbę zmiennoprzecinkową!

Operatory arytmetyczne

- `x // y` – dzieli liczbę x przez y, odrzucając część ułamkową. Wynikiem jest zawsze liczba typu `int`.
- `x % y` – oblicza resztę z dzielenia x przez y.
- `-x` – negacja x,
- `x ** y` – oblicza x do potęgi y. Jest też funkcja `pow(x,y)`.
- `abs(x)` – oblicza wartość bezwzględną z x.

```
>>> c = 2 ** 3
>>> print(c,type(c))
8 <class 'int'>

>>> c = 2 ** -3      # 0.125 <class 'float'>
>>> c = 2.7 ** 0.5   # 1.6431676725154984 <class 'float'>
>>> c = 20 // 7      # 2 <class 'int'>
>>> c = 20 % 7       # 6 <class 'int'>
```

Operatory arytmetyczne

Wszystkie operatory mają też swoje odpowiedniki w formie rozszerzonych operatorów przypisania:

```
+=, -=, *=, /=, //=, %=, **=
```

```
>>> a = 2
>>> a **= 5       # 32
>>> a //= 10      # 3
>>> a *= 20       # 60
>>> a %= 8        # 4
```

Ponieważ liczbowe typy danych są niezmienne, w wyniku użycia takich operatorów tworzony jest nowy obiekt przechowujący wynik i to tego nowego obiektu będzie dalej odnosić się zmienna.

Operatory arytmetyczne

W Pythonie można przeciążać operatory – będą odpowiednio działać także dla klas innych niż podstawowe typy liczbowe.
Przykładowo dla tekstów i list można używać operatorów:
`+, +=, *, *=`.

```
>>> a = 'nowy'
>>> b = 'tekst'
>>> c = a + ' ' + b
>>> c
'nowy tekst'
>>> a += ' wiersz'
>>> a
'nowy wiersz'
>>> d = 'tekst ' * 3
>>> d
'tekst tekst tekst '
>>> a *= 4
>>> a
'nowy wiersznowy wiersznowy wiersznowy wiersz'
>>> b - a
Traceback (most recent call last):
    b - a
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

Operatory arytmetyczne

```
>>> lista = ['aaa','bbb','ccc']
>>> lista1 = [1,2,3]
>>> lista2 = lista + lista1
>>> lista2
['aaa', 'bbb', 'ccc', 1, 2, 3]

>>> lista1 += 4
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    lista1 += 4
TypeError: 'int' object is not iterable

>>> lista1 += [4]
>>> lista1
[1, 2, 3, 4]

>>> lista3 = lista1 * 3
>>> lista3
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
```

Operatory arytmetyczne

```
>>> lista4 = lista + 'tekst'
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    lista4 = lista + 'tekst'
TypeError: can only concatenate list (not "str") to list

>>> lista += 'tekst'
>>> lista
['aaa', 'bbb', 'ccc', 't', 'e', 'k', 's', 't']

>>> lista += ['tekst']
>>> lista
['aaa', 'bbb', 'ccc', 't', 'e', 'k', 's', 't', 'tekst']
```

Operacje wejścia – wyjścia

Do wyświetlania wyników działania programu w konsoli najprościej użyć funkcji `print()`, która dokładniej omówiona będzie dalej.

Wprowadzanie danych z klawiatury można zrealizować z pomocą funkcji `input()`. Jej argumentem może być tekst wyświetlany w konsoli. Funkcja zatrzymuje działanie programu, czeka aż użytkownik wprowadzi dane i naciśnie Enter. Funkcja zwraca wprowadzony tekst (jeśli tylko naciśnięto Enter, zwrócony będzie pusty ciąg tekstowy).

Operacje wejścia – wyjścia

```
print('Wprowadzaj liczby całkowite + Enter')
print('Samo Enter kończy program')

suma = 0
while True:
    liczba = input('Podaj liczbę: ')
    if liczba:
        try:
            wartosc = int(liczba)
        except ValueError as err:
            print(err)
            continue
        suma += wartosc
    else:
        break

print('Suma liczb =', suma)
```

```
Wprowadzaj liczby całkowite + Enter
Samo Enter kończy program
Podaj liczbę: 1
Podaj liczbę: 2
Podaj liczbę: 3
Podaj liczbę: 4
Podaj liczbę:
Suma liczb = 10

Wprowadzaj liczby całkowite + Enter
Samo Enter kończy program
Podaj liczbę: 1
Podaj liczbę: 2
Podaj liczbę: 3
Podaj liczbę: '4'
invalid literal for int() with base 10: "'4'
Podaj liczbę: 4.5
invalid literal for int() with base 10: '4.5
Podaj liczbę: 4
Podaj liczbę:
Suma liczb = 10
```

Funkcje

Ogólna składnia funkcji:

```
def nazwa_funkcji(argumenty):
    blok_kodu
```

- Argumenty są opcjonalne.
- Jeśli jest ich kilka, rozdzielamy je przecinkami.
- Każda funkcja zwraca wartość.
- Domyślnie zwracana jest wartość `None`.
- Można zwrócić inną wartość poleceniem: `return wartość`.
- Zwracana wartość może być pojedynczym elementem lub krotką elementów.
- Wartość zwrotna może być zignorowana w miejscu wywołania.
- Funkcje są obiektami, a polecenie `def` tworzy odniesienie do obiektu funkcji.

Funkcje

```
def pole_trapezu(a,b,h):
    if a < 0 or b < 0 or h < 0:
        return None
    else:
        pole = 0.5*(a+b)*h
        return pole

pole = pole_trapezu(5.5, 7, 4)
print('Pole trapezu =', pole)

-------------------------------------------------

>>>
Pole trapezu = 25.0
```

Funkcje

Python dostarcza wiele gotowych funkcji wbudowanych i funkcji znajdujących się w zewnętrznych modułach (np. w bibliotece standardowej).

Moduł jest zwykłym plikiem tekstowym z rozszerzeniem `*.py`, w którym znajdują się definicje funkcji, klas i zmiennych. Moduł importujemy poleceniem `import nazwa_modułu` (bez rozszerzenia!) i od tego momentu uzyskujemy dostęp do dowolnej funkcji, klasy bądź zmiennej zdefiniowanej w module.

Składnia użycia funkcji z modułu:

```
nazwa_modułu.nazwa_funkcji(argumenty)
```

Polecenia `import` zaleca się umieszczać na początku pliku (najpierw moduły z biblioteki standardowej, potem moduły firm trzecich, na koniec własne).

Funkcje

```
import math

def pierwiastki(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        x1 = (-b + math.sqrt(delta))/(2*a)
        x2 = (-b - math.sqrt(delta))/(2*a)
        return x1, x2
    else:
        print('Funkcja nie liczy pierwiastków w postaci liczb zespolonych!')
        return None,None

pierwiastki(1,9,1)

(x_1,x_2) = pierwiastki(1,2,1)
print(x_1,x_2)

wynik = pierwiastki(1,2,1)
print(wynik)

wynik = pierwiastki(1,1,1)
print(wynik)
```

Funkcje

```
import math

def pierwiastki(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        x1 = (-b + math.sqrt(delta))/(2*a)
        x2 = (-b - math.sqrt(delta))/(2*a)
        return x1, x2
    else:
        print('Funkcja nie liczy pierwiastków w postaci liczb zespolonych!')
        return None,None

pierwiastki(1,9,1)

(x_1,x_2) = pierwiastki(1,2,1)
print(x_1,x_2)

wynik = pierwiastki(1,2,1)
print(wynik)

wynik = pierwiastki(1,1,1)
print(wynik)

------------------------------------------------------

-1.0 -1.0
(-1.0, -1.0)
Funkcja nie liczy pierwiastków w postaci liczb zespolonych!
(None, None)
```

Zmienne i proste typy danych

Zmienne – nazwy

Nazwy zmiennych:

- muszą zaczynać się od litery lub znaku podkreślenia,
- składają się z liter, cyfr, znaku podkreślenia (dopuszczalne są dowolne znaki Unicode – bez znaków odstępu),
- nie mają ograniczenia długości,
- są wrażliwe na wielkość liter.

Zmienne – nazwy

Nazwy zmiennych nie mogą być takie same jak słowa kluczowe języka Python.

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

Zmienne – nazwy

Należy przestrzegać konwencji, zgodnie z którą nazwy zmiennych nie powinny być takie same jak nazwy wbudowanych typów danych, funkcji czy wyjątków Pythona.

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
...
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning'
'UserWarning', 'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError',
'_', '__build_class__', '__debug__', '__doc__', '__import__', '__loader__',
'__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin',
'bool', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile',
'complex', 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
'isinstance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'map',
'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'pri
'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'sl
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip'
```

Zmienne – nazwy

Nazwy rozpoczynające się i kończące dwoma znakami podkreślenia (np. `__str__`) nie powinny być używane.

Nazwy tego typu służą do definiowania zmiennych i metod specjalnych przy tworzeniu obiektów. Można je ponownie zaimplementować (nadpisać), ale nie powinno się tworzyć własnych.

Nazwy, które zaczynają się (ale nie kończą) jednym bądź dwoma podkreśleniami są również w pewnych okolicznościach traktowane w sposób specjalny. Np w Shell-u, podkreślenie może oznaczać ostatnio wykonywane polecenie. W programach lokalizowanych na wiele języków, podkreślenie bywa używane jako funkcja tłumacząca.

```
>>> a = 3
>>> type(a)
<class 'int'>
>>> _
<class 'int'>
```

Liczby całkowite – typ int

Typ `int` reprezentuje liczby całkowite (dodatnie i ujemne). Wielkość liczb całkowitych w Pythonie jest ograniczona jedynie przez ilość pamięci zainstalowanej w komputerze, a nie przez odgórnie ustaloną liczbę bajtów. Liczby można także definiować stosując inne podstawy, np.:

```
>>> a = 132123                  # dziesiętna
132123
>>> b = 0b1100                  # binarna (lub 0B1100)
>>> b
12
>>> c = 0xFF00                  # szestnastkowa (0XFF00)
>>> c
65280
>>> d = 0o1238                  # ósemkowa (0O1238)
SyntaxError: invalid syntax
>>> d = 0o1237
>>> d
671
```

Liczby całkowite – typ int

Operatory arytmetyczne omówiono wcześniej.

Funkcje konwersji dla liczb całkowitych:

- `int(x)` – konwertuje obiekt `x` do typu całkowitego.
- `int(x,baza)` – konwertuje obiekt `x` do typu całkowitego. Baza musi być liczbą całkowitą z przedziału od 2 do 36.
- `bin(x)` – zwraca ciąg tekstowy reprezentujący binarną postać `x`.
- `hex(x)` – zwraca ciąg tekstowy reprezentujący szestnastkową postać `x`.
- `oct(x)` – zwraca ciąg tekstowy reprezentujący ósemkową postać `x`.

```
>>> int('1100',2)
12
>>> int('FFFF',16)
65535
>>> int('1234',5)
194
```

```
>>> bin(15)
'0b1111'
>>> oct(15)
'0o17'
>>> hex(15)
'0xf'
```

Liczby całkowite – typ int

Python udostępnia operatory bitowe działające na liczbach typu `int`: `&`, `|`, `^`, `<<`, `>>`, `~` oraz ich wersje z operatorem przyrównania: `&=`, `|=`, `^=`, `<<=`, `>>=`. Operatory bitowe działają na reprezentacjach binarnych.

- `x & y` – operacja AND na liczbach `x` i `y`.
- `x | y` – operacja OR.
- `x ^ y` – operacja XOR.
- `x << n` – przesunięcie bitów w `x` o `n` miejsc w lewo.
- `x >> n` – przesunięcie bitów w `x` o `n` miejsc w prawo.
- `~x` – odwrócenie bitów.

```
>>> x = 0b110011
>>> y = 0b101010
>>> z = x & y
>>> bin(z)
'0b100010'
>>> bin(x | y)
'0b111011'
```

```
>>> bin(x ^ y)
'0b11001'
>>> bin(x >> 2)
'0b1100'
>>> bin(x << 2)
'0b11001100'
```

Liczby całkowite – typ int

Dla klasy `int` dostępna jest metoda `int.bit_length()` określająca ilość bitów potrzebną do reprezentacji liczby całkowitej.

```
>>> a = 111
>>> a.bit_length()
7
>>> (111111).bit_length()
17
>>> (0xFF).bit_length()
8
>>> (0xFFaa).bit_length()
16
>>> (2**347).bit_length()
348
>>> 2**347
2866873269987589389513526119127608675995706236460351404671986049233653595110606
>>> 348 // 8
43
>>> divmod(348,8)
(43, 4)
>>>
```

Wartości logiczne – typ bool

W Pythonie istnieją 2 wbudowane obiekty logiczne: `True` i `False`.

Liczby zmiennoprzecinkowe

W Pythonie mamy 3 rodzaje wartości zmiennoprzecinkowych: wbudowane `float` i `complex` oraz typ `decimal.Decimal` z biblioteki standardowej.

Typ `float` reprezentuje wartości rzeczywiste (zmiennoprzecinkowe o podwójnej precyzji). Zakres zależy tu od kompilatora C użytego do kompilacji Pythona. Liczby tego typu mają ograniczoną precyzję (problemy z wiarygodnym porównywaniem!).

```
>>> 0.5, 0.1, -27.3, 9.2e-4, -3.6e3
(0.5, 0.1, -27.3, 0.00092, -3600.0)
```

Wszystkie operatory arytmetyczne omówione wcześniej, mogą być stosowane z liczbami typu `float`.

```
>>> import sys
>>> sys.float_info
(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
 min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307,
 dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, ro
```

Liczby zmiennoprzecinkowe – float

Bardziej złożone funkcje i pewne stałe udostępnia moduł `math`.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.inf
inf
>>> math.nan
nan
```

Liczby zmiennoprzecinkowe – float

Ważniejsze funkcje matematyczne.

- `cos(x)`, `sin(x)`, `tan(x)` – zwraca wartość funkcji dla argumentu `x` w radianach.
- `acos(x)`, `asin(x)`, `atan(x)` – zwraca wartość funkcji w radianach dla argumentu `x`.
- `exp(x)` – oblicza e^x.
- `log(x)` – oblicza logarytm naturalny.
- `log(x,b)` – oblicza logarytm o podstawie `b`.
- `log10(x)` – oblicza logarytm o podstawie 10.

```
>>> math.cos(math.pi)
-1.0
>>> math.log(math.e)
1.0
>>> math.log(256,2)
8.0
```

Liczby zmiennoprzecinkowe – float

- `fabs(x)` – zwraca wartość bezwzględną `x`.
- `sqrt(x)` – zwraca pierwiastek kwadratowy `x`.
- `ceil(x)` – zwraca najmniejszą l. całkowitą większą lub równą `x`.
- `floor(x)` – zwraca największą l. całkowitą mniejszą lub równą `x`.
- `trunc(x)` – zwraca część całkowitą `x`.

```
>>> a = round(23.4)
>>> print(a,type(a))
23 <class 'int'>

>>> a = round(23.42324,2)
>>> print(a,type(a))
23.42 <class 'float'>
```

```
>>> a = math.ceil(-3.2)
>>> a,type(a)
(-3, <class 'int'>)

>>> c = math.trunc(-3.2)
>>> c, type(c)
(-3, <class 'int'>)

>>> a = math.floor(-3.2)
>>> a,type(a)
(-4, <class 'int'>)
```

Liczby zmiennoprzecinkowe – float

- `isinf(x)` – zwraca `True` jeśli `x` jest $\pm\infty$.
- `isnan(x)` – zwraca `True` jeśli `x` jest typu `math.nan`.
- `gcd(x,y)` – zwraca największy wspólny podzielnik liczb `x` i `y`.

Przydatne funkcje klasy `float`.

- `float.is_integer(x)` – zwraca `True` gdy liczba ma zerową część ułamkową.
- `float.as_integer_ratio(x)` – zwraca liczbę jako ułamek (w formie krotki).

```
>>> math.gcd(120,45)
15
>>> float.as_integer_ratio(1.75)
(7, 4)
>>> (0.125).as_integer_ratio()
(1, 8)
```

Liczby zespolone – complex

Typ `complex` to niezmienny typ danych, przechowujący parę liczb typu `float`, z których pierwsza reprezentuje część rzeczywistą, a druga część urojoną liczby zespolonej. Część urojoną definiujemy z literą `j`.

Poszczególne składniki liczby zespolonej dostępne są jako atrybuty `real` i `imag` klasy `complex`.

```
>>> a = 2 + 3j
>>> a
(2+3j)

>>> b = 4 - 2.3j
>>> b, type(b)
((4-2.3j), <class 'complex'>)

>>> c = -5j
>>> c
(-0-5j)
```

```
>>> d = 4+0j
>>> d,type(d)
((4+0j), <class 'complex'>)

>>> b.real
4.0
>>> b.imag
-2.3
```

Liczby zespolone – complex

Z wyjątkiem operatorów // i % pozostałe operatory (także rozszerzone przypisania) mogą być wykorzystywane do działań na liczbach zespolonych.

```
>>> a = 3 + 4j
>>> b = 2 - 5j

>>> z1 = a * b
>>> z1
(26-7j)

>>> z2 = a + b
>>> z2
(5-1j)

>>> z3 = a / 2
>>> z3
(1.5+2j)
```

```
>>> z4 = b / a
>>> z4
(-0.56-0.92j)

>>> z6 = a**2
>>> z6
(-7+24j)

>>> z5 = a**b
>>> z5
(2568.926440363592+233.34785796721815j)
```

```
>>> z8 = a ** 0.5
>>> z8
(2+1j)

>>> z7 = a**-2
>>> z7
(-0.0112-0.0384j)
```

Liczby zespolone – complex

Funkcje modułu `math` nie działają z liczbami zespolonymi!
Należy używać modułu `cmath`, który dla liczb zespolonych udostępnia większość podstawowych funkcji.

```
import math
import cmath

def pierwiastki2(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        x1 = (-b + math.sqrt(delta))/(2*a)
        x2 = (-b - math.sqrt(delta))/(2*a)
    else:
        x1 = (-b + cmath.sqrt(delta))/(2*a)
        x2 = (-b - cmath.sqrt(delta))/(2*a)

    return x1, x2

wynik = pierwiastki2(1,2,1)
print(wynik)

wynik = pierwiastki2(1,1,1)
print(wynik)
```

Liczby zespolone – complex

```
import math
import cmath

def pierwiastki2(a,b,c):
    delta = b**2 - 4*a*c
    if delta >= 0:
        x1 = (-b + math.sqrt(delta))/(2*a)
        x2 = (-b - math.sqrt(delta))/(2*a)
    else:
        x1 = (-b + cmath.sqrt(delta))/(2*a)
        x2 = (-b - cmath.sqrt(delta))/(2*a)

    return x1, x2

wynik = pierwiastki2(1,2,1)
print(wynik)

wynik = pierwiastki2(1,1,1)
print(wynik)

------------------------------------------------
(-1.0, -1.0)
((-0.5+0.8660254037844386j), (-0.5-0.8660254037844386j))
```

Liczby typu Decimal

Moduł `decimal` udostępnia niezmienne liczby typu `Decimal`, które zapewniają dokładność ustaloną przez programistę. Obliczenia na liczbach typu `Decimal` sa znacznie wolniejsze niż na liczbach typu `float`. Liczby tego typu można wiarygodnie porównywać.
Domyślną dokładnością liczb typu `Decimal` jest 28 miejsc po przecinku.

Liczby tworzymy za pomocą funkcji `decimal.Decimal()`, a jej argumentem może być wartość całkowita `int` lub ciąg tekstowy. Aby wykorzystać wartość typu `float` do utworzenia liczby typu `Decimal` możemy użyć specjalnej funkcji `decimal.Decimal.from_float()`.

Większość operatorów arytmetycznych (łącznie z rozszerzonym przypisaniem) działa prawidłowo także dla typu `Decimal`.

Liczby typu Decimal

```
>>> import decimal

>>> a = decimal.Decimal(123)
>>> b = decimal.Decimal('456.789')

>>> print(a, b, type(a))
123 456.789 <class 'decimal.Decimal'>

>>> c = decimal.Decimal.from_float(0.1)
>>> c
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> decimal.getcontext().prec = 6
>>> c = a / b
>>> print(c)
0.269271

>>> decimal.getcontext().prec = 50
>>> c = a / b
>>> print(c)
0.26927093253121244163059968606949817092793390383744
```

Liczby typu Decimal

Moduły `math` i `cmath` nie pracują z liczbami typu `Decimal` jednak dla obiektów tego typu zdefiniowane są własne funkcje (o nazwach podobnych do modułu `math`) wykonujące właściwe obliczenia.

```
>>> getcontext().prec = 28
>>> Decimal(2).sqrt()
Decimal('1.4142135623730950488016887242')
>>> Decimal(1).exp()
Decimal('2.7182818284590452353602874713')
>>> Decimal('10').ln()
Decimal('2.3025850929940456840179914547')
>>> Decimal('10').log10()
Decimal('1')
```

Ciągi tekstowe – typ str

Typ `str` reprezentuje ciągi tekstowe (sekwencje znaków Unicode).

```
>>> 'Przykładowy tekst'
'Przykładowy tekst'
>>> "Źdźbło trawy"
'Źdźbło trawy'
>>> ''
''
```

Jeżeli wewnątrz tekstu chcemy wstawić cytat, to jest to dozwolone o ile tekst i cytat będą ograniczone za pomocą odmiennych znaków.

```
>>> t1 = 'To jest "cytat" w tekście'
>>> t2 = "I to jest 'cytat' w tekście"
>>> t1
'To jest "cytat" w tekście'
>>> t2
"I to jest 'cytat' w tekście"
```

Ciągi tekstowe – typ str

Możemy też użyć tzw. sekwencji sterujących: \' lub \"

```
>>> t3 = 'To jest apostrof: \' w tekście, a to cudzysłów \" '
>>> t3
'To jest apostrof: ' w tekście, a to cudzysłów " '
```

W Pythonie zakończeniem polecenia jest znak nowego wiersza. Nie dotyczy to poleceń umieszczonych w nawiasach `()`, `[]`, `{}` oraz tekstów w potrójnych apostrofach.

```
>>> t4 = '''To jest tekst z 'cytatem'
i z innym "cytatem", który składa się
z trzech \
wierszy'''

>>> print(t4)
To jest tekst z 'cytatem'
i z innym "cytatem", który składa się
z trzech wierszy
```

Ciągi tekstowe – typ str

Ważniejsze sekwencje sterujące.

- `\` – ignoruje znak nowego wiersza
- `\'`, `\"`, `\\` – wstaw odpowiedni znak
- `\n` – wstaw nowy wiersz
- `\t` – wstaw tabulator
- `\N{nazwa}` – wstaw znak Unicode o podanej nazwie
- `\xhh` – wstaw znak Unicode o podanej 8-bitowej wartości szesnastkowej
- `\uhhhh` – wstaw znak Unicode o podanej 16-bitowej wartości szesnastkowej
- `\Uhhhhhhhh` – wstaw znak Unicode o podanej 32-bitowej wartości szesnastkowej

Poprzedzając tekst literą `r` możemy definiować niezmodyfikowane ciągi tekstowe, w których sekwencje sterujące nie działają.

Ciągi tekstowe – typ str

```
>>> '\N{dollar sign}'
'$'
>>> '\N{copyright sign}'
'©'

>>> ord('ń')
324
>>> hex(324)
'0x144'
>>> t = 'Pluci\u0144ski'
>>> t
'Pluciński'

>>> a = 'asd \n \N{dollar sign} \\'
>>> print(a)
asd
 $ \

>>> b = r'asd \n \N{dollar sign} \\'
>>> print(b)
asd \n \N{dollar sign} \\
```

Ciągi tekstowe – porównywanie

Dla ciągów tekstowych poprawnie działają tradycyjne operatory porównania: `<`, `<=`, `==`, `!=`, `>=`, `>`.
Problemy:

- Niektóre znaki mają przypisanych kilka kodów.
- Kolejność sortowania pewnych znaków może być specyficzna dla języka.
- Czasami w zdaniu mogą wystąpić słowa w różnych językach.
- Dla niektórych znaków (np. znaki ozdobne, strzałki, itp.) nie ma sensownych kryteriów sortowania.

Porównywanie jest wykonywane z umieszczonych w pamięci bajtów definiujących ciąg tekstowy. Kolejność sortowania bazuje zatem na kodach Unicode.
Sposób porównywania może być dostosowany do własnych potrzeb.

Ciągi tekstowe – indeksowanie

Sposób numerowania elementów ilustruje tabela.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
P	R	Z	Y	K	Ł	A	D
[-8]	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

Przypisanie indeksu spoza dozwolonych wartości powoduje zgłoszenie wyjątku: `IndexError`.

Ciągi tekstowe – indeksowanie

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
P	R	Z	Y	K	Ł	A	D
[-8]	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

Indeksowanie można zrealizować na kilka sposobów:

- `sekwencja[początek]`
- `sekwencja[początek:koniec]`
- `sekwencja[początek:koniec:krok]`

Sekwencja może być np. ciągiem tekstowym, listą, krotką. Wartości `początek`, `koniec` i `krok` muszą być liczbami całkowitymi (lub zmiennymi przechowującymi liczby całkowite).

Ciągi tekstowe – indeksowanie

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
P	R	Z	Y	K	Ł	A	D
[-8]	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

`sekwencja[początek:koniec]` – wyodrębnia elementy od miejsca `początek` do miejsca `koniec` ale bez(!) tego elementu z krokiem 1.

Poszczególne wartości w indeksowaniu można pomijać. Przyjmują one wtedy wartości domyślne.

- `początek` – 0 gdy krok dodatni, -1 gdy krok ujemny
- `koniec` – -1 gdy krok dodatni, 0 gdy krok ujemny
- `krok` – domyślnie 1

Jeśli koniec jest domyślny, to jest uwzględniany w wyniku indeksowania, np. `a[0::1]`. Jeśli koniec jest podany jawnie, np. `a[0:-1:1]`, w wyniku indeksowania uwzględniony nie będzie.

Ciągi tekstowe – indeksowanie

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
P	R	Z	Y	K	Ł	A	D
[-8]	[-7]	[-6]	[-5]	[-4]	[-3]	[-2]	[-1]

```
>>> t = 'PRZYKŁAD'
>>> t[3]
'Y'
>>> t[0]
'P'
>>> t[-1]
'D'
>>> t[0:7]
'PRZYKŁA'
>>> t[-1:-8]
''
>>> t[:5]
'PRZYK'
>>> t[2:]
'ZYKŁAD'
```

```
>>> t[:]
'PRZYKŁAD'
>>> t[-1:-8:-1]
'DAŁKYZR'
>>> t[-1::-1]
'DAŁKYZRP'
>>> t[::-1]
'DAŁKYZRP'
>>> t[0::2]
'PZKA'
>>> t[0:7:2]
'PZKA'
>>> t[-1::-3]
'DKR'
```

Ciągi tekstowe – metody

Replikacja:

```
t1 = 'wyraz '
>>> t4 = t1 * 4
>>> t4
'wyraz wyraz wyraz wyraz '
```

Przynależność tekstu można sprawdzać za pomocą operatora `in`, a brak przynależności za pomocą `not in`.

