Python – podstawy programowania

Marcin Pluciński

mplucinski@zut.edu.pl

Fundusze Europejskie
Wiedza Edukacja Rozwój

Rzeczpospolita Polska

Unia Europejska
Europejski Fundusz Społeczny

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

Python – charakterystyka

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

Python – charakterystyka

Python 3 – Python 2 ?

- Aktualne wersje Python 3.11.5 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
```

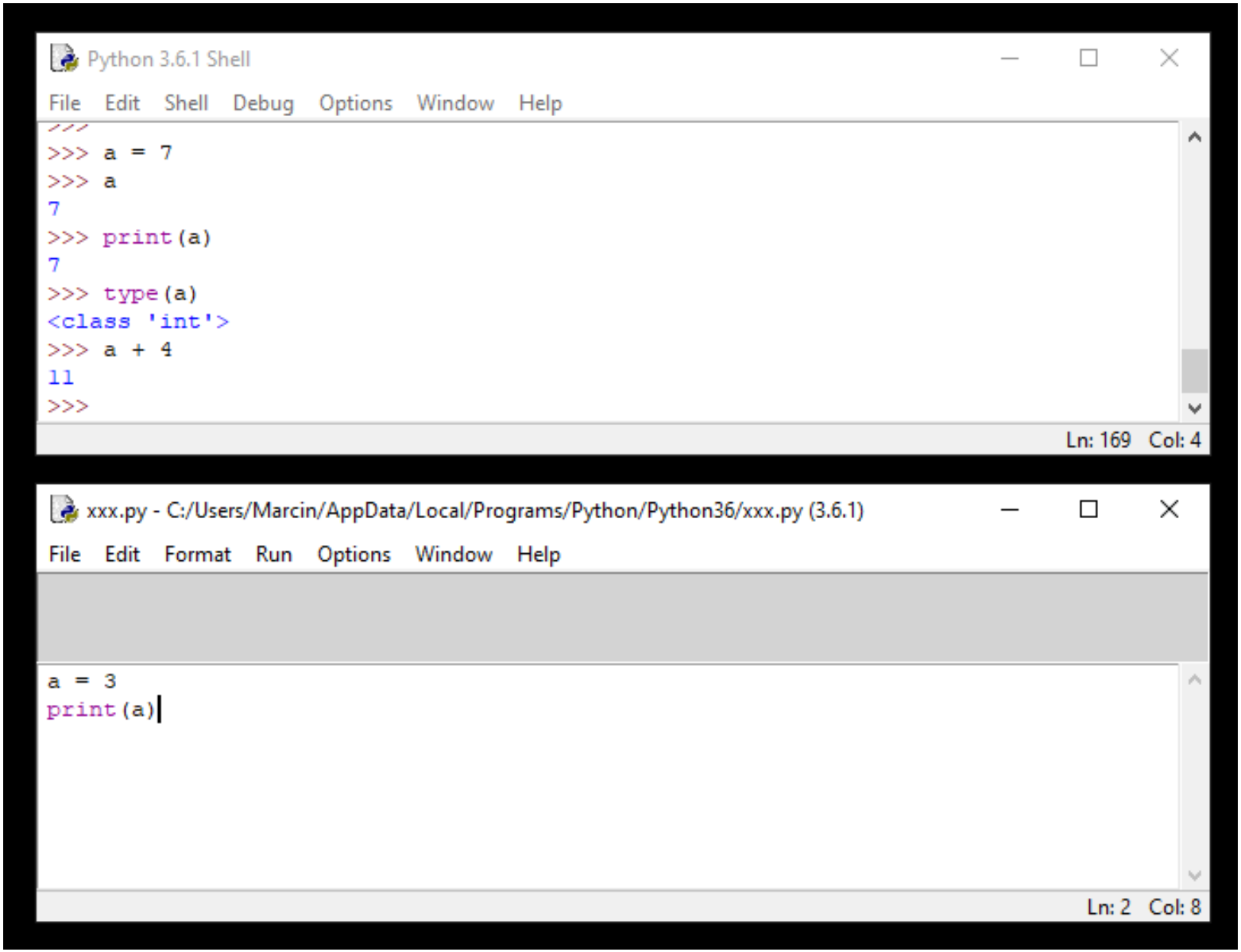
Python – środowisko

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))
```

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

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

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')
```

```
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)
```