Python – podstawy programowania

Marcin Pluciński

mplucinski@zut.edu.pl

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 3 – Python 2 ?

- Aktualne wersje Python 3.11.5 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
```

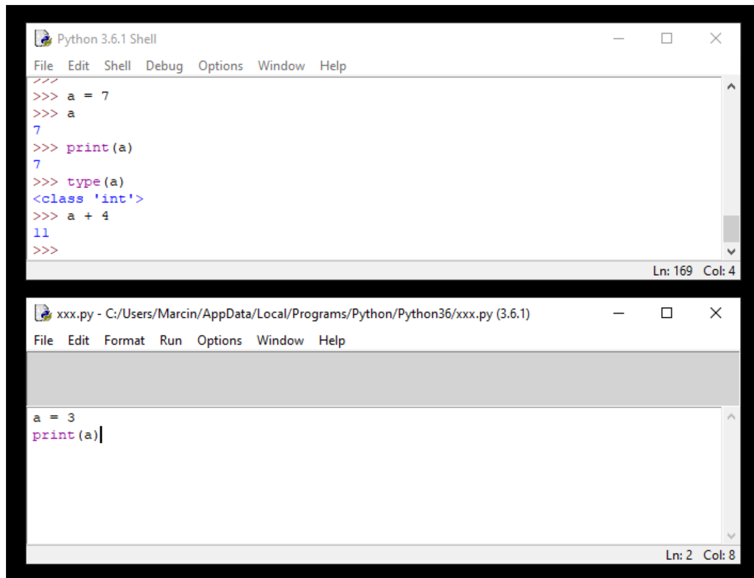
Python – środowisko

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'
>>>
```

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
```

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('1.414213562373095048801688724')
>>> Decimal(1).exp()
Decimal('2.718281828459045235360287471')
>>> Decimal('10').ln()
Decimal('2.302585092994045684017991455')
>>> 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]
```

Iterator

- 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.

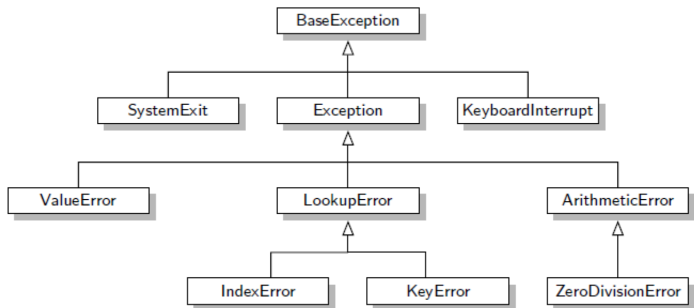
Obsługa wyjątków

```
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.

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")
```