```
>>> zdanie = 'To jest przykładowe zdanie'
>>> 'T' in zdanie
True
>>> 'ą' not in zdanie
True
>>> 'jest' in zdanie
True
>>> 'zdanie ' in zdanie
False
>>> 'wyraz' not in zdanie
True
```

Ciągi tekstowe – metody

- `s.find(t,`*początek*`,`*koniec*`)` – zwraca pierwsze wystąpienie `t` w `s`. Jeśli nie zostanie znalezione, zwraca -1. Ostatnie wystąpienie zwraca funkcja `rfind`.
- `s.index(t,`*początek*`,`*koniec*`)` – zwraca pierwsze wystąpienie `t` w `s`. Jeśli nie zostanie znalezione, zgłasza wyjątek `ValueError`. Ostatnie wystąpienie zwraca funkcja `rindex`.

```
>>> zdanie = 'To jest przykładowe zdanie'
>>> zdanie.find('e')
4

>>> zdanie.find('jest')
3

>>> zdanie.find('nie jest')
-1
```

```
>>> zdanie.index('e')
4

>>> zdanie.index('nie jest')
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <modul
    zdanie.index('nie jest')
ValueError: substring not found
```

Ciągi tekstowe – metody

- `s.find(t,`*początek*`,`*koniec*`)` – zwraca pierwsze wystąpienie `t` w `s`. Jeśli nie zostanie znalezione, zwraca -1. Ostatnie wystąpienie zwraca funkcja `rfind`.
- `s.index(t,`*początek*`,`*koniec*`)` – zwraca pierwsze wystąpienie `t` w `s`. Jeśli nie zostanie znalezione, zgłasza wyjątek `ValueError`. Ostatnie wystąpienie zwraca funkcja `rindex`.

```
zdanie = 'Przykład [tekstu w nawiasie] i jego wycinania'
try:
    start = zdanie.index('[')
    stop = zdanie.index(']')
    print(zdanie[start+1:stop])
except ValueError:
    print('Brak nawiasu!')

----------------------------------------
tekstu w nawiasie
```

Ciągi tekstowe – metody

- `s.count(t,`*początek*`,`*koniec*`)` – zwraca liczbę wystąpień `t` w `s` (bądź we wskazanym fragmencie).
- `s.startswith(t,`*początek*`,`*koniec*`)` – zwraca `True` jeśli ciąg tekstowy `s` (lub wskazany fragment) rozpoczyna się ciągiem `t` (bądź dowolnym ciągiem w krotce `t`). W przeciwnym razie zwraca `False`.
- `s.endswith(t,`*początek*`,`*koniec*`)` – zwraca `True` jeśli ciąg tekstowy `s` (lub wskazany fragment) kończy się ciągiem `t` (bądź dowolnym ciągiem w krotce `t`). W przeciwnym razie zwraca `False`.

```
>>> zdanie = 'To nie jest przykładowe zdanie'
>>> zdanie.count('e')
4
>>> zdanie.count('e',0,-10)
2
>>> zdanie.count('nie')
2
```

Ciągi tekstowe – metody

```
>>> zdanie = 'To nie jest przykładowe zdanie'
>>> zdanie.startswith('To')
True
>>> zdanie.startswith('To jest')
False
>>> zdanie.startswith(('To','To '))
True
>>> zdanie.startswith(('To','Nie'))
True
>>> zdanie.startswith(('[','{','('))
False
>>> zdanie.startswith('nie',3)
True
>>> zdanie.startswith('nie',5,-1)
False
>>> zdanie.endswith('nie',5,-1)
False
>>> zdanie.endswith('nie',5)
True
>>> zdanie[5:].endswith('nie')
True
>>>
```

Ciągi tekstowe – metody

- `s.join(sekwencja)` – zwraca połączenie elementów sekwencji z ciągiem tekstowym `s` (który może być pusty).
- `s.partition(t)` – zwraca krotkę trzech ciągów tekstowych: fragment przed pierwszym wystąpieniem `t`, `t` i resztę tekstu. Jeśli `s` nie zawiera `t`, zwracane jest `s` i dwa ciągi puste. Metoda `rpartition` działa podobnie, względem ostatniego wystąpienia `t`.

```
>>> osoby = ['Jan','Adam','Marek']
>>> lista = ' '.join(osoby)
>>> lista
'Jan Adam Marek'

>>> ''.join(osoby)
'JanAdamMarek'

>>> '\n'.join(osoby)
'Jan\nAdam\nMarek'
>>> print('\n'.join(osoby))
Jan
Adam
Marek
```

Ciągi tekstowe – metody

- `s.join(sekwencja)` – zwraca połączenie elementów sekwencji z ciągiem tekstowym `s` (który może być pusty).
- `s.partition(t)` – zwraca krotkę trzech ciągów tekstowych: fragment przed pierwszym wystąpieniem `t`, `t` i resztę tekstu. Jeśli `s` nie zawiera `t`, zwracane jest `s` i dwa ciągi puste. Metoda `rpartition` działa podobnie, względem ostatniego wystąpienia `t`.

```
>>> tekst = 'C:\Marcin\Dokumenty\Kursy\Python\Examples>python znaki.py'
>>> wynik = tekst.partition(' ')
>>> wynik
('C:\\Marcin\\Dokumenty\\Kursy\\Python\\Examples>python', ' ', 'znaki.py')

>>> tekst.partition('.')
('C:\\Marcin\\Dokumenty\\Kursy\\Python\\Examples>python znaki', '.', 'py')

>>> tekst.partition('\\')
('C:', '\\', 'Marcin\\Dokumenty\\Kursy\\Python\\Examples>python znaki.py')
```

Ciągi tekstowe – metody

- `s.split(t)` – zwraca listę ciągów tekstowych, powstałych po podziale `s` przez `t`.
- `s.split(t,n)` – jak wyżej, ale podział dokonywany jest tylko na `n` pierwszych wystąpieniach `t`.
- `s.rsplit(t,n)` – jak wyżej, ale podział dokonywany jest tylko na `n` ostatnich wystąpieniach `t`.
- `s.splitlines()` – zwraca listę wierszy po podziale `s` w miejscach zakończenia linii.

```
>>> tekst = 'C:\Marcin\Dokumenty\Kursy\Python\Examples>python znaki.py'
>>> tekst.split('\\')
['C:', 'Marcin', 'Dokumenty', 'Kursy', 'Python', 'Examples>python znaki.py']

>>> tekst.split('\\',3)
['C:', 'Marcin', 'Dokumenty', 'Kursy\\Python\\Examples>python znaki.py']

>>> 'To jest zdanie do  podziału'.split(' ')
['To', 'jest', 'zdanie', 'do', '', 'podziału']
>>> 'To jest zdanie do  podziału'.split()
['To', 'jest', 'zdanie', 'do', 'podziału']
```

Ciągi tekstowe – metody

- `s.lower()`, `s.upper()`, `s.swapcase()`
- `s.title()` – pierwsza litera każdego słowa duża, reszta małe.
- `s.capitalize()` – tylko pierwsza litera w ciągu jest duża.

```
>>> tekst = 'C:\Marcin\Dokumenty\Kursy\Python\Examples>python znaki.py'
>>> tekst.lower()
'c:\\marcin\\dokumenty\\kursy\\python\\examples>python znaki.py'

>>> tekst.upper()
'C:\\MARCIN\\DOKUMENTY\\KURSY\\PYTHON\\EXAMPLES>PYTHON ZNAKI.PY'

>>> tekst.swapcase()
'c:\\mARCIN\\dOKUMENTY\\kURSY\\pYTHON\\eXAMPLES>PYTHON ZNAKI.PY'

>>> tekst = 'Mam na imię Marcin. Moje miejsce pracy to ZUT.'
>>> tekst.title()
'Mam Na Imię Marcin. Moje Miejsce Pracy To Zut.'

>>> tekst.capitalize()
'Mam na imię marcin. moje miejsce pracy to zut.'
```

Ciągi tekstowe – metody

- `s.center(długość, znak)`, `s.ljust(długość, znak)`, `s.rjust(długość, znak)` – pozycjonowanie ciągu `s` w polu o zadanej długości. `znak` jest opcjonalny.
- `s.strip()`, `s.lstrip()`, `s.rstrip()` – usuwanie znaków odstępu (bądź znaków podanych jako argument).

```
>>> tekst = 'Centrum'
>>> tekst.center(20)
'       Centrum        '

>>> tekst.center(20,'-')
'------Centrum-------'

>>> tekst.ljust(20)
'Centrum              '

>>> tekst.rjust(20)
'              Centrum'
```

```
>>> tekst = '     przykład    '
>>> tekst.strip()
'przykład'

>>> tekst.lstrip()
'przykład    '

>>> tekst = '.....przykład...'
>>> tekst.strip('.')
'przykład'
```

Ciągi tekstowe – metody

- `s.replace(t,u,n)` – każde wystąpienie (bądź opcjonalnie n) ciągu `t` jest zastępowane ciągiem `u`.
- `tablica = s.maketrans(znaki1, znaki2)` – tworzy tablicę konwersji znaków.
- `s.translate(tablica)` – dokonuje konwersji na podstawie utworzonej tablicy.

```
>>> liczby = '1,2  4,5  5,6  7,8'
>>> liczby.replace(',', '.')
'1.2  4.5  5.6  7.8'

>>> liczby.replace(',', '.', 2)
'1.2  4.5  5,6  7,8'
>>>

>>> tablica = ''.maketrans('łŁąĄęĘ','lLaAeE')
>>> 'Język Łaciński'.translate(tablica)
'Jezyk Laciński'
```

Ciągi tekstowe – metody

Metody `is*()` zwracają wartość `True` jeśli ciąg tekstowy nie jest pusty i wszystkie znaki spełniają określone kryterium.

Przykłady poleceń:

```
>>> 'Abc'.islower()
False
>>> 'abc'.islower()
True

>>> 'ABC'.isupper()
True
>>> 'Abc'.isupper()
False

>>> 'abc'.istitle()
False
>>> 'Abc'.istitle()
True
```

```
>>> '123'.isdigit()
True
>>> '12.3'.isdigit()
False

>>> '12.3'.isalpha()
False
>>> 'abc'.isalpha()
True

>>> '12.3'.isalnum()
False
>>> '123xxx'.isalnum()
True

>>> '\t '.isspace()
True
```

Ciągi tekstowe – metoda format()

Metoda `format()` zwraca nowy ciąg tekstowy, w którym *pola zastępcze* zostają zastąpione odpowiednio sformatowanymi argumentami.

```
>>> 'W {0} roku jestem pracownikiem {1}'.format(2019, 'ZUT')
'W 2019 roku jestem pracownikiem ZUT'
```

Każde pole zastępcze jest identyfikowane przez nazwę lub numer w nawiasach {}. Numer odpowiada pozycji na liście argumentów metody `format()`.

Jeśli w tekście chcemy użyć nawiasów klamrowych, musimy zapisać je podwójnie.

```
>>> 'To jest zbiór liczb: {{ {0}, {1}, {2} }}'.format(3, -0.25, 47.087)
'To jest zbiór liczb: { 3, -0.25, 47.087 }'
```

Pola zastępcze można zagnieżdżać.

Ciągi tekstowe – metoda format()

Od Pythona 3.1 nazwy pól można pomijać. Pola wypełniane są wtedy kolejnymi argumentami.

```
>>> 'Temperatura: {}, wilgotność {}%, wiatr {}'.format(17, 75, 'słaby')
'Temperatura: 17, wilgotność 75%, wiatr słaby'
```

Argumenty mogą mieć nazwy. Oba sposoby można łączyć, ale argumenty pozycyjne muszą być zawsze pierwsze.

```
>>> 'W {rok} roku jestem pracownikiem {firma}'.format(firma = 'ZUT', rok = 2019
'W 2019 roku jestem pracownikiem ZUT'

>>> 'W {0} roku jestem pracownikiem {firma}'.format(2019, firma = 'ZUT')
'W 2019 roku jestem pracownikiem ZUT'
```

Ciągi tekstowe – metoda format()

Nazwy pól mogą się też odwoływać do złożonych typów danych jak: listy, krotki, słowniki i inne obiekty.

```
>>> lista = ['słaby', 'umiarkowany', 'silny']
>>> 'W dniu {0} wiatr jest {1[2]}'.format('20.09.2019', lista)
'W dniu 20.09.2019 wiatr jest silny'

>>> z = 12-3j
>>> z.real, z.imag
(12.0, -3.0)
>>> 'Część rzeczywista: {0.real} i urojona: {0.imag} liczby.'.format(z)
'Część rzeczywista: 12.0 i urojona: -3.0 liczby.'
```

Ciągi tekstowe – specyfikacja formatu

Specyfikacja formatu zaczyna się od dwukropka.

Dla ciągów tekstowych podajemy:

- znak wypełnienia (zawsze ze sposobem poniżej)
- sposób wyrównania (<,^,>)
- minimalną szerokość pola
- maksymalną szerokość pola (po kropce)

Wszystkie parametry są opcjonalne.

```
>>> t = 'Student WI ZUT'
>>> len(t)
14
>>> '{0}'.format(t)
'Student WI ZUT'
>>> '{0:25}'.format(t)
'Student WI ZUT           '
>>> '{0:>25}'.format(t)
'           Student WI ZUT'
```

```
>>> '{0:^25}'.format(t)
'     Student WI ZUT      '
>>> '{0:.^25}'.format(t)
'.....Student WI ZUT.....'
>>> '{0:_<25}'.format(t)
'Student WI ZUT__________'
>>> '{0:.10}'.format(t)
'Student WI'
```

Ciągi tekstowe – specyfikacja formatu

Dla liczb całkowitych podajemy:

- znak wypełnienia (zawsze ze sposobem poniżej)
- sposób wyrównania (<,^,>) lub = dla dopełniania zerami
- + wymuszenie znaku lub - gdy konieczny
- # i specyfikator prefiksu przed liczbą: `b`, `o`, `x`, `X`
- minimalną szerokość pola
- przecinek, gdy chcemy grupować cyfry
- specyfikator typu liczby np. `b`, `o`, `x`, `X`, `d` i inne

Wszystkie parametry są opcjonalne.

```
>>> '{0:0=10}'.format(12345)
'0000012345'
>>> '{0:0=10}'.format(-12345)
'-000012345'
>>> '{0:0>10}'.format(-12345)
'0000-12345'
```

```
>>> '{0:.>10}'.format(12345)
'.....12345'
>>> '{0:^10}'.format(12345)
'  12345   '
```

Ciągi tekstowe – specyfikacja formatu

```
>>> '{0:^+10}'.format(12345)
'  +12345  '
>>> '{0:^-10}'.format(12345)
'  12345   '


>>> '{0:#b}'.format(12345)
'0b11000000111001'
>>> '{0:#x}'.format(12347)
'0x303b'
>>> '{0:#X}'.format(12347)
'0X303B'

>>> '{0:20,}'.format(1234567890)
'       1,234,567,890'
>>> '{0:10x}'.format(0xFF)
'        ff'
>>> '{0:10x}'.format(0b110010100)
'       194'
>>> '{0:10b}'.format(0b110010100)
' 110010100'
>>> '{0:10x}'.format(255)
'        ff'
```

Ciągi tekstowe – specyfikacja formatu

Dla liczb zmiennoprzecinkowych formatowanie jest podobne do całkowitych, jednak:

- możemy jeszcze określać ilość miejsc po przecinku (kropka i liczba)
- specyfikatory formatu liczby to: `f`, `F`, `e`, `E`, `g`, `G` i inne

```
>>> import math
>>> liczba = math.exp(10) # 22026.465794806718
>>> mała = 1.0 / liczba   # 4.539992976248485e-05

>>> '{0:10.3}  {1:10.2}'.format(liczba, mała)
'   2.2e+04     4.5e-05'
>>> '{0:10.5}  {1:10.3}'.format(liczba, mała)
'2.2026e+04    4.54e-05'

>>> '{0:10.3f}  {1:10.2F}'.format(liczba, mała)
' 22026.466        0.00'
>>> '{0:10.3e}  {1:10.2E}'.format(liczba, mała)
' 2.203e+04    4.54E-05'

>>> liczba = math.exp(20) # 485165195.4097903
>>> '{0:10,.3f}'.format(liczba)
'485,165,195.410'
```

Ciągi tekstowe – f-string

Od Pythona w wersji 3.6 możliwe jest uproszczone formatowanie ciągu tekstowego zapisanego jako tzw. f-string

```
>>> imie = 'Marcin'
>>> wzrost = 180
>>> s = f'Mam na imię {imie}. Mój wzrost to {wzrost} cm.'
>>> s
'Mam na imię Marcin. Mój wzrost to 180 cm.'

>>> f'Mam na imię {imie.lower()}. Mój wzrost to {10*wzrost+25} cm.'
'Mam na imię marcin. Mój wzrost to 1825 cm.'

>>> s = f'''Mam na imie {imie}
Mój wzrost to {wzrost} cm.'''
>>> s
'Mam na imie Marcin\nMój wzrost to 180 cm.'

>>> print(s)
Mam na imie Marcin
Mój wzrost to 180 cm.
>>>
```

Sekwencje

Sekwencja to typ danych obsługujący:

- operator przynależności `in`
- funkcję określającą rozmiar `len()`
- indeksowanie `[]`
- przeprowadzanie iteracji

Python oferuje wbudowane sekwencje: `str`, `tuple`, `list`, `bytearray`, `bytes` oraz sekwencje dostępne w bibliotece standardowej.

Krotki

Krotka to uporządkowana i niezmienna sekwencja zera lub większej liczby odniesień do obiektów.

Indeksowanie elementów w krotkach jest podobne do typu znakowego. Jeśli chcemy zmodyfikować krotkę, możemy ją skonwertować na listę: `list(krotka)`.

Krotki definiujemy w nawiasach () lub za pomocą funkcji `tuple`.

```
>>> a = ('abc', 'def', 'ijk')
>>> a
('abc', 'def', 'ijk')
>>> b = (1,3,5,7)
>>> b
(1, 3, 5, 7)
>>> c = ('abc', 1, 3.987, 'x')
>>> c
('abc', 1, 3.987, 'x')
```

```
>>> d = ()
>>> d
()
>>> d = tuple()
>>> d
()
>>> d = (5,)
>>> d
(5,)
```

Krotki

Krotki oferują dwie metody.

- `t.count(x)` – zwraca liczbę wystąpień obiektu `x` w krotce `t`.
- `t.index(x)` – zwraca indeks pierwszego wystąpienia obiektu `x` w krotce `t`. Jeśli obiektu `x` nie ma, zgłaszany jest wyjątek `ValueError`.

```
>>> t = (1,2,3,4,3,2,1,2,3,4)
>>> t.count(3)
3
>>> t.index(4)
3
```

```
>>> t[-5:-1]
(2, 1, 2, 3)
>>> t[-5:-1].index(2)
0
```

Działają operatory: `+`, `*`, `+=`, `*=`, `in`, `not in` oraz operatory porównania `==`, `!=`, `>`, `>=`, `<`, `<=`. Porównywanie jest przeprowadzane element po elemencie.

Krotki

```
>>> t = ('Szczecin', 'Łódź', 'Koszalin', 'Piła')

>>> t1 = t + 'Warszawa'
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    t1 = t + 'Warszawa'
TypeError: can only concatenate tuple (not "str") to tuple

>>> t1 = t + ('Warszawa',)
>>> t1
('Szczecin', 'Łódź', 'Koszalin', 'Piła', 'Warszawa')

>>> t1[-3:]
('Koszalin', 'Piła', 'Warszawa')
>>> t1[:4]
('Szczecin', 'Łódź', 'Koszalin', 'Piła')
```

Krotki

Krotki (i inne kolekcje) możemy zagnieżdżać w sobie do dowolnego poziomu głębokości. Operator indeksowania [] może być wtedy używany tyle razy ile jest to konieczne.

```
>>> t = (1,2,3,(1.0,2.0,('trzy', 'cztery', 'pięć')))
>>> t
(1, 2, 3, (1.0, 2.0, ('trzy', 'cztery', 'pięć')))

>>> t[3]
(1.0, 2.0, ('trzy', 'cztery', 'pięć'))

>>> t[:-1]
(1, 2, 3)

>>> t[3][:-1]
(1.0, 2.0)
>>> t[3][1:]
(2.0, ('trzy', 'cztery', 'pięć'))

>>> t[3][2][1:]
('cztery', 'pięć')
```

Krotki

W wielu wypadkach nawiasy w zapisie krotek mogą być pomijane. Ich używanie może zależeć od przyjętej konwencji kodowania.

```
>>> imie, wiek, płeć = ('Jan', 25, 'M')
>>> (imie, wiek, płeć) = ('Jan', 25, 'M')

>>> a, b = 1, 2
>>> print(a,b)
1 2
>>> a, b = (b, a)
>>> print(a,b)
2 1

############################################################
def funkcja(x):
    return x, 2*x

print(funkcja(5))  # (5, 10)

############################################################
for (a,b) in ((1,2), ('a','b'), (3.0, 4.5)):
    print(a,b)
```

Listy

Lista jest uporządkowaną i zmienną sekwencją zera lub większej liczby odniesień do obiektów.

Indeksowanie elementów w listach jest podobne do typu znakowego. Działają operatory: `+`, `*`, `+=`, `*=`, `in`, `not in` oraz operatory porównania `==`, `!=`, `>`, `>=`, `<`, `<=`. Porównywanie jest przeprowadzane element po elemencie.

Listy definiujemy w nawiasach [] lub za pomocą funkcji `list`.

```
>>> a = ['abc', 'def', 'ijk']
>>> a
['abc', 'def', 'ijk']
>>> b = [1,3,5,7]
>>> b
[1, 3, 5, 7]
>>> c = ['abc', 1, 3.987, 'x']
>>> c
['abc', 1, 3.987, 'x']
```

```
>>> d = []
>>> d
[]
>>> d = list()
>>> d
[]
>>> d = [5]
>>> d
[5]
```

Listy – metody

- `L.append(x)` – dodaje element `x` na końcu listy.
- `L.extend(s)` – dodaje elementy z sekwencji `s` na końcu listy.
- `L += m` – działanie identyczne jak powyżej.

```
>>> L = [1, 2, 3, 4]
>>> L.append('pięć')
>>> L
[1, 2, 3, 4, 'pięć']
>>> L.append([7.0, 8.0])
>>> L
[1, 2, 3, 4, 'pięć', [7.0, 8.0]]
>>> L.extend([7.0, 8.0])
>>> L
[1, 2, 3, 4, 'pięć', [7.0, 8.0], 7.0, 8.0]
>>> L += (9,)
>>> L
[1, 2, 3, 4, 'pięć', [7.0, 8.0], 7.0, 8.0, 9]
```

Listy – metody

- `L.count(x)` – zwraca liczbę wystąpień obiektu `x` w liście `L`.
- `L.index(x)` – zwraca indeks pierwszego wystąpienia obiektu `x` w liście `L`. Jeśli obiektu `x` nie ma, zgłaszany jest wyjątek `ValueError`. Można dodać opcjonalnie: `L.index(x, start, koniec)`.
- `L.insert(i, x)` – wstawia obiekt `x` w miejsce `i`.

```
>>> L = [0,1,2,3,4]
>>> L.insert(3,'nowy')
>>> L
[0, 1, 2, 'nowy', 3, 4]
```

Listy – metody

- `L.reverse()` – odwraca kolejność elementów listy `L`.
- `L.sort()` – sortuje listę `L`. Można podawać klucz sortowania.

```
>>> L = ['aaa', 'Bbb', 'xyz', 'ccCC', 'DDDDDD']

>>> L.reverse()
>>> L
['DDDDDD', 'ccCC', 'xyz', 'Bbb', 'aaa']

>>> L.sort()
>>> L
['Bbb', 'DDDDDD', 'aaa', 'ccCC', 'xyz']

>>> L.sort(key = str.lower)
>>> L
['aaa', 'Bbb', 'ccCC', 'DDDDDD', 'xyz']
```

Listy – metody

- `L.reverse()` – odwraca kolejność elementów listy `L`.
- `L.sort()` – sortuje listę `L`. Można podawać klucz sortowania.

```
>>> L = [1, 2, 3, 'cztery']

>>> L.sort()
Traceback (most recent call last):
  File "<pyshell#153>", line 1, in <module>
    L.sort()
TypeError: '<' not supported between instances of 'str' and 'int
```

Listy – usuwanie elementów

- `L.pop()` – zwraca i usuwa ostatni element z listy `L`.
- `L.pop(i)` – zwraca i usuwa z listy `L` element o indeksie `i`.
- `L.remove(x)` – usuwa ostatnie wystąpienie obiektu `x` w liście `L`. Jeśli obiektu `x` nie ma, zgłaszany jest wyjątek `ValueError`.

```
>>> L = [0,1,2,3,4,5,6,7]
>>> L.pop()
7
>>> L
[0, 1, 2, 3, 4, 5, 6]

>>> L.pop(2)
2
>>> L
[0, 1, 3, 4, 5, 6]

>>> L.remove(3)
>>> L
[0, 1, 4, 5, 6]
```

Listy – usuwanie elementów

Do usuwania elementów można także wykorzystać polecenie `del` i listy puste `[]`.

```
>>> L = [0,1,2,3,4,5,6,7]
>>> del L[6]
>>> L
[0, 1, 2, 3, 4, 5, 7]

>>> L = [0,1,2,3,4,5,6,7]
>>> del L[2:5]
>>> L
[0, 1, 5, 6, 7]

>>> L = [0,1,2,3,4,5,6,7]
>>> del L[1::2]
>>> L
[0, 2, 4, 6]
```

```
>>> L = [0,1,2,3,4,5,6,7]
>>> L[6] = []
>>> L
[0, 1, 2, 3, 4, 5, [], 7]

>>> L = [0,1,2,3,4,5,6,7]
>>> L[2:5] = []
>>> L
[0, 1, 5, 6, 7]

>>> L = [0,1,2,3,4,5,6,7]
>>> L[1::2] = []
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    L[1::2] = []
ValueError: attempt to assign sequence of size
```

Listy – wstawianie elementów

- `L[i] = 'nowy'` – zastępuje element listy `L` nową wartością.
- Możemy też zastępować nowymi wartościami całe segmenty elementów.
- Wskazany segment i zastępująca go grupa lista elementów nie muszą mieć tej samej długości.
- W czasie zastępowania, najpierw usuwany jest wskazany segment, a w jego miejsce wstawiana jest nowa lista.

```
>>> L = [0,1,2,3,4,5,6]
>>> L[2:5] = ['dwa','trzy']
>>> L
[0, 1, 'dwa', 'trzy', 5, 6]
```

Listy składane

Listy o wielu elementach możemy tworzyć w sposób programowy.

- Listy zawierające ciągi liczb całkowitych często tworzy się za pomocą konwersji `list(range(...))`.
- Możemy też tworzyć listy składane za pomocą składni:

```
[wyrażenie for element in iteracja]
[wyrażenie for element in iteracja if warunek]
```

Odpowiada to fragmentowi kodu:

```
L = []
for element in iteracja:
    if warunek:
        L.append(wyrażenie)
```

Listy składane mogą się zagnieżdżać.

Listy składane

```
tekst = 'To Jest Przykład'

L = [ord(znak) for znak in tekst]
print(L)

L = [chr(ord(znak)) for znak in tekst]
print(L)

L = [chr(ord(znak)+1) for znak in tekst]
print(L)
print(''.join(L))

L = [chr(ord(znak)) for znak in tekst if znak.isupper()]
print(L)
print(''.join(L))

##################################################################
[84, 111, 32, 74, 101, 115, 116, 32, 80, 114, 122, 121, 107, 322, 97, 100]
['T', 'o', ' ', 'J', 'e', 's', 't', ' ', 'P', 'r', 'z', 'y', 'k', 'ł', 'a', 'd'
['U', 'p', '!', 'K', 'f', 't', 'u', '!', 'Q', 's', '{', 'z', 'l', 'Ń', 'b', 'e'
Up!Kftu!Qs{zlŃbe
['T', 'J', 'P']
TJP
```

Listy składane

```
litery = 'ABCD'
cyfry = '1234'
L = [l + c for l in litery for c in cyfry]
print(L)

##########################################################################

['A1', 'A2', 'A3', 'A4', 'B1', 'B2', 'B3', 'B4', 'C1', 'C2', 'C3', 'C4', 'D1',
'D2', 'D3', 'D4']
```

Zbiory – typ set

Zbiór (`set`) jest kolekcją, która obsługuje operator sprawdzania przynależności (`in`), obliczanie wielkości (`len`) i umożliwia iterację (nie jest jednak określona kolejność elementów).

Biblioteka standardowa udostępnia dwa typy zbiorów: `set` i `frozenset`. Zbiór `set` jest nieuporządkowaną i modyfikowalną kolekcją zera lub większej liczby odniesień do obiektów (które muszą generować wartość `hash`). Ponieważ są nieuporządkowane – nie jest możliwe indeksowanie elementów.

Do zbioru można dodawać niezmienne typy danych jak np. `int`, `float`, `str`, `tuple`, `frozenset`. Nie można dodawać typów zmiennych jak: `list` czy `set`.

Zbiory zawsze zawierają unikalne elementy, a definiujemy je w nawiasach `{ }`.

```
>>> s = {'abc', 1, 3.987, 'x'}
>>> s
{1, 'abc', 3.987, 'x'}
```

Zbiory – typ set

Puste nawiasy { } tworzą słownik. Pusty zbiór tworzymy za pomocą: `set()`.

Zbiory często są używane do usuwania duplikatów elementów (możliwa zmiana kolejności!).

```
>>> s = {}
>>> type(s)
<class 'dict'>

>>> s = set()

>>> s = {1,2,3,4,3,2,1}
>>> s
{1, 2, 3, 4}

>>> s = set('To jest nowe zdanie')
>>> s
{'d', 'i', 'T', 's', 'e', 'z', 'j', 'w', 'o', 'n', 'a', ' ', 't'}

>>> lista = [1,2,3,4,5,3,2,3,2,1,1,2,3,4,1]
>>> list(set(lista))
[1, 2, 3, 4, 5]
```

Zbiory – metody

- `s.add(x)` – dodaje element `x` do zbioru `s`, o ile nie ma go już w zbiorze.
- `s.copy()` – zwraca kopię (płytką) zbioru `s`.

```
>>> s = {1,2,3,4}
>>> s.add(4)
>>> s
{1, 2, 3, 4}

>>> s.add(5)
>>> s
{1, 2, 3, 4, 5}

>>> s1 = s.copy()
>>> s1
{1, 2, 3, 4, 5}
>>> s is s1
False
```

Zbiory – metody

- `s.clear()` – usuwa wszystkie elementy zbioru `s`.
- `s.discard(x)` – usuwa element `x` ze zbioru `s`, o ile jest on w zbiorze.
- `s.remove(x)` – usuwa element `x` ze zbioru `s` lub zgłasza wyjątek `KeyError` jeśli elementu nie ma w zbiorze.
- `s.pop()` – zwraca i usuwa losowy element ze zbioru `s` lub zgłasza wyjątek `KeyError` jeśli zbiór jest pusty.

```
>>> s = {1,2,3,4,5}
>>> s.clear()
>>> s
set()

>>> s = {'a','b','c','d'}
>>> s.pop()
'd'
>>> s
{'c', 'a', 'b'}
```

Zbiory – metody

```
>>> s = 'abcdef'
>>> s = set('abcdef')
>>> s
{'d', 'c', 'f', 'e', 'b', 'a'}

>>> s.discard('a')
>>> s
{'d', 'c', 'f', 'e', 'b'}

>>> s.discard('x')
>>> s
{'d', 'c', 'f', 'e', 'b'}

>>> s.remove('b')
>>> s
{'d', 'c', 'f', 'e'}

>>> s.remove('x')
Traceback (most recent call last):
  File "<pyshell#103>", line 1, in <module>
    s.remove('x')
KeyError: 'x'
```

Zbiory – metody

- `s.union(t)` – zwraca nowy zbiór równy sumie `s` i `t`.
- `s | t`
- `s.update(t)` – do zbioru `s` dodaje elementy `t`, których nie ma w `s`.
- `s |= t`

```
>>> s = {1,2,3,4,5}
>>> t = {3,4,5,6,7}

>>> s1 = s.union(t)
>>> s1
{1, 2, 3, 4, 5, 6, 7}

>>> s.update(t)
>>> s
{1, 2, 3, 4, 5, 6, 7}
```

```
>>> s = {1,2,3,4,5}
>>> s2 = s | t
>>> s2
{1, 2, 3, 4, 5, 6, 7}

>>> s |= t
>>> s
{1, 2, 3, 4, 5, 6, 7}
```

Zbiory – metody

- `s.intersection(t)` – zwraca nowy zbiór równy części wspólnej `s` i `t`.
- `s & t`
- `s.intersection_update(t)` – od zbioru `s` odejmie elementy, których nie ma w `t`.
- `s &= t`

```
>>> s = {1,2,3,4,5}
>>> t = {3,4,5,6,7}

>>> s1 = s.intersection(t)
>>> s1
{3, 4, 5}

>>> s.intersection_update(t)
>>> s
{3, 4, 5}
```

```
>>> s = {1,2,3,4,5}
>>> s2 = s & t
>>> s2
{3, 4, 5}

>>> s &= t
>>> s
{3, 4, 5}
```

Zbiory – metody

- `s.difference(t)` – zwraca nowy zbiór złożony z elementów `s`, których nie ma w `t`.
- `s - t`
- `s.difference_update(t)` – od zbioru `s` odejmie elementy, które są w `t`.
- `s -= t`

```
>>> s = {1,2,3,4,5}
>>> t = {3,4,5,6,7}

>>> s1 = s.difference(t)
>>> s1
{1, 2}

>>> s2 = s - t
>>> s2
{1, 2}
```

Zbiory – metody

- `s.symmetric_difference(t)` – zwraca nowy zbiór złożony z elementów `s` i `t`, ale wyklucza te które są w obu.
- `s ^ t`
- `s.symmetric_difference_update(t)` – w zbiorze `s` znajdzie się symetryczna różnica `s` i `t`.
- `s ^= t`

```
>>> s = {1,2,3,4,5}
>>> t = {3,4,5,6,7}

>>> s1 = s.symmetric_difference(t)
>>> s1
{1, 2, 6, 7}

>>> s2 = s ^ t
>>> s2
{1, 2, 6, 7}
```

Zbiory – metody

- `s.issubset(t)` – zwraca `True`, jeśli `s` jest podzbiorem `t` lub są równe.
- `s <= t`
- `s < t`
- `s.issuperset(t)` – zwraca `True`, jeśli `s` jest nadzbiorem `t` lub są równe.
- `s >= t`
- `s > t`
- `s.isdisjoint(t)` – zwraca `True`, jeśli zbiory `s` i `t` nie mają elementów wspólnych.

Zbiory – metody

```
>>> s = {1,2,3,4,5}
>>> t = {3,4,5,6,7}
>>> x = {3,4,5}
>>> y = {8,9}

>>> s < t
False

>>> s.isdisjoint(y)
True

>>> x <= t
True

>>> s > x
True
```

Zbiory – metody

Do porównywania zbiorów można wykorzystywać operatory: == i !=

```
>>> z1 = {1,'dwa',3,'IV'}
>>> z1
{'IV', 1, 3, 'dwa'}

>>> z2 = {1,3,'IV','dwa'}
>>> z2
{'IV', 1, 3, 'dwa'}

>>> z1 == z2
True
```

Zbiory składane

Zbiory o wielu elementach możemy tworzyć w sposób programowy jako zbiory składane:

```
{wyrażenie for element in iteracja}
{wyrażenie for element in iteracja if warunek}
```

Odpowiada to fragmentowi kodu:

```
Z = set()
for element in iteracja:
    if warunek:
        Z.add(wyrażenie)
```

Zbiory składane mogą się zagnieżdżać.

Zbiory składane

```
>>> tekst = 'To Jest Przykład Zdania'

>>> Z = {ord(znak) for znak in tekst}
>>> Z
{32, 97, 322, 100, 101, 90, 105, 74, 107, 110, 111, 80, 114, 115, 84,
116, 121, 122}

>>> Z1 = {chr(ord(znak)) for znak in tekst}
>>> Z1
{'y', 's', ' ', 't', 'n', 'z', 'ł', 'P', 'k', 'i', 'a', 'e', 'J', 'T',
'o', 'd', 'r', 'Z'}

>>> Z2 = {chr(ord(znak)) for znak in tekst if znak.isupper()}
>>> Z2
{'J', 'T', 'P', 'Z'}
```

Typ frozenset

- Typ danych `frozenset` to zbiór, który nie może być modyfikowany.
- Zbiory tego typu muszą być tworzone z użyciem funkcji `frozenset()` bez argumentów lub z co najwyżej jednym argumentem.
- Ponieważ zbiory `frozenset` są niezmienne, obsługują tylko te metody i operatory, które ich nie modyfikują.

Słowniki

Słownik (`dict`) jest kolekcją, która obsługuje operator sprawdzania przynależności (`in`), obliczanie wielkości (`len`) i umożliwia iterację (nie jest jednak określona kolejność elementów).
Słownik jest kolekcją par elementów klucz–wartość i zapewnia uzyskanie dostępu do elementów oraz ich kluczy i wartości.

Biblioteka standardowa udostępnia typ wbudowany: `dict`, a w bibliotece `collections` mamy typy: `defaultdict` i `OrderedDict`.

Tylko niezmienne typy danych jak np. `int`, `float`, `str`, `tuple`, `frozenset` mogą być używane jako klucze słownika (muszą generować wartość `hash`). Wartość słownika może być dowolnego typu.

Do porównywania słowników można wykorzystywać operatory: `==` i `!=`
Operatorów: `<`, `<=`, `>`, `>=` do słowników nie używamy.

Słowniki

Słownik `dict` jest nieuporządkowaną i modyfikowalną kolekcją zera lub większej liczby par klucz–wartość, w której klucze to odniesienia do obiektów niezmiennych, a wartości to odniesienia do obiektów dowolnego typu.

Klucze słownika muszą być **unikalne**.

Słowniki definiujemy je w nawiasach `{ }` lub za pomocą funkcji `dict()`.

Słowniki

```
>>> d1 = {}
>>> type(d1)
<class 'dict'>

>>> d2 = dict()

>>> d3 = {'Imię':'Jan', 'Nazwisko':'Kowal', 'wiek':20}
>>> d3
{'Imię': 'Jan', 'Nazwisko': 'Kowal', 'wiek': 20}

>>> d4 = dict([('Imię','Jan'), ('Nazwisko','Kowal'), ('wiek',20)])

>>> d5 = dict(zip(('Imię','Nazwisko','wiek'),('Jan','Kowal',20)))

>>> d6 = dict(Imię='Jan', Nazwisko='Kowal', wiek=20)
```

Słowniki

W nawiasach kwadratowych podajemy klucz słownika w celu uzyskania dostępu do wartości:

```
>>> d3 = {'Imię':'Jan', 'Nazwisko':'Kowal', 'wiek':20}

>>> d3['Imię']
'Jan'
>>> d3['wiek']
20

>>> d3['Płeć']
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    d3['Płeć']
KeyError: 'Płeć'
```

Słowniki

Ponieważ klucze są unikalne, polecenie:

```
>>> d3['wiek'] = 30
```

```
>>> d3
{'Imię': 'Jan', 'Nazwisko': 'Kowalski', 'wiek': 30}
```

zastępuje starą wartość nową wartością.

Aby dodać element do słownika, możemy przypisać wartość używając nieistniejącego klucza:

```
>>> d3['Plec'] = 'M'
>>> d3
{'Imię': 'Jan', 'Nazwisko': 'Kowalski', 'wiek': 30, 'Plec': 'M'}
```

Słowniki – usuwanie elementów

- `del d[klucz]` – usuwa ze słownika `d` element o wskazanym kluczu lub zgłasza wyjątek `KeyError`
- `d.clear()` – usuwa wszystkie elementy ze słownika `d`
- `d.pop(klucz)` – zwraca wartość przypisaną do klucza i usuwa element (jeśli `klucz` nie istnieje: wyjątek `KeyError`)
- `d.pop(klucz,v)` – zwraca wartość przypisaną do klucza i usuwa element; jeśli `klucz` nie istnieje zwracana jest wartość `v`
- `d.popitem()` – zwraca i usuwa dowolną parę (klucz–wartość) lub zgłasza wyjątek `KeyError` jeśli słownik jest pusty

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> del d[4]
>>> d
{1: 'I', 2: 'II', 3: 'III', 5: 'V'}
>>> d.clear()
>>> d
{}
```

Słowniki – usuwanie elementów

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> d.pop(4)
'IV'
>>> d
{1: 'I', 2: 'II', 3: 'III', 5: 'V'}

>>> d.pop(10,'X')
'X'
>>> d
{1: 'I', 2: 'II', 3: 'III', 5: 'V'}

>>> d.popitem()
(5, 'V')
>>> d
{1: 'I', 2: 'II', 3: 'III'}
```

Słowniki – metody

- `d.copy()` – zwraca kopię (płytką) słownika `d`
- `d.fromkeys(s,v)` – zwraca słownik, którego klucze są w sekwencji `s`, a wartości to `None` lub `v` (o ile zostało podane)
- `d.update(a)` – dodaje do słownika `d` każdą parę ze słownika `a`. Jeśli klucze się powielają wartości z `a` zastępują wartości z `d`. `a` może mieć formę argumentów w postaci słów kluczowych.

```
>>> d = {}.fromkeys([1,2,3,4])
>>> d
{1: None, 2: None, 3: None, 4: None}

>>> d = {0:'zero',1:'jeden'}.fromkeys([2,3,4],'X')
>>> d
{2: 'X', 3: 'X', 4: 'X'}

>>> d = {}.fromkeys('Tekst',1)
>>> d
{'T': 1, 'e': 1, 'k': 1, 's': 1, 't': 1}
```

Słowniki – metody

- `d.copy()` – zwraca kopię (płytką) słownika `d`
- `d.fromkeys(s,v)` – zwraca słownik, którego klucze są w sekwencji `s`, a wartości to `None` lub `v` (o ile zostało podane)
- `d.update(a)` – dodaje do słownika `d` każdą parę ze słownika `a`. Jeśli klucze się powielają, wartości z `a` zastępują wartości z `d`. `a` może mieć formę argumentów w postaci słów kluczowych.

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> d.update({6:'VI',7:'VII'})
>>> d
{1: 'I', 2: 'II', 3: 'III', 4: 'IV', 5: 'V', 6: 'VI', 7: 'VII'}

>>> d.update(nowy='???')
>>> d
{1: 'I', 2: 'II', 3: 'III', 4: 'IV', 5: 'V', 6: 'VI', 7: 'VII',
'nowy': '???'}
```

Słowniki – metody

- `d.get(klucz)` – zwraca wartość przypisaną kluczowi lub `None` jeśli klucza nie ma
- `d.get(klucz,v)` – zwraca wartość przypisaną kluczowi lub `v` jeśli klucza nie ma
- `d.setdefault(klucz,v)` – działa jak `get()`. Jeśli klucza nie ma w słowniku zostanie wstawiony nowy element o podanym kluczu i wartości `None` lub `v` jeśli zostało podane.

Słowniki – metody

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> d.get(4)
'IV'
>>> d[4]
'IV'

>>> d.get(7)
>>>

>>> d.get(10,'X')
'X'
>>> d
{1: 'I', 2: 'II', 3: 'III', 4: 'IV', 5: 'V'}

>>> d.setdefault(10,'X')
'X'
>>> d
{1: 'I', 2: 'II', 3: 'III', 4: 'IV', 5: 'V', 10: 'X'}
```

Słowniki – metody

```
tekst = '''Installing Python is generally easy, and nowadays many Linux and
UNIX distributions include a recent Python. Even some Windows computers
(notably those from HP) now come with Python already installed.'''

litery = {}

for znak in tekst:
    litery[znak] = litery.get(znak,0) + 1

print(litery)

lista = tekst.split()
słowa = {}
for s in lista:
    słowa[s] = słowa.get(s,0) + 1

print(słowa)
```

```
{'I': 2, 'n': 19, 's': 11, 't': 12, 'a': 13, 'l': 9, 'i': 10, 'g': 2, ' ': 29,
{'Installing': 1, 'Python': 2, 'is': 1, 'generally': 1, 'easy,': 1, 'and': 2, '
```

Słowniki – metody

- `d.items()` – zwraca widok wszystkich par (klucz–wartość) w słowniku
- `d.keys()` – zwraca widok wszystkich kluczy w słowniku
- `d.values()` – zwraca widok wszystkich wartości w słowniku

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> k = d.keys()
>>> k
dict_keys([1, 2, 3, 4, 5])

>>> v = d.values()
>>> v
dict_values(['I', 'II', 'III', 'IV', 'V'])

>>> i = d.items()
>>> i
dict_items([(1, 'I'), (2, 'II'), (3, 'III'), (4, 'IV'), (5, 'V')
```

Słowniki – metody

Jeśli słownik, dla którego utworzyliśmy widok, ulegnie zmianie, widok odzwierciedli tę zmianę.

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> vk = d.keys()
>>> vk
dict_keys([1, 2, 3, 4, 5])

>>> del d[5]

>>> d
{1: 'I', 2: 'II', 3: 'III', 4: 'IV'}
>>> vk
dict_keys([1, 2, 3, 4])
```

Słowniki – metody

Widoki są obiektami umożliwiającymi iterację.

```
d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}

for klucz in d.keys():
    print(klucz)

for klucz in d:
    print(klucz)

for wart in d.values():
    print(wart)

for obiekt in d.items():
    print(obiekt[0], obiekt[1], obiekt)
```

Słowniki – metody

Dla widoków można używać operatorów: `&`, `|`, `-` , `^`, `in` i wykonywać na nich operacje tak jak na zbiorach.

```
>>> d = {1:'I', 2:'II', 3:'III', 4:'IV', 5:'V'}
>>> vk = d.keys()

>>> 4 in vk
True

>>> s = set([1,2,5,10,11,20])
>>> s
{1, 2, 5, 10, 11, 20}

>>> które_klucze_są = vk & s
>>> które_klucze_są
{1, 2, 5}
```

Słowniki składane

Słowniki o wielu elementach możemy tworzyć w sposób programowy jako słowniki składane:

```
{wyrażenie_klucza:wyrażenie_wartości for element in iteracja}
{wyrażenie_klucza:wyrażenie_wartości for element in iteracja if warunek}
```

Odpowiada to fragmentowi kodu:

```
d = {}
for element in iteracja:
    if warunek:
        d[wyrażenie_klucza] = wyrażenie_wartości
```

Słowniki składane

```
{wyrażenie_klucza:wyrażenie_wartości for element in iteracja if warunek}

tekst = '''Installing Python is generally easy, and nowadays many Linux and
UNIX distributions include a recent Python. Even some Windows computers
(notably those from HP) now come with Python already installed.'''

zbiór = set(tekst)
print(zbiór)

d = {l:ord(l) for l in zbiór if l.isupper()}
print(d)

-------------------------------------------------------------------

{'n', 'L', 'r', 'i', 'P', 'W', '(', 'x', 'H', 'a', 'e', ',', 'f', 'l', 'g', 'm'
{'L': 76, 'P': 80, 'W': 87, 'H': 72, 'E': 69, 'N': 78, 'U': 85, 'I': 73, 'X': 8
```

Kolekcje – wbudowane funkcje Pythona

- `all(x)` – zwraca `True` jeśli każdy element kolekcji `x` ma wartość `True`
- `any(x)` – zwraca `True` jeśli choć jeden element kolekcji `x` ma wartość `True`

```
>>> a = [1,2,'',4]
>>> all(a)
False
>>> any(a)
True
```

```
>>> b = [3,2.5,0.0,'abc']
>>> all(b)
False
>>> any(b)
True
```

```
>>> c = [1,-0.3,'a',(2,3)]
>>> all(c)
True
>>> any(c)
True
```

Kolekcje – wbudowane funkcje Pythona

- `min`(x,*key*) – zwraca najmniejszy element kolekcji `x`
- `max`(x,*key*) – zwraca największy element kolekcji `x`
- `sum`(x,*początek*) – zwraca sumę elementów (jeśli można ją policzyć)

```
>>> s = {5,2,0,-2.3}
>>> min(s)
-2.3
>>> max(s)
5

>>> sum(s)
4.7
```

```
>>> t = (2,-6.2,3,-4,0.3)
>>> min(t)
-6.2
>>> max(t)
3
>>> min(t, key=abs)
0.3
>>> max(t, key=abs)
-6.2
```

Kolekcje – wbudowane funkcje Pythona

- `sorted`(x,*key*,*reverse*) – zwraca listę elementów kolekcji `x` w kolejności sortowania. Dla `reverse=True` kolejność jest odwrócona. Jeśli kolekcja zawiera kolekcje, sortowanie działa rekurencyjnie do dowolnego poziomu.

```
def kwadrat(x):
    return x*x

lista = [1,-3,2,5,-6,2,4]

l2 = sorted(lista)
print(l2)
l3 = sorted(lista, reverse=True)
print(l3)
l4 = sorted(lista, key=kwadrat)
print(l4)
```

```
[-6, -3, 1, 2, 2, 4, 5]
[5, 4, 2, 2, 1, -3, -6]
[1, 2, 2, -3, 4, 5, -6]
```

```
lista = [1,5.2,'-0.3',5,'2','3.5']
l5 = sorted(lista, key=float)
print(l5)
```

```
['-0.3', 1, '2', '3.5', 5, 5.2]
```

Iteratory

- Iterator to obiekt pozwalający na iterację.
- Dla iteratorów działa wbudowana funkcja `next(x)`, która zwraca po kolei poszczególne elementy i zwraca wyjątek `StopIteration`, gdy nie ma elementów do zwrócenia.
- Jeśli obiekt `x` nie umożliwia iteracji, można spróbować utworzyć dla niego iterator wbudowaną funkcją `iter(x)`.

```
for i in [1,3,6,8]:
    print(i)
```

```
x = iter([1,3,6,8])
while True:
    try:
        print(next(x))
    except StopIteration:
        break
```

Iteratory

- `reversed(x)` – zwraca iterator, w którym elementy `x` są w odwrotnej kolejności.
- `zip(x1, ... , xN)` - zwraca iterator krotek, powstałych przez połączenie elementów w `x1` do `xN`.

```
>>> x = [1,2,3,4]
>>> print(x, type(x))
[1, 2, 3, 4] <class 'list'>
>>> y = reversed(x)
>>> print(y)
<list_reverseiterator object at 0x000002746B6196A0>

>>> next(y)
4
.........
>>> next(y)
1
>>> next(y)
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    next(y)
StopIteration
```

Iteratory

- `reversed(x)` – zwraca iterator, w którym elementy `x` są w odwrotnej kolejności.
- `zip(x1, ... , xN)` - zwraca iterator krotek, powstałych przez połączenie elementów w `x1` do `xN`.

```
>>> z = zip(range(5), range(1,4))
>>> z
<zip object at 0x000002746B637AC8>
>>> for t in z:
        print(t)

(0, 1)
(1, 2)
(2, 3)
>>> z = zip(range(5), range(1,4), [9,8,7,6])
>>> for t in z:
        print(t)

(0, 1, 9)
(1, 2, 8)
(2, 3, 7)
```

Kopiowanie kolekcji

Operator przypisania (=) tworzy jedynie kopię odniesienia do obiektu (przez co, tego typu kopiowanie jest bardzo efektywne).

```
>>> lista = [1,2,3,4,5]
>>> lista2 = lista

>>> print(lista, lista2)
[1, 2, 3, 4, 5] [1, 2, 3, 4, 5]

>>> lista[4] = 'abc'
>>> print(lista, lista2)
[1, 2, 3, 4, 'abc'] [1, 2, 3, 4, 'abc']
```

Kopiowanie płytkie kolekcji

Dla sekwencji, jej segment będzie niezależną kopią wskazanych elementów.

```
>>> lista = [1,2,3,4,5]
>>> lista2 = lista[:]

>>> print(lista, lista2)
[1, 2, 3, 4, 5] [1, 2, 3, 4, 5]

>>> lista[4] = 'abc'
>>> print(lista, lista2)
[1, 2, 3, 4, 'abc'] [1, 2, 3, 4, 5]
```

Będzie to kopia płytka, tzn. kopia odniesień do obiektów będących elementami sekwencji.

Kopiowanie płytkie kolekcji

- Dla zbiorów i słowników mamy metodę `copy()` umożliwiającą tworzenie płytkiej kopii kolekcji.
- Możemy też używać nazwy typu jako funkcji z argumentem w postaci kolekcji przeznaczonej do skopiowania.

```
>>> lista = [1,2,3,4,5]
>>> kopia_listy = list(lista)

>>> zbiór = {1,2,3,4,5}
>>> kopia_zbioru = zbiór.copy()
>>> kopia_zbioru = set(zbiór)

>>> słownik = {1:'a', 2:'b', 3:'c'}
>>> kopia_słownika = słownik.copy()
>>> kopia_słownika = dict(słownik)
```

Kopiowanie głębokie kolekcji

Moduł copy z biblioteki standardowej umożliwia kopiowanie głębokie kolekcji.

```
>>> import copy

>>> x = [1,2,3,[1.0,2.0,['trzy', 'cztery', 'pięć']]]

>>> y = copy.copy(x)      # kopiowanie płykie

>>> z = copy.deepcopy(x) # kopiowanie głębokie
```

Polecenia sterujące i funkcje

Polecenie if

Składnia polecenia `if` jest następująca.

```
if wyrażenie_logiczne_1:
    blok_kodu_1
elif wyrażenie_logiczne_2:
    blok_kodu_2
    ....
elif wyrażenie_logiczne_N:
    blok_kodu_N
else:
    blok_else
```

W języku Python wyrażenie będzie fałszywe gdy:

- jawnie będzie równe `False`,
- jest obiektem `None`,
- jest pustą sekwencją bądź kolekcją (np. listą, krotką, tekstem),
- liczbowym typem danych równym 0.

Wyrażenie warunkowe z poleceniem if

`wyrażenie_1 if wyrażenie_logiczne else wyrażenie_2`

Jeśli `wyrażenie_logiczne` jest prawdziwe wynikiem będzie `wyrażenie_1`, w przeciwnym przypadku `wyrażenie_2`.

```
>>> zmienna = 1
>>> wynik = 20 + (5 if zmienna>0 else -5)
# 25

>>> zmienna = -1
>>> wynik = 20 + (5 if zmienna>0 else -5)
# 15

płeć = 'M'
s = 'Pan{0} X zaliczył{1} test'.format('' if płeć=='M' else 'i',
                                       '' if płeć=='M' else 'a')
# Pan X zaliczył test

płeć = 'K'
s = 'Pan{0} X zaliczył{1} test'.format('' if płeć=='M' else 'i',
                                       '' if płeć=='M' else 'a')
# Pani X zaliczyła test
```

Polecenie for i while

- Polecenie `for` jest używane w celu wykonania bloku kodu określoną ilość razy. Blok jest wykonywany dla każdej wartości występującej w sekwencji z nagłówka pętli.
- Polecenie `while` jest używane w celu wykonania bloku kodu wiele razy. Ilość powtórzeń zależy od stanu wyrażenia logicznego, znajdującego się w nagłówku pętli.

```
for zmienna in sekwencja:
    blok_kodu
else:
    blok_else
```

```
while wyrażenie_logiczne:
    blok_kodu
else:
    blok_else
```

W pętlach można używać poleceń `break` i `continue`.

Polecenie for i while

```
for zmienna in sekwencja:
    blok_kodu
else:
    blok_else
```

```
while wyrażenie_logiczne:
    blok_kodu
else:
    blok_else
```

- Klauzula `else` jest opcjonalna i jej blok kodu jest wykonywany, gdy pętla zakończy się w normalny sposób.
- Przerwanie pętli za pomocą `break`, `return` (gdy pętla jest w funkcji) lub po zgłoszeniu wyjątku skutkuje nie wykonaniem bloku kodu po `else`.

Polecenie for i while

```
def czy_są_cyfry_w_tekście(zdanie):
    '''
    Funkcja sprawdza czy w tekście są cyfry.
    Zwraca krotkę (True, pierwsza_cyfra) gdy są,
    lub (False, None) gdy nie ma.
    '''
    for z in zdanie:
        if z.isdigit():
            wynik = True
            break
    else:
        wynik = False
        z = None

    return wynik, z
```

Obsługa wyjątków

Składnia obsługi wyjątków jest następująca.

```
try:
    blok_kodu
except wyjątek_1 as zmienna_1:
    blok_kodu_1
...
except wyjątek_N as zmienna_N:
    blok_kodu_N
else:
    blok_else
finally:
    blok_finally
```

Użycie zmiennych jest opcjonalne. Zmienne przydają się przy wyświetlaniu informacji o zaistniałym wyjątku.

W konstrukcji musi się znajdować przynajmniej jeden blok `except`, natomiast bloki `else` i `finally` są opcjonalne.

Obsługa wyjątków

- Blok `else` jest wykonywany, gdy blok `try` zakończy działanie normalnie. **Nie jest wykonywany** gdy wystąpi wyjątek.
- Blok `finally` jest wykonywany **zawsze** na końcu.
- Wyjątki w klauzuli `except` mogą być pojedyncze lub w postaci krotki wyjątków w nawiasach.
- Dostęp do zmiennych jest możliwy w bloku obsługującym wyjątek.

Obsługa wyjątków

```
try:
    blok_kodu
except wyjątek_1 as zmienna_1:
    blok_kodu_1
...
except wyjątek_N as zmienna_N:
    blok_kodu_N
```

Jeśli wszystkie polecenia bloku `try` zostaną wykonane bez zgłoszenia wyjątku, bloki `except` będą pominięte.

Jeśli wyjątek wystąpi, program natychmiast przeskoczy do bloku kodu powiązanego z pierwszym z kolei dopasowanym typem wyjątku. Pozostałe polecenia w bloku `try` zostaną pominięte. Jeśli zdefiniowano zmienną, wówczas będzie ona odniesieniem do obiektu wyjątku.

Podstawy obsługi wyjątków

```
a = [1, 2, 3, 4, 5]
print(a[7])

-------------------------------------------------------------

>>>
Traceback (most recent call last):
  File "D:\Marcin\Kursy\Python\Examples\proby.py", line 2, in <module>
    print(a[7])
IndexError: list index out of range
>>>
```

Podstawy obsługi wyjątków

```
try:
    a = [1, 2, 3, 4, 5]
    for i in range(10):
        print(a[i] + 2)
except ValueError as zm_err_v:
    print('Nieprawidłowa wartość!')
    print(zm_err_v)
except IndexError as zm_err_i:
    print('Nieprawidłowy indeks!')
    print(zm_err_i)
except TypeError as zm_err_t:
    print('Nieprawidłowy typ danych!')
    print(zm_err_t)
```

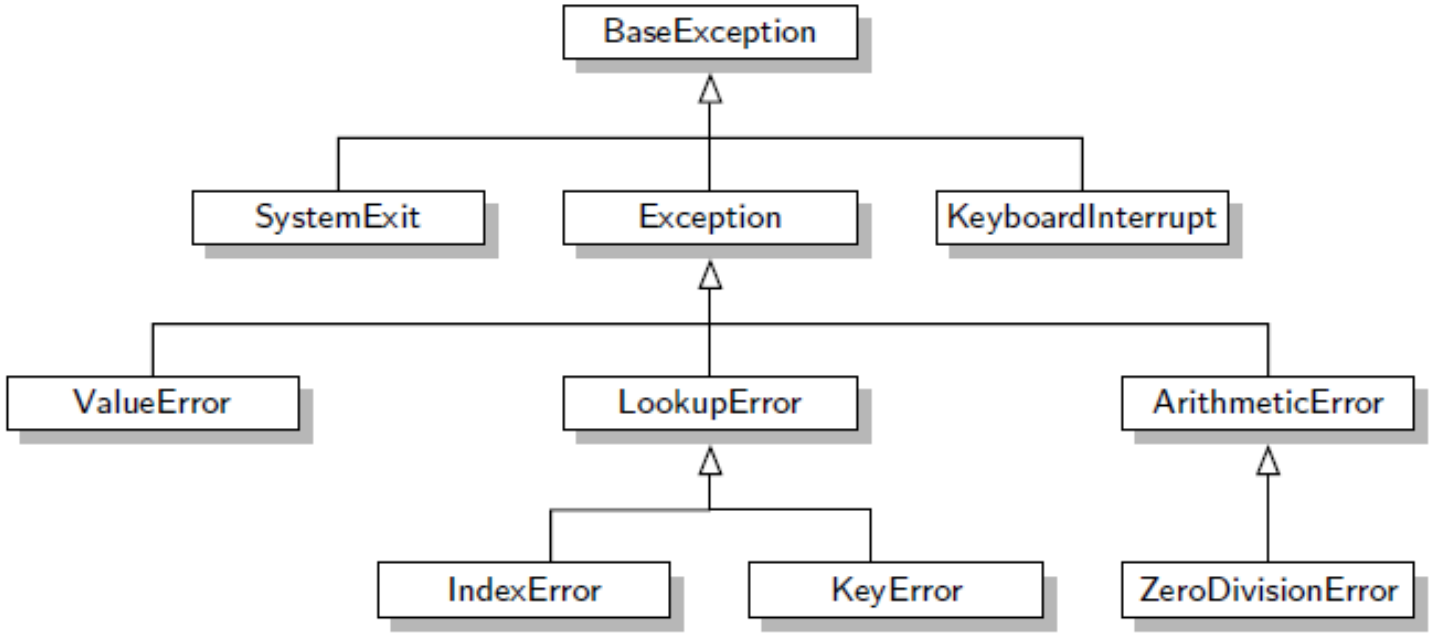
```
3
4
5
6
7
Nieprawidłowy indeks!
list index out of range
```

Podstawy obsługi wyjątków

```
try:
    a = [1, 'dwa', 3, 4, 5]
    for i in range(10):
        print(a[i] + 2)
except ValueError as zm_err_v:
    print('Nieprawidłowa wartość!')
    print(zm_err_v)
except IndexError as zm_err_i:
    print('Nieprawidłowy indeks!')
    print(zm_err_i)
except TypeError as zm_err_t:
    print('Nieprawidłowy typ danych!')
    print(zm_err_t)
```

```
3
Nieprawidłowy typ danych!
must be str, not int
```

Obsługa wyjątków




```
a = [1,2,3]
try:
    print(a[10])
except LookpupError:
    print('Błąd indeksowania')
except IndexError:
    print('Zły numer indeksu')
```

W przykładzie pokazana jest niewłaściwa kolejność bloków except.

Obsługa wyjątków

W przypadku wielu bloków except, należy stosować kolejność od najdokładniejszego do najbardziej ogólnego.

```
try:
    x = a[b]
except Exception:
    print('Coś się stało ?!')
except:
    print('Dziedziczenie po BaseException')
```

Używanie formy `except Exception:` jest złą praktyką, bo przechwyci wszystkie wyjątki.

Obsługa wyjątków

Hierarchia wyjątków w Pythonie.

```
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
      |    |    +-- 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
           +-- PendingDeprecationWar
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

Obsługa wyjątków

Gdy wyjątek nie znajdzie dopasowania, program będzie się poruszał w górę stosu wywołań i gdy nie znajdzie odpowiedniej procedury obsługi będzie przerwany, a w konsoli pojawi się komunikat o błędzie.

Wcześniej zostanie wykonany blok `finally`.

Możliwe jest także użycie:

```
try:
    blok_try
finally:
    blok_finally
```

Obsługa wyjątków

Częstym przykładem użycia poleceń `try except finally` jest obsługa błędów związanych z plikami.

```
def read_data(filename)
    lines = []
    fh = None
    try:
        fh = open(filename, encoding='utf8')
        for line in fh:
            if line.strip():
                lines.append(line)
    except (IOError, OSError) as err:
        print(err)
        return []
    finally:
        if fh is not None:
            fh.close()
    return lines
```

Zgłaszanie wyjątków

Wyjątki można zgłaszać za pomocą polecenia:

```
raise wyjątek(argumenty)
```

Jako argument można przekazać tekst, który będzie pokazany przy wyświetlaniu informacji o wyjątku.

Można tworzyć też własne wyjątki. Są one wtedy zwykle klasami dziedziczącymi po wyjątkach wbudowanych (bazowych).

Zgłaszanie wyjątków

```
def kwadrat_listy(s):
    lista = []
    for x in s:
        if isinstance(x,int) or isinstance(x,float):
            lista.append(x**2)
        else:
            raise TypeError('Element {0} nie jest liczbą'.format(x))
    return lista

try:
    a = []
    a = kwadrat_listy([1, 'dwa', 3])
except TypeError as err:
    print(err)

try:
    b = kwadrat_listy([5, 2.5, -1])
except TypeError as err:
    print(err)

print(a,b)
```

```
>>> Element dwa nie jest liczbą          [] [25, 6.25, 1]
```

Polecenia match i case

Od Pythona w wersji 3.10 dostępne są polecenia `match` i `case` (nie są słowami kluczowymi).

```
match wartość:
    case 5:
        print("Liczba dodatnia")
    case 0:
        print("Liczba zero")
    case -5:
        print("Liczba ujemna")
    case _:
        print("Nie znam takiej liczby!")
```

Można grupować kilka wartości używając operatora |.

```
    case 5 | 0:
        print("Liczba nieujemna")
```

Funkcje

W Pythonie mamy 4 rodzaje funkcji.

1. F. globalne – obiekty globalne są dostępne dla kodu znajdującego się w tym samym module (pliku *.py). Dostęp do obiektów globalnych jest możliwy także z innych modułów.
2. F. lokalne (zagnieżdżone) – funkcje zdefiniowane w innych funkcjach. Są dostępne tylko dla funkcji, w której są zdefiniowane. Zwykle są to małe funkcje pomocnicze.
3. F. lambda – są wyrażeniami tworzonymi w chwili użycia. Mają ograniczone możliwości.
4. Metody – funkcje zdefiniowane dla klasy.

Funkcja wbudowana `dir(klasa)` – wyświetla nazwy wszystkich metod zdefiniowanych dla klasy.

Funkcje

Ogólna składnia funkcji:

```
def nazwa_funkcji(argumenty):
    blok_kodu
```

- Argumenty są opcjonalne.
- Jeśli jest ich kilka, rozdzielamy je przecinkami.
- Każda funkcja zwraca wartość.
- Domyślnie zwracana jest wartość `None`.
- Można zwrócić inną wartość poleceniem: `return wartość`.
- Zwracana wartość może być pojedynczym elementem lub krotką elementów.
- Wartość zwrotna może być zignorowana w miejscu wywołania.
- Funkcje są obiektami, a polecenie `def` tworzy odniesienie do obiektu funkcji.

Funkcje

Parametry – sekwencja identyfikatorów lub par:
`identyfikator = wartość`

```
def pole_trapezu(a,b,h):                    # w przykładzie mamy argumenty
    if a < 0 or b < 0 or h < 0:             # pozycyjne
        return None                         #
    else:                                   # Jeśli ich liczba jest nieprawidłowa
        pole = 0.5*(a+b)*h                  # zgłaszany jest wyjątek
        return pole                         # TypeError

pole = pole_trapezu(5.5, 7, 4)
print('Pole trapezu =', pole)

pole = pole_trapezu(5.5, 7)

-------------------------------------------------

Pole trapezu = 25.0
Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 46, in <module>
    pole = pole_trapezu(5.5, 7)
TypeError: pole_trapezu() missing 1 required positional argument: 'h'
```

Funkcje

Dla parametrów funkcji można podać wartości domyślne:
`parametr = wartość_domyślna`

```
def skracaj(tekst, długość=20, zakończenie=' ...'):
    if len(tekst) > długość:
        tekst = tekst[:długość - len(zakończenie)] + zakończenie
    return tekst
```

- W definicji funkcji, parametrów z wartościami domyślnymi nie można umieszczać przed parametrami pozycyjnymi.
- Przy wywołaniu można przekazywać parametry w dowolnej kolejności używając formy `parametr = wartość` czyli tzw. słów kluczowych.
- Parametry, które mają w definicji wartość domyślną, można pomijać w wywołaniu. Parametry bez wartości domyślnej są obowiązkowe.
- Jeśli w wywołaniu funkcji część parametrów to wartości, a część przekazujemy z użyciem słów kluczowych – pierwsze muszą być zawsze przekazywane przez wartość.

Funkcje

```
def skracaj(tekst, długość=20, zakończenie=' ...'):
    if len(tekst) > długość:
        tekst = tekst[:długość - len(zakończenie)] + zakończenie
    return tekst

>>> skracaj('Przykładowy tekst do skrócenia')
'Przykładowy teks ...'

>>> skracaj(długość = 10, tekst='Przykładowy tekst do skrócenia')
'Przykł ...'

>>> skracaj('Przykładowy tekst do skrócenia', 15, ' xxx')
'Przykładowy xxx'

>>> skracaj('Przykładowy tekst do skrócenia', zakończenie=' <<<', długość=15)
'Przykładowy <<<'

>>> skracaj(długość = 10, 'Przykładowy tekst do skrócenia')
SyntaxError: positional argument follows keyword argument
```

Funkcje

Wiele funkcji z biblioteki standardowej ma argumenty pozycyjne (obowiązkowe) i kilka argumentów opcjonalnych, które możemy przekazywać z użyciem słów kluczowych.

Np. funkcja `open` – ma jeden obowiązkowy parametr pozycyjny (nazwa pliku) i sześć opcjonalnych, dzięki temu przy wywołaniu podajemy tylko te istotne w momencie użycia:

```
fh = open(filename, encoding='utf8')
```

Funkcje

Wartości domyślne (a dokładniej ich obiekty) są tworzone podczas definicji funkcji poleceniem `def`. Nie ma to znaczenia dla argumentów niemodyfikowalnych, jednak dla modyfikowalnych może powodować błędy.

```
# ŹLE:

def dodaj_dodatnie(x, lista=[])
    for element in x:
        if element > 0:
            lista.append(element)
    return lista
```

```
# DOBRZE:

def dodaj_dodatnie(x, lista=None)
    if lista is None:
        lista = []
    for element in x:
        if element > 0:
            lista.append(element)
    return lista
```

Funkcje

Należy przyjąć schemat nadawania nazw funkcji i konsekwentnie go stosować, np.:

- stałe – nazywać dużymi literami,
- nazwy klas i wyjątków – każde słowo zaczynać dużą literą,
- nazwy funkcji – małe litery.

Funkcje i metody powinny mieć nazwy informujące o tym co robią (a nie jak!) lub co zwracają. Należy też zwracać uwagę na odpowiednio informatywne nazwy argumentów.

Dokumentację do funkcji powinno się podawać w postaci **dokumentującego ciągu tekstowego** po nagłówku `def`, ale przed właściwym kodem funkcji. Przykłady w dokumentacji funkcji można wykorzystać do przeprowadzania testów jednostkowych.

Funkcje

```
def skracaj(tekst, długość=20, zakończenie=' ...'):
    '''
    Funkcja zwraca tekst srócony do wskazanej długości i z podanym zakończeniem

    tekst: dowolny tekst
    długość: oznacza maksymalną długość zwracanego wyniku razem z zakończeniem
    zakończenie: oznacza znaki wstawiane w miejscu skracania tekstu

    >>> skracaj('Przykładowy tekst do skrócenia')
    'Przykładowy teks ...'

    >>> skracaj(długość = 10, tekst='Przykładowy tekst do skrócenia')
    'Przykł ...'
    '''

    if len(tekst) > długość:
        tekst = tekst[:długość - len(zakończenie)] + zakończenie
    return tekst
```

Funkcje

Przy wywoływaniu funkcji można wykorzystywać operator rozpakowania sekwencji *

```
def pole_trapezu(a,b,h):
    return 0.5*(a+b)*h

>>> krotka = (3,4,5)
>>> pole_trapezu(*krotka)
17.5

>>> lista = [1,2,3]
>>> pole_trapezu(*lista)
4.5
```

Funkcje

Przy wywoływaniu funkcji można też wykorzystywać operator rozpakowania mapowania **

```
def skracaj(tekst, długość=20, zakończenie=' ...'):
    if len(tekst) > długość:
        tekst = tekst[:długość - len(zakończenie)] + zakończenie
    return tekst

>>> słownik = {'długość':15, 'zakończenie':' !!!'}

>>> skracaj('To jest zbyt długi tekst', **słownik)
'To jest zby !!!'
```

Funkcje

Operatorów rozpakowania można też używać na liście parametrów funkcji. Jest to użyteczne przy tworzeniu funkcji, które pobierają zmienną liczbę parametrów.

```
def średnia_geometryczna(*argumenty):
    n = len(argumenty)
    print(n, argumenty, type(argumenty))
    iloczyn = 1
    for x in argumenty:
        iloczyn *= x

    return iloczyn ** (1.0/n)

>>> średnia_geometryczna(1,2,3,4,5)
5 (1, 2, 3, 4, 5) <class 'tuple'>
2.605171084697352

>>> średnia_geometryczna(2,2,3)
3 (2, 2, 3) <class 'tuple'>
2.2894284851066637

>>> średnia_geometryczna(4,2,3,4,2,3,3,4,5,2,1,1)
12 (4, 2, 3, 4, 2, 3, 3, 4, 5, 2, 1, 1) <class 'tuple'>
2.5310475884616803
```

Funkcje

Po argumentach pozycyjnych można jeszcze przekazywać argumenty w postaci słów kluczowych.

```
def wykres_słupkowy(*argumenty, znak = '='):
    for x in argumenty:
        print(znak * x, ':', x)

>>> wykres_słupkowy(5,14,7,4)
===== : 5
============== : 14
======= : 7
==== : 4

>>> wykres_słupkowy(5,14,7,4,10,3,znak='*')
***** : 5
************** : 14
******* : 7
**** : 4
********** : 10
*** : 3
```

Funkcje

Przykładem funkcji działającej w ten sposób (akceptującej dowolną liczbę argumentów pozycyjnych) jest funkcja `print()`. Dodatkowo można do niej przekazać 3 argumenty w postaci słów kluczowych: `sep = ' '`, `end = '\n'`, `file = sys.stdout`

```
>>> print('a','b','c')
a b c
>>> print('a','b','c', sep = '')
abc
>>> print('a','b','c', sep = ' , ')
a , b , c

>>> print('a','b','c', sep = ' , ', end = ' '); print(1,2,3)
a , b , c 1 2 3

>>> print('a','b','c', sep = ' , ', end = ' -> '); print(1,2,3)
a , b , c -> 1 2 3
```

Funkcje

Operatora * można też użyć na liście parametrów funkcji do wskazania, że po nim nie będzie już argumentów pozycyjnych, choć możliwe jest użycie argumentów w postaci słów kluczowych.

```
def pole_trapezu(a, b, h, *, jednostki='cm^2', wynik='wartość'):
    pole = 0.5*(a+b)*h
    if wynik == 'wartość':
        return pole
    else:
        return '{0} [{1}]'.format(pole,jednostki)

>>> pole_trapezu(5.5, 7, 4)
25.0

>>> pole_trapezu(5.5, 7, 4, jednostki='m^2', wynik='tekst')
'25.0 [m^2]'

>>> pole_trapezu(5.5, 7, 4, 'm^2', 'tekst')
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    pole_trapezu(5.5, 7, 4, 'm^2', 'tekst')
TypeError: pole_trapezu() takes 3 positional arguments but 5 were given
```

Funkcje

Ustawiając * jako pierwszy parametr w definicji funkcji, możemy uniemożliwić jej wywoływanie z użyciem argumentów pozycyjnych.

```
def ustawienia(*, jednostki='m^2', wynik='tekst')
    blok_funkcji
```

Przykłady wywołania:

```
ustawienia()

ustawienia(wynik='wartość')

ustawienia('cm^2', 'wartość')  # błąd i zgłoszenie
                               # wyjątku TypeError
```

Funkcje

Operatora rozpakowania mapowania ** także można używać w definicji funkcji. Można w ten sposób tworzyć funkcje akceptujące dowolną liczbę argumentów w postaci słów kluczowych.

```
def dodaj_studenta(słownik, indeks, **kwargs):
    opis = ''
    for klucz in kwargs:
        opis += ' {0} = {1},'.format(klucz, kwargs[klucz])
    słownik[indeks] = opis
    return słownik

>>> słownik = dict()
>>> słownik = dodaj_studenta(słownik, 12345, imię='Jan', nazwisko='Nowak')

>>> słownik = dodaj_studenta(słownik, 23456, nazwisko='Kowalski', wiek=22,
                             średnia=5.0)
>>> print(słownik)
{12345: ' imię = Jan, nazwisko = Nowak,',
 23456: ' nazwisko = Kowalski, wiek = 22, średnia = 5.0,'}
>>>
```

Funkcje

Można też tworzyć funkcje przyjmujące zarówno zmienną liczbę argumentów pozycyjnych, jak i zmienną liczbę argumentów w formie słów kluczowych.

```
def pokaż_argumenty(*args, **kwargs):
    for i, argument in enumerate(args):
        print('Arg. pozycyjny nr {0} = {1}'.format(i,argument))
    for klucz in kwargs:
        print('Arg. w formie sł. kl. {0} = {1}'.format(klucz, kwargs[klucz]))

>>> pokaż_argumenty(1,2,'c','d', par1=5.3, par2='dwa', par3=0)
Arg. pozycyjny nr 0 = 1
Arg. pozycyjny nr 1 = 2
Arg. pozycyjny nr 2 = c
Arg. pozycyjny nr 3 = d
Arg. w formie sł. kl. par1 = 5.3
Arg. w formie sł. kl. par2 = dwa
Arg. w formie sł. kl. par3 = 0
```

Funkcja `enumerate()` zwraca iterator złożony z dwuelementowych krotek. Pierwszy element to nr iteracji, drugi to właściwa wartość.

Funkcje

Mamy możliwość definiowania zmiennych, które będą globalne także w funkcjach za pomocą polecenia:

```
global zmienna1,zmienna2
```

Nie jest to dobrą praktyką i zaleca się stosowanie tego typu rozwiązania w ograniczonym zakresie.

```
x = 3
y = 5
def funkcja():
    print(y)
    x = 0

print(x)
```

```
x = 3

def funkcja():
    global x
    x = 0

print(x)
```

Funkcje

```
x = 3
def funkcja():
    x = 0


print(x)    # 3
```

```
x = 3
def funkcja():
    global x
    x = 0


print(x)    # 0
```

Gdy Python napotyka zmienną `x`, szuka jej w zasięgu lokalnym, ale jej nie znajduje. Wówczas szuka zmiennej w zasięgu globalnym (plik *.py).

Polecenie `global` informuje, że zmienna istnieje w zasięgu globalnym (przypisanie do zmiennej globalnej, a nie tworzenie nowej zmiennej lokalnej).

Funkcje lambda

Funkcje `lambda` to funkcje utworzone za pomocą składni:

```
lambda parametry: wyrażenie
```

Część `parametry` zwykle składa się z argumentów pozycyjnych (choć można też używać innych technik stosowanych w definicjach funkcji globalnych).

Wyrażenie nie może zawierać rozgałęzień (dozwolone jest używanie wyrażeń warunkowych), pętli i polecenia `return`. Wynikiem jest funkcja anonimowa.

Funkcje lambda

```
>>> pole_trapezu = lambda a,b,h: 0.5*(a+b)*h
>>> pole_trapezu
<function <lambda> at 0x000001C8EEF45378>

>>> pole_trapezu(2,5,6)
21.0

>>> pole_trapezu = lambda a,b,h=5: 0.5*(a+b)*h

>>> pole_trapezu(3,2,2)
5.0
>>> pole_trapezu(3,2)
12.5
```

Funkcje lambda

```
>>> abs_max = lambda x,y: x if abs(x) >= abs(y) else y

>>> abs_max(1,-6)
-6
>>> abs_max(10,-6)
10
```

Funkcje lambda

```
>>> abs_max = lambda x,y: x if abs(x) >= abs(y) else y

>>> abs_max(1,-6)
-6
>>> abs_max(10,-6)
10
```

Funkcje `lambda` często stosowane są jako funkcje kluczowe przy sortowaniu.

```
>>> lista = [2,-4,5,-10,0]

>>> lista.sort(key = lambda x: x**2)
>>> lista
[0, 2, -4, 5, -10]
```

Moduły i pakiety

Moduły

Moduł to najprościej mówiąc plik `*.py`, w którym może się znajdować dowolny kod Pythona.

Pakiet to zbiór modułów pogrupowanych razem w jednym katalogu (zwykle z powodu wspólnego obszaru zastosowań lub występujących wzajemnych zależności między modułami).

- W modułach możemy definiować funkcje, zmienne, klasy i po zaimportowaniu używać ich w innych programach.
- Moduły zwykle przeznaczone są do importowania, jednak często mogą też stanowić odrębny, działający program.

Moduły importujemy poleceniem:

```
import element
import element_1, element_2, ..., element_N
import element as nowa_nazwa
```

Moduły

Element zwykle oznacza moduł, ale może być także pakietem, bądź modułem w pakiecie. W ostatnim przypadku nazwę pakietu i modułu rozdzielamy kropką.

```
>>> import math
>>> x = math.sin(math.pi/6)
>>> x
0.4999999999999994
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh

>>> import cmath as cm
>>> z = cm.sqrt(-1.0)
>>> z
1j
```

Moduły

`Element` zwykle oznacza moduł, ale może być także pakietem, bądź modułem w pakiecie. W ostatnim przypadku nazwę pakietu i modułu rozdzielamy kropką.

```
>>> import os.path
>>> os.path.getsize(r'c:\users\local\programs\python\python36\lib\ntpath.py')
23840
```

```
>>> import os.path as op
>>> op.getsize(r'c:\users\local\programs\python\python36\lib\ntpath.py')
23840
```

Importujemy w pierwszej kolejności moduły z biblioteki standardowej, moduły z bibliotek opracowanych przez firmy trzecie i dopiero na końcu własne.

Moduły

Możemy też importować tylko określone obiekty i odwoływać się do nich bezpośrednio (ryzyko konfliktów nazw!).

```
from element import obiekt
from element import obiekt as nowa_nazwa

from element import obiekt_1, obiekt_2, ..., obiekt_N
from element import (obiekt_1, obiekt_2, ...,
                     obiekt_N)

from element import *
```

W ostatnim przykładzie importowane będzie wszystko to, co nie jest prywatne (każdy obiekt w module, z wyjątkiem zaczynających się znakiem podkreślenia).

Jeśli moduł ma zdefiniowaną zmienną `__all__` przechowującą listę nazw, wówczas * importuje tylko obiekty z listy.

Moduły

```
>>> from math import sqrt
>>> x = sqrt(4.0)
>>> x
2.0
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'sqrt', 'x']

>>> from math import sin, cos, pi
>>> y = sin(pi/6)
>>> y
0.49999999999999994
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'cos', 'pi', 'sin', 'sqrt', 'x', 'y']
```

Moduły

```
# Plik: litery.py
####################################

__all__ = ['napisz_a', 'napisz_b']

def napisz_a():
    print('a')

def napisz_b():
    print('b')

def napisz_c():
    print('c')
```

```
>>> import litery
>>> dir()
['__builtins__', 'litery', 'sys']
>>> litery.napisz_c()
c
```

```
>>> from litery import *
>>> dir()
['__builtins__', 'napisz_a', 'napisz_b',
```

Moduły

Ze względu na ryzyko kolizji nazw składnia:

```
from element import *
```

nie jest zalecana.

Podczas importu Python wykorzystuje listę wbudowanego modułu `sys` o nazwie `sys.path`, przechowującą listę katalogów tworzących ścieżkę dostępu.

```
>>> import sys
>>> sys.path
['', 'C:\\Users\\M\\AppData\\Local\\Programs\\Python\\Python36',
'C:\\Users\\M\\AppData\\Local\\Programs\\Python\\Python36\\DLLs',
'C:\\Users\\M\\AppData\\Local\\Programs\\Python\\Python36\\Lib',
'C:\\Users\\M\\AppData\\Local\\Programs\\Python\\Python36\\Lib\\site-packages']
```

Moduły

- Pierwszy na liście jest katalog, w którym znajduje się uruchomiony program.
- Kolejny jest katalog wskazany w zmiennej środowiskowej PYTHONPATH.
- Na końcu listy są ścieżki prowadzące do biblioteki standardowej Pythona i dalej do bibliotek firm trzecich.

Tworząc własne moduły należy unikać nazw wykorzystanych już w bibliotekach. Najprostszym sposobem sprawdzenia czy dana nazwa jest już w użyciu, jest próba zaimportowania.

Moduły

- Moduły mogą także importować kolejne moduły.
- Podczas importu Python sprawdza, czy dany moduł jest już zaimportowany. Jeśli tak, nie podejmie żadnej akcji.
- Domyślnie, moduły są kompilowane (tworzony jest tzw. kod bajtowy) przy pierwszym poleceniu importu. Wynik kompilacji jest zapisywany w podkatalogu `__pycache__` w plikach z rozszerzeniem `*.pyc`.
- Używanie skompilowanych plików powoduje szybsze uruchamianie.
- Moduły biblioteki standardowej są kompilowane podczas instalacji.

Moduły

Każdy program i moduł ma zdefiniowaną zmienną __name__. Jeśli importujemy moduł, jego zmienna __name__ ma taką wartość jak nazwa pliku (bez rozszerzenia). Jeśli uruchamiamy moduł jako program, zmienna __name__ ma wartość '__main__'.
Zmienna __doc__ przechowuje dokumentujący ciąg tekstowy modułu.

```
# Plik: litery.py
####################################
'''Opis modułu litery'''

__all__ = ['napisz_a', 'napisz_b']

def napisz_a():
    print('a')

def napisz_b():
    print('b')

if __name__ == '__main__':
    print('Moduł uruchomiono jako program')
```

```
import litery

print(__name__)          # __main__
print(litery.__name__) # litery
print(litery.__doc__)  # Opis modułu lit
```

Pakiety

- Pakiet jest katalogiem zawierającym zestaw modułów oraz plik o nazwie `__init__.py`
- Plik `__init__.py` może być pusty.
- Pakiety można zagnieżdżać do dowolnego poziomu (każdy podkatalog musi zawierać plik `__init__.py`)
- Katalog z pakietem może być podkatalogiem katalogu zawierającego program lub może być w ścieżce dostępu Pythona.

Pakiety

```
Matematyka/
    __init__.py
    dodawanie.py
    odejmowanie.py
```

```
# dodawanie.py
#
def oblicz(x,y):
    return x + y
```

```
# odejmowanie.py
#
def oblicz(x,y):
    return x - y
```

```
import Matematyka.dodawanie as md
md.oblicz(1,2)
3
```

```
import Matematyka.odejmowanie
Matematyka.odejmowanie.oblicz(1,2)
-1
```

```
from Matematyka import dodawanie
dodawanie.oblicz(1,2)
3
```

```
from Matematyka.dodawanie import oblicz
oblicz(1,2)
3
```

Pakiety

Jeśli chcemy mieć możliwość importu wszystkich modułów poleceniem:

```
from Matematyka import *
dir()
['__builtins__', 'dodawanie', 'odejmowanie', 'sys']
dodawanie.oblicz(1,2)
3
```

musimy w pliku `__init__.py` zdefiniować zmienną `__all__` przechowującą listę nazw modułów:

```
__all__ = ['dodawanie', 'odejmowanie']
```

Dla pakietów istnieje zmienna `__path__` określająca ścieżkę dostępu do pakietu:

```
Matematyka.__path__
['D:\\Marcin\\Projekty\\Matematyka']
```

Biblioteka standardowa

Biblioteka standardowa dodawana jest podczas instalacji Pythona – w jej skład wchodzi ponad 200 pakietów i modułów.

Obsługa ciągów tekstowych

- `string` – dostarcza użyteczne stałe takie jak `string.ascii_letters`, `string.digits`, `string.hexdigits`, `string.whitespace` i inne. Udostępnia także bardziej zaawansowane techniki formatowania tekstu.
- `re` – obsługa wyrażeń regularnych.
- `difflib` – dostarcza klasy i funkcje służące do porównywania np. ciągów tekstowych i generowania raportów różnic.
- `textwrap` – wiele pomocniczych operacji na ciągach tekstowych.
- `unicodedata` – baza znaków Unicode
- `codecs` – funkcje pomagające przy różnych sposobach kodowania ciągów tekstowych.

Biblioteka standardowa

Złożone typy danych

- `datetime` – typy danych obsługujących czas i datę (obliczenia, formatowanie).
- `calendar` – pomocnicze funkcje związane z kalendarzem.
- `collections` – dodatkowe kolekcje danych, np. `namedtuple`, `OrderedDict`, `defaultdict` i inne.
- `array` – moduł udostępnia typ `array`, służący do przechowywania liczb bądź znaków w macierzach.
- `weakref` – słabe odniesienia do obiektów.
- `copy` – płytkie i głębokie kopiowanie obiektów.

Biblioteka standardowa

Moduły matematyczne

- `numbers` – numeryczne abstrakcyjne klasy bazowe i przydatne stałe
- `math` – podstawowe funkcje matematyczne
- `cmath` – podstawowe funkcje matematyczne dla liczb zespolonych
- `decimal` – udostępnia typ `Decimal` i jego funkcje matematyczne
- `fractions` – udostępnia typ `Fraction` – obliczenia na ułamkach i przydatne funkcje
- `random` – różne funkcje generujące liczby losowe
- `statistics` – udostępnia podstawowe funkcje umożliwiające analizę statystyczną

Biblioteka standardowa

Dostęp do plików i katalogów

- `os.path` – wiele funkcji manipulujących nazwami plików, katalogów, ścieżkami dostępu
- `shutil` – funkcje przeznaczone do kopiowania, przenoszenia, zmiany nazwy i usuwania plików
- `pathlib` – funkcje przetwarzające ścieżki dostępu do plików
- `filecmp` – funkcje do porównywania plików i katalogów
- `tempfile` – generowanie plików i katalogów tymczasowych
- `stat` – praca z atrybutami plików i katalogów
- `glob` – funkcje odnajdujące ścieżki i pliki, spełniające określone kryteria dopasowania

Biblioteka standardowa

Zapis i odczyt danych

- `pickle` – funkcje umożliwiające serializację obiektów (wygodny zapis i odczyt z dysku całych obiektów Pythona)
- `sqlite3` – interface umożliwiający komunikację z bazami danych SQLite
- `zlib`, `gzip`, `bz2`, `lzma`, `zipfile`, `tarfile` – funkcje umożliwiające kompresję i dekompresję danych zapisanych w różnych formatach
- `csv` – zapis i odczyt plików w formacie CSV
- `hashlib`, `hmac`, `secrets` – funkcje udostępniające różne usługi kryptograficzne

Biblioteka standardowa

Współpraca z systemem operacyjnym

- `os` – różnorodne funkcje współpracujące z systemem operacyjnym
- `io` – funkcje do pracy z plikami
- `time` – dostęp do czasu systemowego
- `platform` – funkcje udostępniające informacje o sprzęcie i systemie operacyjnym
- `threading`, `multiprocessing` – funkcje umożliwiające zrównoleglanie wykonywania programów

Biblioteka standardowa

Internet

- `email` – funkcje udostępniające obsługę e-maili
- `html`, `html.parser` – HyperText Markup Language – funkcje wspierające obsługę formatu
- `xml.dom`, `xml.sax` – pakiety wspierające obsługę formatu XML
- `webbrowser` – funkcje wspierające przeglądanie/odczyt stron Internetowych
- `urllib` – pakiet modułów obsługujących adresy URL
- `http` – pakiet modułów obsługujących protokół `htpp`
- `ftplib` – funkcje obsługujące protokół `ftp`
- `ipaddress` – funkcje manipulujące adresami IP

Biblioteka standardowa

Graphical User Interface

- `tkinter` – interface Pythona do biblioteki Tcl/Tk
- `tkinter.ttk`, `tkinter.tix` – widget'y do wykorzystania przy pracy z modułem `tkinter`

Biblioteki udostępniane przez firmy trzecie:

- PyQt – udostępnia funkcje wykorzystujące bibliotekę Qt.
- wxPython – biblioteka umożliwiająca tworzenie GUI dla różnych platform. Udostępnia dużo widgetów.

Biblioteka standardowa

Wspomaganie programowania

- `pydoc` – generator dokumentacji tworzonej na podstawie dokumentujących ciągów tekstowych
- `doctest` – testy wykorzystujące dokumentujące ciągi tekstowe
- `unittest` – moduł wspomagający przeprowadzanie testów jednostkowych
- `2to3` – wspomaganie automatycznego tłumaczenia kodu z wersji 2 na 3
- `timeit` – pomiar czasu wykonywania wskazanych fragmentów kodu
- `tracemalloc` – pomiar wykorzystania pamięci
- `sys` – parametry i funkcje systemu
- `gc` – funkcje Garbage Collector-a

Python Package Index

Python Package Index (PyPI) jest największym repozytorium pakietów i modułów opracowanych przez firmy trzecie dla Pythona. Udostępniony jest na stronie:

`https://pypi.org/`

- Do instalacji modułów i pakietów najwygodniej użyć programu: `pip` (package manager for Python packages).
- Od Pythona w wersji 3.4 jest on domyślnie zainstalowany.
- `pip` domyślnie instaluje pakiety ze strony powyżej, choć można też wskazać inne źródło.

pip – package manager

Programu używamy podając polecenie:

```
pip <command> [options]
```

Najważniejsze komendy:

- `install` – instalacja (lub upgrade) pakietu
- `uninstall` – odinstalowanie pakietu
- `list` – pokaż listę zainstalowanych pakietów i ich wersję
- `show` – pokaż informacje o zainstalowanym pakiecie
- `check` – sprawdzaj poprawność zależności między pakietami
- `help` – pokaż pomoc (pełna lista komend jest obszerniejsza)

pip – package manager

Zainstaluj NowyPakiet

```
pip install NowyPakiet
```

Zainstaluj NowyPakiet we wskazanej wersji

```
pip install NowyPakiet==1.7
```

Zainstaluj NowyPakiet w wersji z podanego przedziału

```
pip install NowyPakiet>=1,<2
```

Zainstaluj nową wersję pakietu NowyPakiet

```
pip install --upgrade NowyPakiet
```

pip – package manager

Zainstaluj NowyPakiet ze wskazanego archiwum:

```
pip install ./downloads/NowyPakiet-1.3.7.tar.gz
```

Pokaż listę zainstalowanych pakietów:

```
pip list
```

```
altgraph (0.16.1)
asn1crypto (0.24.0)
bcrypt (3.1.6)
bleach (2.1.2)
cffi (1.12.3)
colorama (0.3.9)
cryptography (2.6.1)
.........
```

pip – package manager

Pokaż informację o zainstalowanym pakiecie:

```
pip show matplotlib
```

```
Name: matplotlib
Version: 2.2.2
Summary: Python plotting package
Home-page: http://matplotlib.org
Author: John D. Hunter, Michael Droettboom
Author-email: matplotlib-users@python.org
License: BSD
Location: c:\users\marcin\appdata\local\programs\python\python36
Requires: pytz, six, python-dateutil, cycler, pyparsing, numpy,
```

Przykłady innych pakietów

- `numpy` – oferuje narzędzia do pracy z n-wymiarowymi tablicami (możliwość wygodnego i efektywnego indeksowania), podstawowe funkcje algebry liniowej i inne narzędzia matematyczne.
- `scipy` – rozszerza powyższy o wiele dodatkowych zadań matematycznych jak np. interpolacja, obliczanie całek, rozwiązywanie równań różniczkowych, przetwarzanie sygnału, obliczenia statystyczne i wiele innych.
- `matplotlib` – wspomaga kreślenie wykresów i wizualizację wyników obliczeń.
- `sympy` – wspomaga wykonywanie różnych obliczeń symbolicznych.
- `pandas` – definiuje użyteczne struktury danych wykorzystywane w analizie i uczeniu maszynowym.
- `scikit-learn` – zbiór funkcji do uczenia maszynowego i analizy danych.

Przykłady innych pakietów

- `docx` – odczyt i zapis dokumentów Worda, praca z testem, odczyt i zmiana atrybutów tekstu
- `openpyxl` – odczyt i zapis dokumentów Excela, dostęp do skoroszytów i komórek, odczyt i zmiana atrybutów komórek, praca z formułami i wykresami
- `pyPDF2` – odczyt i zapis plików w formacie PDF, edycja dokumentów, praca z tekstem dokumentu
- `pillow (PIL)` – praca z obrazami rastrowymi: odczyt i zapis obrazów, operacje edycyjne (np. przycinanie, zmiana wielkości, obrót, kreślenie figur, dodawanie tekstu)

Programowanie obiektowe

- W Pythonie można tworzyć własne klasy, które mogą być używane jak wbudowane typy danych.
- Klasy hermetyzują dane. W ich skład wchodzić mogą zmienne (atrybuty) i funkcje (metody), które można zastosować względem atrybutów.
- Wiele klas ma także metody specjalne, których nazwy zaczynają i kończą się dwoma znakami podkreślenia (np. `__add__()` lub `__len__()`). Metody specjalne umożliwiają zdefiniowanie zachowania obiektu przy użyciu operatorów (np. +) czy pewnych funkcji wbudowanych (np. `len()`).
- Nie należy definiować własnych metod zaczynających się dwoma znakami podkreślenia, o ile nie jest to jedna z metod specjalnych.
- Wewnątrz metod, możemy tworzyć zmienne lokalne i mamy też dostęp do zmiennych globalnych. Dostęp do atrybutów klasy wymaga podania pełnej nazwy, łącznie z nazwą klasy, np. `self.x`

Programowanie obiektowe

- Klasa może dziedziczyć po innych klasach (możliwe jest dziedziczenie wielokrotne), dzięki czemu możemy wykorzystać atrybuty i metody z klasy bazowej, dodać nowe lub zmienić te, które były (polimorfizm).
- W Pythonie, wszystkie klasy domyślnie dziedziczą po klasie bazowej `object`.
- W Pythonie nie jest możliwe przeciążanie metod (posiadanie metod o takich samych nazwach, ale różnych parametrach). Dzięki bardzo elastycznym możliwościom obsługi argumentów, zwykle nie jest to ograniczeniem.
- Nie ma też mechanizmów ograniczających dostęp do danych (prywatne, publiczne) jakie znamy z innych języków. W praktyce, nazwy zmiennych prywatnych będziemy zaczynali dwoma znakami podkreślenia. Mamy też możliwość stosowania tzw. dekoratorów metod.

Programowanie obiektowe

Klasy będziemy tworzyli poleceniem:

```
class NowaKlasa:
    blok_klasy

class NowaKlasa(KlasyBazowe):
    blok_klasy
```

Egzemplarze klasy będziemy tworzyć poprzez wywołanie jej nazwy z argumentami, np.:

```
z = complex(1,2)
```

Przykład – klasa Interval

```
import numbers

class Interval:

    def __init__(self, min_val = 0, max_val = 1):
        if  min_val > max_val:
            raise ValueError('Dolny brzeg interwału musi być mniejszy od górneg
        self.min_val = float(min_val)
        self.max_val = float(max_val)

    def __str__(self):
        return '[{0.min_val},{0.max_val}]'.format(self)

    def __repr__(self):
        return 'Interval({0.min_val},{0.max_val})'.format(self)

    def __eq__(self, other):
        return self.min_val == other.min_val and self.max_val == other.max_val

    def width(self):
        return self.max_val - self.min_val
```

Przykład – klasa Interval

- Klasa `Interval` ma 2 atrybuty: `min_val` i `max_val` oraz 5 metod (w tym 4 specjalne).
- Dostęp do atrybutów po zdefiniowaniu `a = Interval(1,10)`: `a.min_val`, `a.max_val`
- Zdefiniowano też konwersję na ciąg tekstowy i tzw. formę reprezentacyjną klasy.
- Klasa udostępnia obsługę operatora porównania == i automatycznie operatora nierówności `!=`
- W liście parametrów metod, pierwsze zawsze jest odniesienie do samego siebie: `self`
- Przy wywołaniu metod parametr ten nie jest przekazywany (dodawany jest automatycznie).
- Nazwy wszystkich atrybutów i metod (używane w definicjach metod klasy) muszą być pełne, np. `self.min_val`

Przykład – klasa Interval

```
>>> a = Interval(1,10)
>>> b = Interval()
>>> print(a, b, a.max_val)
[1.0,10.0] [0.0,1.0] 10.0

>>> repr(a)
Interval(1.0,10.0)
>>> str(a)
[1.0,10.0]

>>> a.width()
9.0

>>> print(a == b, a!=b)
False True

>>> c = Interval(10,2)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    c = Interval(10,2)
ValueError: Dolny brzeg interwału musi być mniejszy od górnego!
```

Przykład – klasa Interval

Klasa `Interval` dziedziczy domyślnie po obiekcie bazowym `object`.

```
object:          <---    Interval:
   __new__()                min_val
   __init__()               max_val
   __eq__()                 __init__()
   __repr__()               __str__()
   __str__()                __repr__()
   ......                   __eq__()
                            width()
```

- Przy tworzeniu obiektu, w pierwszej kolejności wywoływana jest metoda specjalna `__new__()`, tworząca obiekt. Ponieważ nie zdefiniowaliśmy jej w naszej klasie użyta będzie metoda z klasy bazowej.
- W drugiej kolejności wywołana jest klasa inicjalizująca `__init__()`
- Jeśli jakaś metoda nie istnieje, zgłaszany jest wyjątek `AttributeError`.

Przykład – klasa Interval

- Przy tworzeniu obiektu wywołana będzie metoda `__init__()` zdefiniowana ponownie w klasie `Interval`
- Jeśli z jakiegoś powodu chcemy wywołać metodę `__init__()` z klasy bazowej, musimy to zrobić za pomocą polecenia `super().__init__()`
- W przypadku klas dziedziczących tylko po klasie `object`, nie ma takiej potrzeby.

Przykład – klasa Interval

```
def __str__(self):
    return '[{0.min_val},{0.max_val}]'.format(self)
    # [1.0,10.0]

def __repr__(self):
    return 'Interval({0.min_val},{0.max_val})'.format(self)
    # Interval(1.0,10.0)
```

- Metoda specjalna `__repr__(self)` zwraca ciąg tekstowy, który wykorzystywany jest przy użyciu wbudowanej funkcji `repr(obiekt)` względem obiektu. Zwrócony ciąg tekstowy powinien mieć formę możliwą do interpretacji przez Pythona, w celu utworzenia odpowiednika obiektu.
- Metoda specjalna `__str__(self)` zwraca ciąg tekstowy czytelny dla człowieka. Jest ona wywoływana przy konwersji obiektu do postaci tekstu za pomocą funkcji `str(obiekt)`.

Przykład – klasa Interval

```
def __eq__(self, other):
    return self.min_val == other.min_val and self.max_val == other.max_val
```

- Metoda specjalna `__eq__(self,other)` zwraca `True` jeśli porównywane operatorem == obiekty są takie same. Jeśli nie zaimplementowano metody `__ne__(self,other)` automatycznie będzie działał też operator `!=`, na zasadzie zaprzeczenia metody `__eq__(self,other)`.
- W przykładzie założono, że porównywane obiekty są zawsze tego samego typu. Jeśli nie będą, zgłoszony będzie wyjątek `AttrributeError`.
- W praktyce, w metodach tego typu sprawdza się typ operandów (funkcja `isinstance()`) i odpowiednio programuje zachowanie metody.

Metody specjalne – porównania

- `__eq__(self,other)` – sposób użycia: `x == y`
- `__ne__(self,other)` – sposób użycia: `x != y`
- `__lt__(self,other)` – sposób użycia: `x < y`
- `__le__(self,other)` – sposób użycia: `x <= y`
- `__gt__(self,other)` – sposób użycia: `x > y`
- `__ge__(self,other)` – sposób użycia: `x >= y`

Metody specjalne – konwersja

- `__bool__(self)` – sposób użycia: `bool(x)` – konwersja do odpowiednika logicznego
- `__abs__(self)` – sposób użycia: `abs(x)`
- `__int__(self)` – sposób użycia: `int(x)` – konwersja do typu całkowitego
- `__float__(self)` – sposób użycia: `float(x)` – konwersja do typu rzeczywistego
- `__complex__(self)` – sposób użycia: `complex(x)` – konwersja do typu zespolonego
- `__round__(self,digits)` – sposób użycia: `round(x,digits)` – zaokrąglanie

Metody specjalne – operatory

```
__pos__(self)    +x
__neg__(self)    -x

__add__(self,other)    x + y
__iadd__(self,other)   x += y
__radd__(self,other)   y + x

__sub__(self,other)    x - y
__isub__(self,other)   x -= y
__rsub__(self,other)   y - x

__mul__(self,other)    x * y
__imul__(self,other)   x *= y
__rmul__(self,other)   y * x

__mod__(self,other)    x % y
__imod__(self,other)   x %= y
__rmod__(self,other)   y % x
```

```
__floordiv__(self,other)    x // y
__ifloordiv__(self,other)   x //= y
__rfloordiv__(self,other)   y // x

__truediv__(self,other)    x / y
__itruediv__(self,other)   x /= y
__rtruediv__(self,other)   y / x

__pow__(self,other)    x ** y
__ipow__(self,other)   x **= y
__rpow__(self,other)   y ** x

__and__(self,other)    x & y
__iand__(self,other)   x &= y
__rand__(self,other)   y & x

__or__(self,other)    x | y
__ior__(self,other)   x |= y
__ror__(self,other)   y | x
```

Metody specjalne – operatory

```
__xor__(self,other)    x ^ y
__ixor__(self,other)   x ^= y
__rxor__(self,other)   y ^ x

__lshift__(self,other)    x << y
__ilshift__(self,other)   x <<= y
__rlshift__(self,other)   y << x

__rshift__(self,other)    x >> y
__irshift__(self,other)   x >>= y
__rrshift__(self,other)   y >> x

__invert__(self)    ~x
```

Przykład – klasa Interval

```
def __neg__(self):
    return Interval(-self.max_val, -self.min_val)


def __add__(self, other):
    if  isinstance(other, Interval):
        return Interval(self.min_val + other.min_val, self.max_val + other.max_
    if  isinstance(other,numbers.Number):
        return Interval(self.min_val + other, self.max_val + other)
    raise NotImplemented()

def __radd__(self, other):
    if  isinstance(other,numbers.Number):
        return Interval(self.min_val + other, self.max_val + other)
    raise NotImplemented()
```

Dziedziczenie i polimorfizm

Python umożliwia dziedziczenie klas (także wielokrotne).

```
class NowaKlasa(KlasyBazowe):
    blok_klasy
```

- Klasa potomna dziedziczy zmienne i metody zdefiniowane w klasie bazowej.
- W klasie potomnej można dodawać nowe zmienne i metody. Można też definiować ponownie metody istniejące już w klasie bazowej (polimorfizm).
- Po ponownej definicji metody, jeśli występuje taka potrzeba, za pomocą funkcji `super()` można odwołać się do implementacji funkcji w klasie bazowej.

Dziedziczenie – przykład

```
class WektorSwobodny2D():
    def __init__(self, x=0, y=0):
        self.x = float(x)
        self.y = float(y)

    def __str__(self):
        return '[{0.x},{0.y}]'.format(self)

    def __repr__(self):
        return 'WektorSwobodny2D({0.x},{0.y})'.format(self)

    def __eq__(self,other):
        return self.x == other.x and self.y == other.y

    def długość(self):
        return (self.x*self.x + self.y*self.y)**0.5

##############################################################
ws1 = WektorSwobodny2D(3,-5)
ws2 = WektorSwobodny2D()
print(ws1, ws2, ws1.x, ws1.y)      # [3.0,-5.0] [0.0,0.0] 3.0 -5.0
print(ws1 == ws2)                  # False
print(ws1.długość())               # 5.830951894845301
```

Dziedziczenie – przykład

```
class WektorZaczepiony2D(WektorSwobodny2D):
    def __init__(self, x=0, y=0, x0=0, y0=0):
        super().__init__(x,y)  # lub: self.x = float(x); self.y = float(y)
        self.x0 = float(x0)
        self.y0 = float(y0)

    def __str__(self):
        return '({0.x0},{0.y0})[{0.x},{0.y}]'.format(self)

    def __repr__(self):
        return 'WektorZaczepiony2D({0.x},{0.y},0.x0},{0.y0})'.format(self)

    def __eq__(self,other):
        return self.x0 == other.x0 and self.y0 == other.y0
                                   and super().__eq__(other)

    def koniec(self):
        return self.x0+self.x, self.y0+self.y
```

Dziedziczenie – przykład

```
wz1 = WektorZaczepiony2D(4,-7)
wz2 = WektorZaczepiony2D(3,8,1,1)
wz3 = WektorZaczepiony2D()

print(wz1,wz2,wz3)
print(wz1.x, wz1.y, wz1.x0, wz1.y0)
print(wz2.koniec(),wz2.długość())
print(wz1 == wz2)

############################################################

(0.0,0.0)[4.0,-7.0] (1.0,1.0)[3.0,8.0] (0.0,0.0)[0.0,0.0]
4.0 -7.0 0.0 0.0
(4.0, 9.0) 8.54400374531753
False
```

Dziedziczenie – przykład

- Klasa `WektorSwobodny2D` dziedziczy domyślnie po obiekcie bazowym `object`.
- Klasa `WektorZaczepiony2D` dziedziczy po klasie `WektorSwobodny2D`.

```
object:          <---   WektorSwobodny2D:  <---   WektorZaczepiony2D
    __new__()               x                       x0
    __init__()              y                       y0
    __eq__()                __init__()              __init__()
    __repr__()              __str__()               __str__()
    __str__()               __repr__()              __repr__()
    ......                  __eq__()                __eq__()
                            długość()               koniec()
```

Atrybuty specjalne

Wszystkie obiekty mają atrybuty specjalne, automatycznie dostarczone przez Pythona:

- `__class__` – zawiera odniesienie do klasy obiektu,
- `__name__` – przechowuje nazwę klasy.

Formę reprezentacyjną można więc określić następująco (tylko raz w klasie bazowej):

```
class Kwadrat():
    def __init__(self, bok=0):
        self.podstawa = float(bok)

    def __repr__(self):
        return '{0}({1})'.format(self.__class__.__name__, self.podstawa)

##############################################################

a = Kwadrat();
print(repr(a))    # Kwadrat(1.0)
```

Własne klasy kolekcji

Python udostępnia metody specjalne pozwalające tworzyć własne kolekcje.

- Dzięki nim możemy łatwo samodzielnie zmienić działanie operatora dostępu [] i zaimplementować sposoby dodawania, usuwania i pobierania elementów z kolekcji.
- Elementy kolekcji przechowywane są zwykle w zmiennej prywatnej, do której bezpośredni dostęp mają tylko metody specjalne kolekcji.

Metody specjalne kolekcji

- `__contains__(self, x)` – użycie: `x in a`
  Zwraca wartość `True` jeśli `x` występuje w kolekcji `a`.
- `__delitem__(self, k)` – użycie: `del a[k]`
  Usuwa element o indeksie `k` z kolekcji `a`.
- `__getitem__(self, k)` – użycie: `a[k]`
  Zwraca element o indeksie `k` z kolekcji `a`.
- `__setitem__(self, k, v)` – użycie: `a[k] = v`
  Przypisuje elementowi o indeksie `k` w kolekcji `a` wartość `v`.
- `__len__(self)` – użycie: `len(a)`
  Zwraca rozmiar kolekcji `a`.
- `__iter__(self)` – użycie: `for element in a`
  Zwraca iterator dla kolekcji `a`.
- `__reversed__(self)` – użycie: `reversed(a)`
  Zwraca iterator z elementami w odwrotnej kolejności dla kolekcji `a`.

Własna kolekcja – przykład

Klasa obsługująca trójwymiarową macierz rzadką.

```
class SparseMartrix3D():
    def __init__(self, rows=1, columns=1, pages=1):
        if  rows<1 or columns<1 or pages<1:
            raise AttributeError('Rozmiary mac. nie mogą być mniejsze od 1!')
        self.rows = rows
        self.columns = columns
        self.pages = pages
        self.__data = [0 for i in range(self.rows*self.columns*self.pages)]

    def __str__(self):
        return 'SparseMatrix3D: {0.rows} x {0.columns} x {0.pages}'.format(self

    def __repr__(self):
        return 'SparseMatrix3D({0.rows},{0.columns},{0.pages})'.format(self)

    def __eq__(self,other):
        if  self.rows != other.rows or self.columns != other.columns or self.pa
            return False
        else:
            return True
```

Własna kolekcja – przykład

```
>>> a = SparseMartrix3D(3,4,2)
>>> b = SparseMartrix3D()

>>> print(a, b, sep=', ')
SparseMatrix3D: 3 x 4 x 2, SparseMatrix3D: 1 x 1 x 1



>>> print(repr(a), repr(b), sep=', ')
SparseMartrix3D(3,4,2), SparseMartrix3D(1,1,1)

>>> print(a == b)
False
```

Własna kolekcja – przykład

Metody umożliwiające użycie operatora dostępu [] i potrójne indeksowanie elementów.

```
    def __getitem__(self, item):
        r ,c, p = item
        return self.__data[p*(self.rows*self.columns) + c*self.rows + r]

    def __setitem__(self, item, v):
        r, c, p = item
        self.__data[p*(self.rows*self.columns) + c*self.rows + r] = v


##########################################################################

>>> a = SparseMartrix3D(3,4,2)
>>> a[1,3,1] = 4
>>> a[2,2,0] = 7
>>> a[0,3,1] = 5

>>> print(a[1,3,1])
4
```

Własna kolekcja – przykład

Metoda umożliwiająca użycie operatora `in`.

```
    def __contains__(self,x):
        if x in self.__data:
            return True
        else:
            return False
```

```
##########################################################################

>>> a = SparseMartrix3D(3,4,2)
>>> a[1,3,1] = 4
>>> a[2,2,0] = 7
>>> a[0,3,1] = 5

>>> print(3 in a, 4 in a)
False True
```

Własna kolekcja – przykład

Metoda umożliwiająca użycie względem kolekcji wbudowanej funkcji `len()`.

```
    def __len__(self):
        '''
        Zwraca ilość niezerowych elementów
        '''
        sum = 0
        for i in self.__data: sum += bool(i)
        return sum

##########################################################################

>>> a = SparseMartrix3D(3,4,2)
>>> a[1,3,1] = 4
>>> a[2,2,0] = 7
>>> a[0,3,1] = 5

>>> print(len(a))
3
```

Własna kolekcja – przykład

Metoda umożliwiająca użycie względem kolekcji polecenia `del`.

```
    def __delitem__(self, item):
        '''
        Zeruje wskazany element
        '''
        r, c, p = item
        self.__data[p * (self.rows * self.columns) + c * self.rows + r] = 0
```

```
##########################################################################

>>> a = SparseMartrix3D(3,4,2)
>>> a[1,3,1] = 4
>>> a[2,2,0] = 7
>>> a[0,3,1] = 5

>>> del a[0,3,1]
>>> print(a[0,3,1])
0
```

Własna kolekcja – przykład

Metody umożliwiające iterację kolekcji w pętli `for`.

```
    def __iter__(self):
        return iter(self.__data)

    def __reversed__(self):
        return  iter(reversed(self.__data))


#########################################################################

>>> a = SparseMartrix3D(3,4,2)
>>> a[1,3,1] = 4
>>> a[2,2,0] = 7
>>> a[0,3,1] = 5

>>> for i in a: print(i,end='')
>>> print('')
000000007000000000000540

>>> for i in reversed(a): print(i,end='')
>>> print('')
045000000000000700000000
```

Własna kolekcja – przykład

Pomocnicza metoda, umożliwiająca wyświetlanie macierzy stronami.

```
    def show_matrix(self):
        for i in range(self.pages):
            print('Strona ',i,':',sep='')
            for j in range(self.rows):
                for k in range(self.columns):
                    print(self[j,k,i],', ',end='')
                print('')

##########################################################################
```

```
>>> a = SparseMartrix3D(3,4,2)
>>> a[1,3,1] = 4
>>> a[2,2,0] = 7
>>> a[0,3,1] = 5
```

```
>>> a.show_matrix()
Strona 0:
0 , 0 , 0 , 0 ,
0 , 0 , 0 , 0 ,
0 , 0 , 7 , 0 ,
Strona 1:
0 , 0 , 0 , 5 ,
0 , 0 , 0 , 4 ,
0 , 0 , 0 , 0 ,
```

Własne kolekcje – kopiowanie

W samodzielnie tworzonych kolekcjach można też na nowo zdefiniować funkcje:

- `__copy__()` – umożliwiającą płytkie kopiowanie kolekcji,
- `__deepcopy__()` – umożliwiającą głębokie kopiowanie kolekcji.

Po zaimportowaniu modułu `copy`, funkcja `y = copy.copy(x)` próbuje zawsze użyć metody specjalnej `__copy__()` zdefiniowanej dla kolekcji. Jeśli jej nie znajdzie, użyje własnego kodu.

Podobnie działa funkcja `y = copy.deepcopy(x)`.

Obsługa plików

Typy danych: bytes i bytearray

Python udostępnia dwa typy danych do obsługi ciągów bajtów:

- `bytes` – niemodyfikowalny,
- `bytearray` – modyfikowalny.

Oba przechowują sekwencję zera lub większej liczby ośmiobitowych liczb całkowitych baz znaku (bajtów) z zakresu od 0 do 255.

Oba typy są bardzo podobne do ciągów tekstowych i większość metod istniejących dla tekstów, działa także w ich przypadku.

Typ `bytearray` oferuje także metody istniejące dla list.

Typy danych: bytes i bytearray

```
>>> c = b'Przykład'
SyntaxError: bytes can only contain ASCII literal characters.

>>> c = b'Przyklad'
>>> c
b'Przyklad'

>>> str(c)
"b'Przyklad'"

>>> str(c, encoding='utf8')
'Przyklad'

>>> c = bytes('Przykład')
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    c = bytes('Przykład')
TypeError: string argument without an encoding

>>> c = bytes('Przykład', encoding='utf8')
>>> c
b'Przyk\xc5\x82ad'
>>> str(c, encoding='utf8')    # wynikiem jest: 'Przykład'
```

Typy danych: bytes i bytearray

```
>>> a = bytearray('Przykład', encoding='utf8')
>>> a
bytearray(b'Przyk\xc5\x82ad')

>>> b = a.upper()
>>> b
bytearray(b'PRZYK\xc5\x82AD')

>>> str(b, encoding='utf8')
'PRZYKŁAD'

>>> a.append('y')
Traceback (most recent call last):
  File "<pyshell#41>", line 1, in <module>
    a.append('y')
TypeError: an integer is required

>>> a.append(ord('y'))
>>> a
bytearray(b'Przyk\xc5\x82ady')
```

Typy danych: bytes i bytearray

```
>>> a = bytearray('Przykłady', encoding='utf8')
>>> a.extend(' tekstu')
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    a.extend(' tekstu')
TypeError: an integer is required

>>> a.extend(bytearray(' tekstu', encoding='utf8'))
>>> a
bytearray(b'Przyk\xc5\x82ady tekstu')

>>> x = b'To jest '
>>> y = b'typ bytes'
>>> x + y
b'To jest typ bytes'

>>> a = bytes.fromhex('31 32 33 34') # ignoruje spacje!
>>> a
b'1234'

>>> a = b'\x31 \x32 \x33 \x34'
>>> a
b'1 2 3 4'
```

Typy danych: bytes i bytearray

Typy danych `bytes` i `bytearray` indeksujemy podobnie jak ciągi tekstowe i listy.

- Segment zwraca obiekt tego samego typu.
- Dostęp do pojedynczego elementu ([]) zwraca typ `int`.

```
>>> a = b'To jest tekst'
>>> a
b'To jest tekst'
>>> len(a)
13

>>> b = a[::-1]
>>> b
b'tsket tsej oT'

>>> c = a[1:10:2]
>>> c
b'ojs e'
```

```
>>> d = a[:7]
>>> d
b'To jest'

>>> e = a[3]
>>> e
106
```

Opracje na plikach i katalogach

Jednym z podstawowych zadań systemu operacyjnego jest obsługa dyskowego systemu plików. Moduł standardowy os udostępnia wiele funkcji służących do obsługi katalogów oraz operacji na plikach (kopiowanie, przenoszenie, usuwanie, itp.).

Funkcja `os.getcwd()` pokazuje w jakim katalogu dyskowym pracujemy aktualnie.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\Marcin\\AppData\\Local\\Programs\\Python\\Python36'
```

Funkcja `os.chdir('nowy katalog')` pozwala na zmianę bieżącego katalogu na inny.

```
>>> os.chdir(r'C:\Marcin\Dokumenty\Dydaktyka\Kursy\Python\Prezentacja')
>>> os.getcwd()
'C:\\Marcin\\Dokumenty\\Dydaktyka\\Kursy\\Python\\Prezentacja'
```

Opracje na plikach i katalogach

```
>>> os.chdir(r'C:\Marcin\Dokumenty\Dydaktyka\Kursy\Python\Prezentacja')
>>> os.getcwd()
'C:\\Marcin\\Dokumenty\\Dydaktyka\\Kursy\\Python\\Prezentacja'

os.chdir(r'..')
>>> os.getcwd()
'C:\\Marcin\\Dokumenty\\Dydaktyka\\Kursy\\Python'

>>> os.chdir(r'..\..')
>>> os.getcwd()
'C:\\Marcin\\Dokumenty\\Dydaktyka'
```

Opracje na plikach i katalogach

Zawartość określonego katalogu pokazuje funkcja `os.listdir('katalog')`.

```
>>> import os
>>> os.listdir()
['DLLs', 'Doc', 'etc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'py

>>> os.listdir('.')
['DLLs', 'Doc', 'etc', 'include', 'Lib', 'libs', 'LICENSE.txt', 'NEWS.txt', 'py

>>> os.listdir(r'libs')
['libpython36.a', 'python3.lib', 'python36.lib', '_tkinter.lib']

>>> os.listdir(r'c:\Marcin')
['Dokumenty', 'Inst', 'Instrukcje', 'Programy', 'Projekty', 'Stare', 'Wykaz_cza
```

Opracje na plikach i katalogach

Funkcja `os.mkdir('katalog')` tworzy na dysku nowy katalog.

```
>>> os.chdir(r'C:\Marcin\Dokumenty\Dydaktyka\Kursy\Python')
>>> os.listdir()
['Cwiczenia', 'Examples', 'Power', 'Prezentacja', 'Python_cw.txss', 'Sylabusy']

>>> os.mkdir('Nowy')
>>> os.listdir()
['Cwiczenia', 'Examples', 'Nowy', 'Power', 'Prezentacja', 'Python_cw.txss', 'Sy

>>> import os.path
>>> os.path.exists('Nowy')
True
```

Funkcja `os.path.exists('obiekt')` sprawdza, czy dany obiekt dyskowy (plik lub katalog) istnieje.

Opracje na plikach i katalogach

Funkcja `os.rename('stara_nazwa','nowa_nazwa')` zmienia nazwę pliku lub katalogu.

```
>>> os.chdir(r'C:\Marcin\Dokumenty\Dydaktyka\Kursy\Python')
>>> os.listdir()
['Cwiczenia', 'Examples', 'Power', 'Prezentacja', 'Python_cw.txss', 'Sylabusy']

>>> os.rename('Power','Power_1')
>>> os.listdir()
['Cwiczenia', 'Examples', 'Power_1', 'Prezentacja', 'Python_cw.txss', 'Sylabusy
```

Funkcja może także służyć do przenoszenia obiektów pomiędzy katalogami.

```
>>> os.rename('Python_cw.txss',r'Power\Python_cw.txss')
>>> os.listdir()
['Cwiczenia', 'Examples', 'Power', 'Prezentacja', 'Sylabusy']
>>> os.chdir('Power')
>>> os.listdir()
['Karta_2019.docx', 'Python_cw.txss']
```

Opracje na plikach i katalogach

Funkcja `os.path.isfile('plik')` sprawdza, czy dany obiekt dyskowy jest plikiem.

```
>>> os.path.isfile('Python_cw.txss')
True
>>> os.path.isfile('Power')
False
```

Funkcja `os.path.isdir('katalog')` sprawdza, czy dany obiekt dyskowy jest katalogiem.

```
>>> os.path.isdir('Power')
True
>>> os.path.isdir('Python_cw.txss')
False
```

Funkcja `os.path.ismount('obiekt')` sprawdza, czy dany obiekt dyskowy jest dyskiem.

```
>>> os.path.ismount('Power')
False
>>> os.path.ismount('C:\\')
True
```

Opracje na plikach i katalogach

Funkcja `os.path.getsize('plik')` zwraca długość pliku w bajtach.

```
>>> os.path.getsize('Python_cw.txss')
1548

>>> for x in os.listdir('.'):
        print(x, os.path.getsize(x))

Cwiczenia 4096
Examples 4096
Power 0
Prezentacja 8192
Python_cw.txss 1548
Sylabusy 0

>>> os.chdir('Prezentacja')
>>> sum([os.path.getsize(f) for f in os.listdir('.') if os.path.isfile(f)])
459752
>>> len([os.path.getsize(f) for f in os.listdir('.') if os.path.isfile(f)])
18
```

Opracje na plikach i katalogach

Funkcja `os.path.getctime('plik')` zwraca czas stworzenia, a `os.path.getmtime('plik')` czas ostatniej modyfikacji obiektu. Czas zwracany jest w sekundach od momentu zależnego od systemu (poniżej 01.01.1970 r.).

```
>>> import time
>>> time.gmtime(0)
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec

>>> os.path.getctime('Python_cw.txss')
1576156069.937526

>>> tc = os.path.getctime('Python_cw.txss')
>>> time.ctime(tc)
'Thu Dec 12 14:07:49 2019'

>>> tm = os.path.getmtime('Python_cw.txss')
>>> time.ctime(tm)
'Thu Dec 12 16:22:02 2019'
```

Opracje na plikach i katalogach

Funkcja `os.path.join()` łączy ciąg katalogów w ścieżkę, a `os.path.isabs('sciezka')` sprawdza, czy podana ścieżka jest absolutna.

```
>>> s = os.path.join('Marcin', 'Dokumenty', 'Dydaktyka')
>>> s
'Marcin\\Dokumenty\\Dydaktyka'

>>> os.path.isabs(s)
False
```

Funkcja `os.remove('plik')` usuwa z dysku plik, a `os.rmdir('katalog')` usuwa z dysku katalog.

```
>>> os.rmdir('Power1')
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    os.rmdir('Power1')
OSError: [WinError 145] Katalog nie jest pusty: 'Power1'

>>> os.rmdir('Power1')
>>> os.remove('proba.py')
```

Opracje na plikach i katalogach

Funkcja `os.walk('katalog', kolejność)` służy do rekursywnego przechodzenia podanego katalogu wraz ze wszystkimi jego podkatalogami.

Dla każdego znalezionego katalogu (łącznie ze wskazanym) zwraca trójkę: (ścieżka, podkatalogi, pliki), gdzie:

- ścieżka – oznacza ścieżkę dostępu do katalogu,
- podkatalogi – listę nazw zawartych w nim podkatalogów,
- pliki – listę nazw zawartych w nim plików.

Parametr kolejność (domyślnie `True`) ustawiony na `False` zwraca katalogi w odwrotnym porządku (zaczyna od najgłębiej położonego) Może być to przydatne, jeżeli zamierzamy np. kasować podkatalogi.

Opracje na plikach i katalogach

```
import os
os.chdir(r'C:\Marcin\Kursy\Python')

for sciezka, podkatalogi, pliki in os.walk(r'C:\Marcin\Kursy\Python'):
    print(sciezka)
    print(podkatalogi)
    print(pliki, '\n')

##########################################################################

C:\Marcin\Kursy\Python
['Cwiczenia', 'Examples', 'Power', 'Prezentacja', 'Sylabusy']
['Python_cw.txss']

C:\Marcin\Kursy\Python\Cwiczenia
['Zadania']
['mwbk_mp.cls', 'Python_cw.aux', 'Python_cw.idx', 'Python_cw.log', 'Python_cw.p

C:\Marcin\Kursy\Python\Cwiczenia\Zadania
[]
['Język Python - Laboratorium 1.pdf', 'Język Python - Laboratorium 10.pdf', 'Ję

......
```

Opracje na plikach i katalogach

```
import os

for sciezka, podkatalogi, pliki in os.walk(r'C:\Marcin\Kursy\Python'):
    print('W katalogu {0} jest {1} bajtów w {2} plikach'.format(sciezka,
        sum([os.path.getsize(os.path.join(sciezka,nazwa)) for nazwa in pliki]),
        len(pliki)))

#######################################################################

W katalogu C:\Marcin\Kursy\Python jest 1548 bajtów w 1 plikach
W katalogu C:\Marcin\Kursy\Python\Cwiczenia jest 165374 bajtów w 12 plikach
W katalogu C:\Marcin\Kursy\Python\Cwiczenia\Zadania jest 4031605 bajtów w 20 pl
W katalogu C:\Marcin\Kursy\Python\Examples jest 2059 bajtów w 6 plikach
W katalogu C:\Marcin\Kursy\Python\Examples\Matematyka jest 103 bajtów w 3 plika
W katalogu C:\Marcin\Kursy\Python\Power jest 47448 bajtów w 1 plikach
W katalogu C:\Marcin\Kursy\Python\Prezentacja jest 523904 bajtów w 18 plikach
W katalogu C:\Marcin\Kursy\Python\Prezentacja\Rys jest 29050 bajtów w 2 plikach
W katalogu C:\Marcin\Kursy\Python\Sylabusy jest 277657 bajtów w 2 plikach
```

Zapis i odczyt plików

```
fh = open('plik.txt','w')
```

Funkcja spowoduje utworzenie i otwarcie do zapisu danych tekstowych plik "plik.txt" w aktualnym katalogu dyskowym.

Obiekty plikowe mają kilka atrybutów:

- `fh.name` – zawiera nazwę pliku,
- `fh.mode` – określa tryb, w jakim otwarto plik,
- `fh.closed` – określa czy plik jest zamknięty (wartość logiczna),
- `fh.encoding` – określa sposób kodowania tekstu.

Zapis i odczyt plików

Zapis tekstu do pliku:

```
fh.write('Pierwsza linia\n')
fh.write('Druga linia')
```

Funkcja `write` nie kończy zapisanych danych znakiem końca linii, stąd konieczność wstawienia `\n`.

Zapis danych z bufora do pliku i zamknięcie pliku:

```
fh.close()
```

Zapis danych z bufora do pliku (bez zamykania):

```
fh.flush()
```

Normalne zakończenie programu powoduje automatyczne zamknięcie wszystkich otwartych plików, jednak dobrą praktyką jest zamykanie otworzonych plików.

Zapis i odczyt plików

Operacje zapisu i odczytu danych należy przeprowadzać z obsługą wyjątków (uwaga na zmiany w kolejnych wersjach Pythona!).

```
import os
import sys
os.chdir(r'C:\Marcin\Dokumenty\Dydaktyka\Kursy\Python')

fh = None
try:
    fh = open('plik.txt','w',encoding='utf8')
    fh.write('Pierwsza linia pliku\n')
    fh.write('Druga linia pliku')
except OSError as err:
    print('{0}: błąd zapisu: {1}'.format(os.path.basename(sys.argv[0]),err))
finally:
    if fh is not None:
        fh.close()

# plik.txt #
Pierwsza linia pliku
Druga linia pliku
```

Zapis i odczyt plików

Przykład otwarcia pliku do odczytu i wczytanie jego zawartości (funkcja `read()`).

```
fh = open('plik.txt','r')
print(fh.encoding)
tekst  = fh.read()
print(tekst)
fh.close

##################################
cp1250
Pierwsza linia pliku
Druga linia pliku
```

Należy zwracać uwagę na sposób kodowania tekstu. W powyższym przykładzie, przy odczycie przyjęte zostało domyślnie kodowanie cp1250. Poprawniej byłoby:

```
fh = open('plik.txt','r',encoding='utf8')
```

Zapis i odczyt plików

- `fh.read(długość)` – wczytywanie z pliku tylko fragmentu o określonej długości,
- `fh.tell()` – podaje aktualną pozycję odczytu w pliku,
- `fh.seek(pozycja)` – ustawia pozycję odczytu w pliku na podaną, np: `fh.seek(0)`.

Zapis i odczyt plików – przykład

Przykład kopiowania plików (dla uproszczenia – bez obsługi wyjątków).

```
fh_oryg = open('plik.txt', 'rb')          # otwieramy oryginał do odczytu
fh_kop  = open('plik_kopia.txt', 'wb')    # otwieramy kopię do zapisu
                                          # binarnego
while True:
      b = fh_oryg.read(1)          # wczytujemy 1 bajt z oryginału
      if not b: break              # nic się nie wczytało? Koniec pliku!
      fh_kop.write(b)              # zapisujemy 1 bajt do kopii

fh_kop.close()                     # zamykamy kopię
fh_oryg.close()                    # zamykamy oryginał
print('Kopiowanie zakończone pomyślnie')
```

Zapis i odczyt plików – przykład

Przykład szyfrowania pliku tekstowego z zastosowaniem tzw. szyfru Cezara czyli przesuwania kodów liter o podaną wartość (dla uproszczenia – bez obsługi wyjątków).

```
p = -5                                  # przesunięcie
# do odszyfrowania można podać liczbę przeciwną

fh = open('plik_kopia.txt', 'r')        # plik otwarty do odczytu tekstu
t = fh.read()                           # wczytujemy tekst
fh.close()

ts = ''                                 # sz oznacza tekst zaszyfrowany

fhs = open('plik_szyfr.txt', 'w')
for c in t:                             # dla każdego znaku w t
    a=ord(c)                            # wyliczamy kod ASCII
    if 48 < a < 122:                    # szyfrujemy tylko litery i cyfry
        c = chr((a+p) % 256)            # innych nie zmieniamy
    ts += c                             # dodajemy znak

fhs.write(ts)               # zapisujmy sz
fhs.close()                 # zamykamy plik
```

```
Pierwsza linia mojego pliku
Druga linia pliku
```

```
Kd‘mrnz\ gdid\ hje‘bj kgdfp
?mpb\ gdid\ kgdfp
```

Zapis i odczyt plików – parametry

Parametr `mode` jest ciągiem tekstowym określającym tryb w jakim otworzony jest plik.

- `r` – domyślny, oznaczający plik do odczytu w trybie tekstowym – to samo co `rt`),
- `w` – zapis pliku w trybie tekstowym (jeśli plik istnieje, będzie nadpisany) – to samo co `wt`),
- `x` – otwieranie pliku w trybie 'exclusive creation' – jeśli plik już istnieje, nie udaje się,
- `a` – otwieranie do zapisu, w trybie dodawania do końca pliku (jeśli istnieje).

- `b` – modyfikator oznaczający tryb binarny,
- `t` – domyślny, modyfikator oznaczający tryb tekstowy.

Zapis i odczyt plików – parametry

- W trybie tekstowym, kodowanie (parametr `encoding`) ustawiane jest zgodnie z kodowaniem określonym w systemie operacyjnym.
- W trybie binarnym, parametru tego nie trzeba określać.
- Dla plików otworzonych w trybie binarnym zapisujemy i odczytujemy obiekty typu `bytes` (bez określonego kodowania).
- W trybie tekstowym, odczytana zawartość pliku zwracana jest jako typ `str` (z uwzględnieniem kodowania).

Zapis i odczyt plików – przykład

Przykład wczytania pliku tekstowego i wyświetlenia jego zawartości z ponumerowanymi wierszami.

```
import sys

fh = None
try:
    fh = open('znaki.py',mode='r',encoding='utf8')
    t = fh.read()
    linie = t.splitlines()

    nr = 0
    for lin in linie:
        nr += 1
        print(nr, ': ', lin, sep='')

except OSError as err:
    print('{0}: błąd podczas odczytu: {1}'.format(os.path.basename(sys.argv[0])
finally:
    if fh is not None:
        fh.close()
```

Zapis i odczyt plików – przykład

Domyślne strumienie danych wejściowych i wyjściowych zdefiniowane są w zmiennych `sys.stdout` i `sys.stdin` wskazujących na konsolę i klawiaturę.

Możemy przekierować te strumienie wskazując jako źródło zwykły plik.

```
import sys

ekran=sys.stdout

sys.stdout = open('plik2.txt','w')
print('Przykład zapisu  do pliku')
print('Tekst będzie w dwóch wierszach')

sys.stdout=ekran

print(open('plik2.txt','r').read())
```

Zapis i odczyt danych – moduł pickle

Zapis danych wygodnie jest realizować za pomocą modułu `pickle` – który udostępnia tzw. peklowanie. Oznacza to serializację danych, czyli zamianę na sekwencję bajtów.

- Zaletą peklowania jest to, że przetwarzany obiekt może być typem prostym (np. int, float, complex) lub kolekcją (np. tekstem, krotką, listą, zbiorem czy słownikiem).
- Jeśli obiekt zawiera w sobie inne obiekty (łącznie z innymi kolekcjami, które zawierają kolekcje ...), całość zostanie poddana skutecznemu peklowaniu.
- Zapeklowane dane można wczytać bezpośrednio do zmiennej (nie jest konieczne dodatkowe przetwarzanie czy konwersja).
- Peklowane dane nie są w żaden sposób zabezpieczane (brak mechanizmów szyfrowania, podpisu cyfrowego).

Zapis i odczyt danych – moduł pickle

Najważniejsze funkcje:

- `dump(obj, file, protocol)` – zapis danych do pliku,
- `dumps(obj, protocol)` – konwersja danych na sekwencję bajtów (`byte_object`),
- `load(file)` – odczyt obiektu z pliku (protokół jest wykrywany automatycznie),
- `loads(byte_object)` – rekonstrukcja danych z sekwencji bajtów.

Jako protokół zwykle wskazujemy wartość `pickle.DEFAULT_PROTOCOL`. Aktualnie, od wersji 3.8 domyślnie jest to wartość 4 (istnieje od wersji 3.4). Od wersji 3.0 była to wartość 3.

Zapis i odczyt plików – moduł pickle

```
import pickle

a = SparseMartrix3D(3,4,2)
a[1,3,1] = 4
a[2,2,0] = 7
a[0,3,1] = 5

lista = [1, 'abcd', 5.0, a, {1:'x', 2:'y'}]

zapis=pickle.dumps(lista)
lista1=pickle.loads(zapis)

print(lista)
print(lista1)

m = lista1[3]
m.show_matrix()

################################################
[1, 'abcd', 5.0, SparseMartrix3D(3,4,2), {1: 'a', 2: 'b'}]
[1, 'abcd', 5.0, SparseMartrix3D(3,4,2), {1: 'a', 2: 'b'}]
Strona 0:
0 , 0 , 0 , 0 ,
0 , 0 , 0 , 0 ,
```

Zapis i odczyt plików – moduł pickle

```
import pickle

a = SparseMartrix3D(3,4,2)
a[1,3,1] = 4
a[2,2,0] = 7
a[0,3,1] = 5
lista = [1, 'abcd', 5.0, a, {1:'x', 2:'y'}]

fh = None
try:
    fh = open('dane.dat',mode='wb')
    pickle.dump(lista, fh, pickle.HIGHEST_PROTOCOL)

except (OSError, pickle.PicklingError) as err:
    print('Błąd zapisu!')
finally:
    if fh is not None:
        fh.close()
```

Zapis i odczyt plików – moduł pickle

```
import pickle

fh = None
try:
    fh = open('dane.dat',mode='rb')
    (nr, t, w, m, s) = pickle.load(fh) # lub lista = pickle.load(fh)

except (OSError, pickle.PicklingError) as err:
    print('Błąd odczytu!')
finally:
    if fh is not None:
        fh.close()

print(nr, t, w, m, s) # 1 abcd 5.0 SparseMatrix3D: 3 x 4 x 2 {1: 'a', 2: 'b'}
m.show_matrix()       # Strona 0:
                      # 0 , 0 , 0 , 0 ,
                      # 0 , 0 , 0 , 0 ,
                      # 0 , 0 , 7 , 0 ,
                      # Strona 1:
                      # 0 , 0 , 0 , 5 ,
                      # 0 , 0 , 0 , 4 ,
                      # 0 , 0 , 0 , 0 ,
```

Zapis i odczyt plików – moduł pickle

```
import pickle
import gzip

fh = None
try:
    fh = gzip.open('dane.gzip',mode='wb')
    pickle.dump(lista, fh, pickle.HIGHEST_PROTOCOL)

except (OSError, pickle.PicklingError) as err:
    print('Błąd zapisu!')
finally:
    if fh is not None:
        fh.close()
```

Zapis i odczyt plików – moduł pickle

```
import pickle
import gzip

GZIP_MAGIC_NUMBER = b'\x1F\x8B'
fh = None
try:
    fh = open('dane.gzip',mode='rb')
    magic = fh.read(len(GZIP_MAGIC_NUMBER))
    fh.close

    if magic == GZIP_MAGIC_NUMBER: fh = gzip.open('dane.gzip',mode='rb')
    else:                          fh = open('dane.gzip',mode='rb')

    (nr, t, w, m, s) = pickle.load(fh) # lub lista = pickle.load(fh)

except (OSError, pickle.PicklingError) as err:
    print('Błąd odczytu!')
finally:
    if fh is not None:
        fh.close()
```

Menedżer kontekstu

Obiekt zwracany przez wbudowaną funkcję `open()` jest tzw. menedżerem kontekstu.

Składnia używania menedżerów kontekstu jest następująca.

```
with wyrażenie as zmienna:
    blok_kodu
```

`wyrażenie` musi być obiektem menedżera kontekstu lub tworzyć taki obiekt. Zmienna będzie odwołaniem do obiektu.

Menedżer kontekstu

Menedżery kontekstu pozwalają na uproszczenie programu poprzez zagwarantowanie, że określone działania będą przeprowadzone przed i po wykonaniu wskazanego bloku kodu.

Takie zachowanie jest możliwe, ponieważ menedżer kontekstu ma zdefiniowane dwie metody specjalne:

- `__enter__()` – metoda jest automatycznie wywoływana, kiedy menedżer jest tworzony w poleceniu `with`,
- `__exit__()` – metoda jest wywoływana po zakończeniu bloku kodu związanego z poleceniem `with`.

Programista może tworzyć własne menedżery kontekstu bądź używać predefiniowanych.

Menedżer kontekstu

Menedżery kontekstu wykonują kod metody `__exit__()`, nawet po zgłoszeniu wyjątku.

W przypadku otwierania plików pozwala to wyeliminowanie bloku `finally`, bo kod wyjściowy menedżera jakim jest `open()` zawsze zamyka plik.

```
fh = None
try:
    fh =  open('plik.txt', 'w'):
    fh.write('Pierwsza linia\n')
    fh.write('Druga linia')
except OSError as err:
    print('Błąd zapisu')
finally:
    if fh is not None:
        fh.close()
```

```
try:
    with open('plik.txt', 'w') as fh:
        fh.write('Pierwsza linia\n')
        fh.write('Druga linia')
except OSError as err:
    print('Błąd zapisu')
```

Menedżer kontekstu

Możemy też używać więcej niż jednego menedżera kontekstu jednocześnie.

```
with open('plik.txt', 'rb') as fh_oryg, open('plik_kop.txt', 'wb') as fh_kop:
    while True:
        b = fh_oryg.read(1)        # wczytujemy 1 bajt z oryginału
        if not b: break            # nic się nie wczytało? Koniec pliku!
        fh_kop.write(b)            # zapisujemy 1 bajt do kopii
```

Menedżer kontekstu

Jeśli programista chce utworzyć własnego menedżera kontekstu, musi utworzyć klasę dostarczającą metody `__enter__()` i `__exit__()`.

- Kiedy zostanie użyte polecenie `with` względem egzemplarza tego typu klasy, wywołana będzie metoda `__enter__()`, która zwrócić powinna odwołanie do obiektu przypisane do zmiennej.
- Kiedy program opuści blok kody polecenia `with`, wywołana jest metoda `__exit__()` wraz z informacjami szczegółowymi dotyczącymi ewentualnego wyjątku jaki wystąpił.

```
class MyManager:
    def __init__(self, parametry):
        .....

    def __enter__(self):
        .....
        return self

    def __exit__(self, exception_type, exception_value, traceback):
        .....
```

Menedżer kontekstu

```
class Kwadrat:
    def __init__(self, bok=0):  self.podstawa = float(bok)

    def __enter__(self):
        print('Metoda ENTER')
        return self

    def __exit__(self, exception_type, exception_value, traceback):
        print('Metoda EXIT')
        if exception_type is not None:
            print(exception_type, exception_value, traceback)

    def pole(self):
        if self.podstawa < 0.0:  raise ValueError('Ujemny bok!')
        else:                    return self.podstawa**2.0

#################################################################
with Kwadrat(-4) as kwa:
    c = kwa.pole()

# Metoda ENTER
# Metoda EXIT
# <class 'ValueError'> Ujemny bok! <traceback object at 0x000001EDC0C2B148>
```

Funkcje jako obiekty

- Funkcje są obiektami jak wszystko w Pythonie.
- Nazwa funkcji jest odniesieniem do obiektu, który odwołuje się do funkcji.
- Jeśli nazwa funkcji jest użyta bez nawiasów, Python wie, że chodzi o odniesienie do obiektu.

```
>>> def funkcja2(x):
    return x**2

>>> type(funkcja2)
<class 'function'>

>>> y=funkcja2
>>> y(3)
9
```

Funkcje jako obiekty

Możemy tworzyć słowniki lub listy funkcji, które wywoływane są stosownie do opcji wybranej przez użytkownika – jest to dobra alternatywa do rozgałęziania programu za pomocą poleceń `if` – `elif`.

```
def funkcja2(x):
return x**2

def funkcja3(x):
return x**3

def funkcja4(x):
return x**4

def funkcja_1(x):
return x**(-1)

d = {2:funkcja2, 3:funkcja3, 4:funkcja4, -1:funkcja_1}

wybór = input('Podaj funkcję: ')
wybór = int(wybór)
a = d[wybór](2) # 16 po wybraniu 4
b = d[-1](2)    # 0.5
```

Funkcje jako obiekty

Nazwy funkcji mogą być też argumentami innych funkcji.

```
def funkcja2(x):
return x**2

def funkcja3(x):
return x**3

def oblicz(x,funkcja):
    return funkcja(x)

c = oblicz(2, funkcja3) # 8
```

Generatory

- Funkcje (metody) generatorów zawierają wyrażenie `yield`.
- Podczas wywoływania funkcji generatora wartością zwrotną jest iterator.
- Wartości są wyodrębniane pojedynczo z generatora za pomocą metody `__next__()` – każde jej wywołanie zwraca wartość wyrażenia `yield` funkcji generującej.
- Jeśli funkcja generatora zakończy działanie, wówczas następuje zgłoszenie wyjątku `StopIteration`.
- Generator jest zwykle używany tak, jak każdy inny obiekt pozwalający na iterację.

Generatory

```
# funkcja tworzy listę
def litery(pocz, kon):
    lista = []
    while ord(pocz) < ord(kon):
        lista.append(pocz)
        pocz = chr(ord(pocz)+1)

    return lista

for z in litery('c', 'k'):
    print(z)
```

```
# funkcja generuje wartość na żądanie
def litery1(pocz, kon):
    while ord(pocz) < ord(kon):
        yield pocz
        pocz = chr(ord(pocz)+1)

for z in litery1('c', 'k'):
    print(z)
```

```
##################################################################

x = litery('c', 'k')      # ['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
print(x)

x1 = litery1('c', 'k')
print(x1)                 # <generator object litery1 at 0x0000014ADF490518>
print(list(x1))           # ['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

Generatory

- Polecenie `yield` pełni podobną rolę do polecenia `return`.
- Generatory wykonują tzw. leniwe obliczenia – obliczają tylko tę wartość, która jest potrzebna w danej chwili.
- Takie rozwiązanie jest dużo efektywniejsze, niż tworzenie dużych sekwencji.

Generatory

Możliwe jest także tworzenie generatorów tworzących nieskończoną liczbę wartości.

```
def dziesiąte(kolejna=0.0):
    while True:
        yield kolejna
        kolejna +=0.1

lista = []
for x in dziesiąte():
    if x > 0.5: break
    lista.append(x)

lista1 = []
for x in dziesiąte():
    if x > 0.5: break
    lista1.append(x)


print(lista)   # [0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5]
print(lista1)  # [0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5]
```

Generatory

- Istnieje również możliwość tworzenia tzw. wyrażeń generatora.
- Składnia jest podobna do list składanych – różnica polega na zastosowaniu nawiasów okrągłych.

```
gen = (wyrażenie for element in iteracja if warunek)
```

```
gen = (0.1*a for a in range(7))

lista2=[]
for x in gen:
    lista2.append(x)

print(lista2)
# [0.0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001]
```

Generatory

- Wyrażenie `yield` zwraca po kolei każdą wartość.
- Jeśli wywołujący użyje metody `send(wartość)` generatora, wysłana wartość jest wynikiem wyrażenia `yield` w generatorze.
- Można w ten sposób inicjować generator pożądaną wartością.

```
def dziesiąte(kolejna=0.0):
    while True:
        wynik = yield kolejna
        if wynik is None:  kolejna += 0.1
        else:              kolejna = wynik


lista = []
generator = dziesiąte()
for i in range(10):
    x = next(generator)
    if 0.3<=x<=0.5:  x = generator.send(0.8)
    lista.append(x)

print(lista)
# [0.0, 0.1, 0.2, 0.8, 0.9, 1.0, 1.1, 1.200000002, 1.300000003, 1.400000004]
```

Dynamiczne wykonywanie kodu

- Dynamiczne wykonywanie kodu umożliwia przetwarzanie kodu, nie będącego integralną częścią programu.
- Technika jest wykorzystywana przy tzw. wtyczkach Pythonowych, pozwalających na dodawanie nowych funkcji do innych programów.

Dwie podstawowe funkcje wbudowane, które umożliwiają dynamiczne wykonywanie kodu to:

- `eval(tekst, globals, locals)` – zwraca wynik obliczenia pojedynczego wyrażenia tekstowego. Parametry `globals` i `locals` są opcjonalne. Oznaczają odpowiednio kontekst globalny i lokalny kodu, w postaci słowników (nazwa zmiennej i jej wartość).
- `exec(tekst, globals, locals)` – wykonuje kod znajdujący się w zmiennej tekst. Umożliwia przetwarzanie większych fragmentów kodu.

Dynamiczne wykonywanie kodu

```
import math

x = eval('2**5')
x1 = eval('math.sin(math.pi/2)')
print(x,x1)                                  # 32, 1.0

#############################################################

tekst = '''
def pole_koła(r):
    return math.pi * r**2
'''

exec(tekst)

PX = pole_koła(2.0)
print(PX)                                  # 12.566370614359172
```

Dynamiczne wykonywanie kodu

```
# Moduł: pola.py

import math

def pole_kwadratu(a):
    return a**2

def pole_prostokąta(a,b):
    return a*b

def pole_trójkąta(a,h):
    return 0.5*a*h

def pole_koła(r):
    return math.pi * r**2
```

```
import math

fh = None
try:
    fh = open('pola.py', 'r', encoding='utf8')
    tekst = fh.read()
    exec(tekst)
except (IOError,SyntaxError) as err:
    print(err)
finally:
    if fh is not None:
        fh.close()

P1 = pole_koła(2.0)
P2 = pole_kwadratu(3)
P3 = pole_prostokąta(2,5)
P4 = pole_trójkąta(3,8)
print(P1, P2, P3, P4)  # 12.56637 9 10 12.0
```

Dynamiczne wykonywanie kodu

Inne funkcje użyteczne przy dynamicznym wykonywaniu kodu:

- `globals()` zwraca słownik bieżącego kontekstu globalnego.
- `locals()` zwraca słownik bieżącego kontekstu lokalnego.

```
def pole_kuli(r):
    y = locals()
    print(y)                          # {'r': 3}
    return 4 * math.pi * r**2

P5 = pole_kuli(3)

x = globals()
print(x)
# {'__name__': '__main__', '__doc__': None, '__package__': None,
# .....
# 'pole_kuli': <function pole_kuli at 0x00000197EE8BDAE8>, 'P5': 113.0973355292
```

Dynamiczne wykonywanie kodu

Inne funkcje użyteczne przy dynamicznym wykonywaniu kodu:

- `dir(obj)` zwraca listę nazw w zasięgu lokalnym lub nazw (atrybuty i metody) wskazanego obiektu.
- `vars(obj)` zwraca kontekst obiektu (wartości atrybutów) jako słownik (nazwa, wartość).

```
z = SparseMartrix3D()
z1 = vars(z)
z2 = dir(z)

print(z1)
# {'rows': 1, 'columns': 1, 'pages': 1, '_SparseMartrix3D__data': [0]}

print(z2)
# ['_SparseMartrix3D__data', '__class__', '__contains__', '__delattr__',
# '__delitem__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
# '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
# '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
# '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
# '__reversed__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__sub
# '__weakref__', 'columns', 'pages', 'rows', 'show_matrix']
```

Funkcje rekurencyjne

W Pythonie można tworzyć funkcje rekurencyjne.

```
def silnia(n):
    if n <= 1:
        return 1
    else:
        return n * silnia(n-1)

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

a = silnia(50)
print(a)
# 30414093201713378043612608166064768844377641568960512000000000000

for i in range(1,10):
    print(i, fibo(i), end='; ')

# 1 1; 2 1; 3 2; 4 3; 5 5; 6 8; 7 13; 8 21; 9 34;
```

Funkcje rekurencyjne

- Funkcje rekurencyjne mogą stanowić duże obciążenie pod względem ilości obliczeń i wymaganej pamięci.
- Liczbę możliwych do wykonania wywołań rekurencyjnych podaje funkcja: `sys.getrecursionlimit()`. Wartość ta jest domyślnie zwykle ustawiona na 1000.
- Można ją zmienić za pomocą funkcji: `sys.setrecursionlimit(wartość)`, a jej przekroczenie generuje wyjątek `RecursionError`.

Funkcje lokalne

W Pythonie można tworzyć funkcje zagnieżdżone w innych funkcjach – funkcje lokalne.

```
def rownanie_kw(a,b,c):

    def delta():
        nonlocal a,b,c
        return b**2 - 4*a*c

    if delta() >= 0:
        x1 = (-b - delta()**0.5) / (2*a)
        x2 = (-b + delta()**0.5) / (2*a)
        return x1, x2

wynik = rownanie_kw(1,4,2)
print(wynik)
```

W przykładzie użyte jest słowo kluczowe `nonlocal`, które daje dostęp do zmiennej znajdującej się w zewnętrznym zasięgu względem funkcji lokalnej. Przykład działałby poprawnie, również bez tego polecenia.

Dekoratory funkcji i metod

Dekorator to funkcja pobierająca jako jedyny argument funkcję oryginalną i zwracająca nową funkcję zawierającą w sobie udekorowaną funkcję oryginalną (ze zmienionymi lub rozszerzonymi możliwościami).

Użycie dekoratorów pozwala między innymi na:

- kontrolę wartości wyniku zwracanego przez funkcję,
- kontrolę wartości i typu parametrów przekazywanych do funkcji.

Dekoratory funkcji i metod

Dekorator poniżej, sprawdza czy wynik funkcji jest nieujemny. W przypadku gdy jest, generowany jest wyjątek `ValueError` z odpowiednim komunikatem.

```
def wartosc_nieujemna(funkcja):
    def opakowanie(*args, **kwargs):
        wynik = funkcja(*args, **kwargs)
        if wynik < 0.0:
            raise ValueError('Ujemna wartość!')
        return wynik
    return opakowanie

@wartosc_nieujemna  # kontrola wartości funkcji
def delta(a,b,c):
    return b**2 - 4*a*c

w = delta(1,2.5,1)
print(w)                # 2.25
```

Dekorator definiuje nową funkcję lokalną, która wywołuje funkcję oryginalną.

```
def wartosc_nieujemna(funkcja):
    def opakowanie(*args, **kwargs):
        wynik = funkcja(*args, **kwargs)
        if wynik < 0.0:
            raise ValueError('Ujemna wartość!')
        return wynik
    return opakowanie

@wartosc_nieujemna  # kontrola wartości funkcji
def delta(a,b,c):
    return b**2 - 4*a*c

w = delta(1,1,1)

Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 684, in <module>
    w = delta(1,1,1)
  File "C:/Marcin/Projekty/proby.py", line 673, in opakowanie
    raise ValueError('Ujemna wartość!')
ValueError: Ujemna wartość!
```

Dekoratory funkcji i metod

Dekorator poniżej, sprawdza poprawność argumentów przekazywanych do funkcji. Gdy nie są poprawne, generowany jest wyjątek `TypeError` z odpowiednim komunikatem.

```
def prawidlowe_oceny(funkcja):
    def opakowanie(*args, **kwargs):
        for ocena in args:
            if ocena not in [2,3,3.5,4,4.5,5]:
                raise TypeError('Ocena {0} jest nieprawidłowa'.format(ocena))
        wynik = funkcja(*args, **kwargs)
        return wynik
    return opakowanie

@prawidlowe_oceny  # kontrola poprawności argumentów
def srednia_ocen(*args):
    srednia = 0.0
    for ocena in args:
        srednia += ocena
    return srednia/len(args)

s = srednia_ocen(2,3,4,5,3.0,2.0)
print(s)     # 3.1666666666666665
```

```
def prawidlowe_oceny(funkcja):
    def opakowanie(*args, **kwargs):
        for ocena in args:
            if ocena not in [2,3,3.5,4,4.5,5]:
                raise TypeError('Ocena {0} jest nieprawidłowa'.format(ocena))
        wynik = funkcja(*args, **kwargs)
        return wynik
    return opakowanie

@prawidlowe_oceny  # kontrola poprawności argumentów
def srednia_ocen(*args):
    srednia = 0.0
    for ocena in args:
        srednia += ocena
    return srednia/len(args)

s = srednia_ocen(2,3,4,5,3.0,2.0,1)
print(s)
```

```
Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 706, in <module>
    s = srednia_ocen(2,3,4,5,3.0,2.0,1)
  File "C:/Marcin/Projekty/proby.py", line 692, in opakowanie
    raise TypeError('Ocena {0} jest nieprawidłowa'.format(ocena))
TypeError: Ocena 1 jest nieprawidłowa
```

Dekoratory funkcji i metod

Dekoratory mogą mieć też przekazywane parametry. Realizuje się to poprzez utworzenie funkcji, do której przekazujemy parametry i która zwraca dekorator, wykorzystany dalej do udekorowania funkcji oryginalnej.

Dekorator może być używany wielokrotnie do dekorowania różnych funkcji.

Dekorator pokazany poniżej, sprawdza czy wynik funkcji należy do podanego zakresu i jeśli nie należy, zwraca jedną z wartości brzegowych.

```
def zakres(minimum, maximum):
    def dekorator(funkcja):
        def opakowanie(*args,**kwargs):
            wynik = funkcja(*args,**kwargs)
            if wynik > maximum:
                return maximum
            elif wynik < minimum:
                return minimum
            else:
                return wynik
        return opakowanie
    return dekorator
```

Dekoratory funkcji i metod

```
@zakres(18,70)
def wiek_osoby_pelnoletniej():
    wiek = int(input('Podaj wiek: '))
    return wiek

w = wiek_osoby_pelnoletniej()
print(w)

Podaj wiek: 35
35

Podaj wiek: 10
18
```

Dekoratory funkcji i metod

```
@zakres(0,100)
def oblicz_procent_przeceny(stara_cena, nowa_cena):
    wynik = 100.0 * (nowa_cena - stara_cena)/stara_cena
    return wynik

w = oblicz_procent_przeceny(200,550)
print(w)

100

w = oblicz_procent_przeceny(200,250)
print(w)

25.0
```

Właściwości obiektów

Rozpatrzmy klasę reprezentującą kwadrat.

```
class Kwadrat():
    def __init__(self, bok=0):
        self.podstawa = float(bok)

    def __str__(self):
        return 'Kwadrat o boku: {0.podstawa}'.format(self)

    def __repr__(self):
        return 'Kwadrat({0.podstawa})'.format(self)

    def __eq__(self,other):
        return self.podstawa == other.podstawa

    def pole(self):
        return self.podstawa**2.0

a = Kwadrat(); b = Kwadrat(-3); c = Kwadrat(3)

print(a==b)                          # False
print(a,b,c)                         # Kwadrat o boku: 0.0 Kwadrat o boku: -3.0
print(repr(a), repr(b), repr(c))     # Kwadrat(0.0) Kwadrat(-3.0) Kwadrat(3.0)
print(a.pole(), b.pole(), c.pole())  # 0.0 9.0 9.0
```

Właściwości – dekorator property

```
class Kwadrat():
    .....
    def pole(self):
        return self.podstawa**2.0
```

- Metoda `pole()` zwraca jedną wartość `float`.
- Można ją inaczej napisać jako **właściwość** klasy z użyciem dekoratora `@property`.
- Dekorator jest funkcją (w tym przypadku wbudowaną funkcją `property()`), która pobiera jako argument funkcję i zwraca jej wersję 'udekorowaną' (zmodyfikowaną).

```
class Kwadrat():
    .....
    @property
    def pole(self):
        return self.podstawa**2.0
```

Właściwości obiektów

- Funkcja wbudowana `property()` pobiera maksymalnie 4 argumenty: funkcję pobierającą wartość, funkcję ustawiającą wartość, funkcję usuwającą wartość i dokumentujący ciąg tekstowy.
- Efekt użycia dekoratora `@property` jest taki sam jak wywołanie funkcji `property()` z jednym tylko argumentem w postaci funkcji pobierającej wartość.

```
class Kwadrat():
    .....
    def pole(self):
        return self.podstawa**2.0
    pole = property(pole)
```

Właściwości obiektów

```
class Kwadrat():
    def __init__(self, bok=1.0):
        self.podstawa = float(bok)

    def __str__(self):
        return 'Kwadrat o boku: {0.podstawa}'.format(self)

    def __repr__(self):
        return 'Kwadrat({0.podstawa})'.format(self)

    def __eq__(self,other):
        return self.podstawa == other.podstawa

    @property
    def pole(self):
        return self.podstawa**2.0

a = Kwadrat(); b = Kwadrat(3)

print(a==b)                 # False
print(a,b, sep = ', ')      # Kwadrat o boku: 1.0, Kwadrat o boku: 3.0
print(repr(a), repr(b))     # Kwadrat(1.0) Kwadrat(3.0)
print(a.pole, b.pole)       # 1.0 9.0  -  użycie pole() nie działa
```

Właściwości obiektów

- Za pomocą atrybutów zdefiniowanych jako właściwości, można łatwo kontrolować poprawność argumentów przekazywanych przy tworzeniu obiektów.
- Aby zmienić atrybut klasy na właściwość pozwalającą na odczyt i zapis, trzeba utworzyć atrybut prywatny i napisać metody, które odczytują i zapisują jego wartość.

```
class Kwadrat():
    def __init__(self, bok=1.0):
        self.podstawa = float(bok)

    @property
    def podstawa(self):
        return self.__podstawa

    @podstawa.setter
    def podstawa(self, podstawa):
        if  podstawa <=0:
            raise AttributeError('Bok kwadratu musi być większy od 0!')
        self.__podstawa = podstawa
```

Właściwości obiektów

- Po zdefiniowaniu atrybutu jako własność, normalnie się go używa odczytując i przypisując do niego wartość.
- W tle jednak, wykonywane są metody zdefiniowane przez dekoratory, działające na zmiennej prywatnej `__podstawa`.

```
self.podstawa = 2.0

print(self.podstawa)
```

Właściwości obiektów

- Wartość podstawy jest przechowywana w atrybucie prywatnym `__podstawa`.
- Każda właściwość ma atrybuty: `getter`, `setter`, `deleter`, więc po utworzeniu właściwości `podstawa` za pomocą dekoratora `@property`, dostępne są atrybuty: `podstawa.getter`, `podstawa.setter`, `podstawa.deleter`.
- Atrybut `podstawa.getter` powstaje automatycznie po użyciu dekoratora `@property`. Pozostałe dwa możemy (ale nie musimy) utworzyć samodzielnie.
- Metoda `podstawa()` jest zdefiniowana 2 razy, jednak dekoratory odpowiednio zmieniają jej nazwę, więc nie stanowi to problemu.
- Użycie w metodzie `__init__()` polecenia: `self.podstawa = float(bok)` wywołuje atrybut `podstawa.setter`, dzięki czemu możliwa jest kontrola wartości.

Właściwości obiektów – przykład

```
class Kwadrat():
    def __init__(self, bok=1.0):
        self.podstawa = float(bok)

    def __str__(self):
        return 'Kwadrat o boku: {0.podstawa}'.format(self)
    def __repr__(self):
        return 'Kwadrat({0.podstawa})'.format(self)
    def __eq__(self,other):
        return self.podstawa == other.podstawa

    @property
    def podstawa(self):
        return self.__podstawa

    @podstawa.setter
    def podstawa(self, podstawa):
        if  podstawa <=0:
            raise AttributeError('Bok kwadratu musi być większy od 0!')
        self.__podstawa = podstawa

    @property
    def pole(self):  return self.__podstawa**2.0
```

Właściwości obiektów – przykład

```
a = Kwadrat(); b = Kwadrat(3)

print(a==b)                    # False
print(a, b, sep = ', ')        # Kwadrat o boku: 1.0, Kwadrat o boku: 3.0
print(a.podstawa)              # 1.0
print(repr(a), repr(b))        # Kwadrat(1.0) Kwadrat(3.0)
print(a.pole, b.pole)          # 1.0 9.0

c = Kwadrat(-2)

Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 205, in <module>
    c = Kwadrat(-2)
  File "C:/Marcin/Projekty/proby.py", line 178, in __init__
    self.podstawa = float(bok)
  File "C:/Marcin/Projekty/proby.py", line 196, in podstawa
    raise AttributeError('Bok kwadratu musi być większy od 0!')
AttributeError: Bok kwadratu musi być większy od 0!
```

Dekorator `staticmethod`

- Metody statyczne nie modyfikują atrybutów obiektu i nie wymagają jego wcześniejszego utworzenia.
- Umożliwiają dodanie do obiektu funkcjonalności, które mogą być także użyteczne poza nim.
- Umożliwiają grupowanie w ramach jednego obiektu różnych funkcji użytkowych.
- Przy wywoływaniu metody statycznej, nie przekazujemy odniesienia `self` jako pierwszego argumentu.

```
class Kalkulator:

    @staticmethod
    def dodawanie(x, y):
        return x + y

    @staticmethod
    def odejmowanie(x, y):
        return x - y

print('+:', Kalkulator.dodawanie(7, 4), 'i -:', Kalkulator.odejmowanie(7, 4))
```

Dekoratory klas

- Podobnie jak można tworzyć dekoratory funkcji i metod, istnieje możliwość tworzenia dekoratorów klas.
- Dekorator klasy pobiera obiekt klasy (wynik wykonania polecenia `class`) i powinien zwrócić klasę w postaci udekorowanej.

Adnotacje funkcji

Funkcje i metody można zdefiniować wraz z adnotacjami – wyrażeniami, które umieszczane są w nagłówku funkcji.

```
def funkcja(arg1: typ1, arg2: typ2, ..., argN: typN) -> typ_wy:
    polecenia
```

- Każda część składająca się z dwukropka i typu (`: typN`) jest opcjonalną adnotacją.
- Podobnie opcjonalne jest określenie typu wyniku (`-> typ_wyniku`).
- Ostatni (lub jedyny) parametr pozycyjny (o ile występuje) w postaci `*args` może być bez adnotacji lub z nią.
- Podobnie ostatni (lub jedyny) parametr w postaci słów kluczowych `**kwargs` może być bez adnotacji lub z nią.
- Składnia pozwala na adnotowanie wszystkich parametrów, wybranych, bądź żadnego.

Adnotacje funkcji

- Jeżeli adnotacje są dostępne, wówczas będą dodane do słownika `__annotations__` funkcji.
- Gdy adnotacji nie ma, słownik będzie pusty.
- Kluczami słownika są nazwy parametrów, a wartościami odpowiadający im typ.
- Adnotacje nie mają żadnego bezpośredniego wpływu na działanie funkcji.
- Jedyne co robi Python, to umieszczenie ich w słowniku – reszta zależy od programisty.

Adnotacje funkcji

```
def delta(a, b, c):
    print(dir())
    print(delta.__annotations__)
    print(type(a), type(b), type(c))
    return b**2 - 4*a*c

w = delta(1,3,2)
print(w)

############################################################

['a', 'b', 'c']
{}
<class 'int'> <class 'int'> <class 'int'>
1
```

Adnotacje funkcji

```
def delta(a:float, b:float, c:float) -> float :
    print(dir())
    print(delta.__annotations__)
    print(type(a), type(b), type(c))
    if isinstance(a, int):
        a = float(a)
    print(type(a),type(b),type(c))
    return b**2 - 4*a*c

w = delta(1,3,2)
print(w)

############################################################

['a', 'b', 'c']
{'a': <class 'float'>, 'b': <class 'float'>, 'c': <class 'float'>,
'return': <class 'float'>}
<class 'int'> <class 'int'> <class 'int'>
<class 'float'> <class 'int'> <class 'int'>
1.0
```

Adnotacje funkcji

```
def delta(a:int, b:float, c:bool = 5) -> str :
    print(dir())
    print(delta.__annotations__)
    print(type(a), type(b), type(c))
    if isinstance(a, int):
        a = float(a)
    print(type(a),type(b),type(c))
    return b**2 - 4*a*c

w = delta(1,3)
print(w)

#########################################################

['a', 'b', 'c']
{'a': <class 'int'>, 'b': <class 'float'>, 'c': <class 'bool'>,
'return': <class 'str'>}
<class 'int'> <class 'int'> <class 'int'>
<class 'float'> <class 'int'> <class 'int'>
-11.0
```

Adnotacje funkcji

```
def delta(a:float, b, c:float) -> float :

{'a': <class 'float'>, 'c': <class 'float'>, 'return': <class 'float'>}

##########################################################

def delta(a:float, b:float, c:float, *args:int, **kwargs:str) -> float:

{'a': <class 'float'>, 'b': <class 'float'>, 'c': <class 'float'>,
'args': <class 'int'>, 'kwargs': <class 'str'>, 'return': <class 'float'>}
```

Asercje

Asercje pozwalają na sprawdzenie przez programistę, czy kod nie wykonuje błędnych działań. Sprawdzenie jest realizowane za pomocą słowa kluczowego `assert`.

```
assert warunek_logiczny, komunikat
```

- Jeśli warunek logiczny jest równy `False`, zgłoszony będzie wyjątek `AssertionError`.
- Komunikat jest opcjonalny. Jeśli będzie podany, będzie przekazany jako argument wyjątku – w ten sposób można wygodnie wyświetlić komunikat o błędzie.
- Asercje są przeznaczone dla programistów, a nie użytkowników końcowych.
- W przeciwieństwie do innych wyjątków, wyjątek `AssertionError` nie powinien być obsługiwany za pomocą konstrukcji `try ... except`.
- Jeśli warunek logiczny w asercji jest fałszywy powinien wystąpić błąd, który wskaże nam błędne działanie kodu.

Asercje

```
def średnia_geometryczna(*args):
    wynik = 1.0
    for x in args:
        wynik *= x

    assert len(args) > 0, 'Brak argumentów'
    assert wynik != 0.0, 'Jeden z argumentów jest zerowy'
    assert wynik > 0.0, 'Niektóre z argumentów są ujemne'

    return wynik ** (1/len(args))

x = średnia_geometryczna()
print(x)

############################################################

Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 770, in <module>
    x = średnia_geometryczna()
  File "C:/Marcin/Projekty/proby.py", line 764, in średnia_geometryczna
    assert len(args) > 0, 'Brak argumentów'
AssertionError: Brak argumentów
```

Asercje

```
def średnia_geometryczna(*args):
    wynik = 1.0
    for x in args:
        wynik *= x

    assert len(args) > 0, 'Brak argumentów'
    assert wynik != 0.0, 'Jeden z argumentów jest zerowy'
    assert wynik > 0.0, 'Niektóre z argumentów są ujemne'

    return wynik ** (1/len(args))

x = średnia_geometryczna(1,2,3,4,5)
print(x)

############################################################

2.605171084697352
```

Asercje

```
def średnia_geometryczna(*args):
    wynik = 1.0
    for x in args:
        wynik *= x

    assert len(args) > 0, 'Brak argumentów'
    assert wynik != 0.0, 'Jeden z argumentów jest zerowy'
    assert wynik > 0.0, 'Niektóre z argumentów są ujemne'

    return wynik ** (1/len(args))

x = średnia_geometryczna(1,2,-3,4,5)
print(x)

############################################################

Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 770, in <module>
    x = średnia_geometryczna(1,2,-3,4,5)
  File "C:/Marcin/Projekty/proby.py", line 766, in średnia_geometryczna
    assert wynik > 0.0, 'Niektóre z argumentów są ujemne'
AssertionError: Niektóre z argumentów są ujemne
```

Asercje

```
def średnia_geometryczna(*args):
    wynik = 1.0
    for x in args:
        wynik *= x

    assert len(args) > 0, 'Brak argumentów'
    assert wynik != 0.0, 'Jeden z argumentów jest zerowy'
    assert wynik > 0.0, 'Niektóre z argumentów są ujemne'

    return wynik ** (1/len(args))

x = średnia_geometryczna(1,2,0,4,5)
print(x)

############################################################

Traceback (most recent call last):
  File "C:/Marcin/Projekty/proby.py", line 770, in <module>
    x = średnia_geometryczna(1,2,0,4,5)
  File "C:/Marcin/Projekty/proby.py", line 765, in średnia_geometryczna
    assert wynik != 0.0, 'Jeden z argumentów jest zerowy'
AssertionError: Jeden z argumentów jest zerowy
```

Asercje

Kiedy program jest gotowy, asercje można wyłączyć. W tym celu możemy uruchamiać program z opcją `-O`

```
python -O program.py
```

- W trakcie wykonania programu wszystkie polecenia `assert` będą wtedy ignorowane.
- Asercje przeznaczone są do stosowania na etapie tworzenia programu, a nie w ostatecznym produkcie.