Python 3

Jest to język wieloparadygmatowy, zawiera między innymi paradygmat programowania obiektowego i funkcyjnego.

Operatory arytmetyczne

```
+, -, *, /, **(potęgowanie), //(dzielenie bez reszty), %
```

Wyrażenia – działania wykonywane w kolejności znanej z algebry:

```
>>> 25+2*-3**3+12/3
-25
```

W wyrażeniach używamy nawiasów okrągłych:

```
>>> -((2+1)*(8-3))**2
-225
```

Działają operatory: `+=, -=, *=, /=`

Systemy liczbowe

Liczba w systemie ósemkowym:

```
>>> 0764
500
```

Liczba w systemie szesnastkowym:

```
>>> 0x100
256
```

Typy liczbowe

- liczby całkowite (32 bity) np. 7
- długie liczby całkowite np. 7L
- liczby rzeczywiste np. 2.25 lub 1e+2
  całkowity wynik dzielenia:
  ```
  >>> 3.0//2.0
  1.0
  ```
- liczby zespolone:
  ```
  >>> -1+1j
  (-1+1j)
  ```

Język nietypowany – typ jest określany na podstawie przypisanej wartości.
type(x) – podaje typ zmiennej

Komentarze

```
#To jest komentarz - do końca linii
"""To jest komentarz
wieloliniowy"""
```

Napisy

Napisy ograniczamy cudzysłowami lub apostrofami.

```
>>> "napis"
'napis'


>>> 'inny napis'
'inny napis'
```

W napisach ograniczonych cudzysłowami można używać apostrofów:

```
>>> "I can't help"
"I can't help"
```

W napisach ograniczonych apostrofami można używać cudzysłowów:

```
>>> '"Moby Dick" is thick'
'"Moby Dick" is thick'
```

Napisy ograniczone pojedynczymi cudzysłowami bądź apostrofami muszą kończyć się przed końcem linii.

Napisy mogą ciągnąć się przez wiele linii jeżeli ograniczymy je potrójnymi cudzysłowami:

```
>>> """Ten napis
ma
wiele
linii"""
'Ten napis\nma\nwiele\nlinii'
```

Operacje na napisach

```
>>> p="pies"
>>> k="kot"
```

Łączenie:

```
 >>> p+k
'pieskot'
```

Powielanie:

```
>>> p*3
'piespiespies'
```

Łączenie + powielanie:

```
>>> 2*k+" "+p
'kotkot pies'
```

Napisy w Pythonie są niezmienne np. `k[0]='l'` - błąd

Konwersje

Liczba -> napis

```
>>> a=2
```

```
>>> "A="+str(a)
'A=2'
```

Napis -> liczba

```
>>> x="1"
>>> int(x)
1
>>> long(x)
1L
>>> float(x)
1.0
>>> complex(x)
(1+0j)
```

Konwersja na ASCII:

```
chr(65) - zwraca A
ord("A") - zwraca 65
```

Napisy (stringi) są obiektami. Składnia wywołania metody obiektu przypomina C++. Przykłady metod, które można wywołać dla napisu s:
s.capitalize() – zwraca napis ze zmienioną pierwszą literą na wielką
s.isdigit() – sprawdza, czy wszystkie znaki są cyframi
s.islower() – sprawdza, czy wszystkie litery są małe
s.center(długość) – centruje napis w polu o podanej długości (uzupełniając spacjami)
s.rjust(długość) – wyrównuje do prawej w polu o podanej długości (uzupełniając spacjami)
s.count(s1) – zlicza wystąpienia podciągu s1 w s
s.lstrip() – zwraca napis z usuniętymi wiodącymi białymi znakami

Funkcja len(s) zwraca długość ciągu.

Funkcje te nie zmieniają s.

Instrukcja warunkowa

```
if warunek:
   # zrób coś
elif warunek:
  # tu też zrób coś
else:
  # ewent. zrób coś w takim przypadku
```

W warunkach można używać operatorów koniunkcji (and), alternatywy (or) i negacji (not).

Przykład:

```
if x > 3 and y == -1 and 2 < z < 4:
  print (…)
```

Listy

Listy w Pythonie to w zasadzie tablice dynamiczne.
Elementy listy nie muszą być tego samego typu.
Przykład:

```
lista = ["abc", 31, -3.14, 2+4j, [2, 1]]
print (lista[2])   # -3.41
print (lista[-1])   # [2, 1]
print (type(lista))  # 'list'
```

Elementem listy może być inna lista – standardowy sposób otrzymywania tablic 2- i więcej wymiarowych.

```
lista1 = []  # ta lista jest pusta
len(lista1)  # wyświetli 0
lista[1:3]   # wyświetli [31, -3.14]
lista[1::2]  # co drugi element zaczynając od lista[1]
```

Listy można powielać, np. `lista *= 2`,
Można doklejać inną listę na końcu: `lista1 += lista2`

```
# zakładając, że lista2 istnieje i jest typu "list"
```

albo inaczej: `lista1.extend(lista2)`

Można wyrzucać elementy z listy:

```
del lista1[1:3]  # elementy na pozycjach 1 i 2
del lista[-2]    # przedostatni element
```

Doklejenie elementu na końcu:

```
x = 2
lista1.append(x)
```

albo:

```
lista1 += [x]
```

Listy można porównywać przy użyciu operatorów `==`, `!=`, a także `>`, `>=`, `<`, `<=`

Porównywanie list odbywa się na zasadzie porównywania poszczególnych elementów:

- jeżeli elementy obu list są sobie równe, listy są równe
- jeżeli listy różnią się choć jednym elementem, to są nierówne
- jeżeli pierwszy element pierwszej listy jest większy od pierwszego elementu drugiej listy, to pierwsza lista jest większa od drugiej
- jeżeli pierwszy element pierwszej listy jest taki sam jak pierwszy element drugiej listy, decyduje porównanie drugich elementów, itd.
- element nieistniejący jest zawsze mniejszy od każdego innego elementu

Można sprawdzić, czy dany element należy do listy:
```
31 in lista  # otrzymamy True
```

Sortowanie: `lista.sort()`

Krotki (typ tuple)

Krotka przypomina listę, tyle że jest niemutowalna (niezmienialna). Składnia:
```
kolor = (128, 0, 255)   # nawiasy okrągłe
```

Próba zmiany jakiejś składowej nie powiedzie się:
```
kolor[1] = 20  # błąd
```

Jeśli chcemy zmienić krotkę, trzeba powołać do życia nowy obiekt o tej samej nazwie:
```
kolor = (128, 20, 255)
```

Pętla for

```
for i in range(10):
  print (i)       # wypisze liczby od 0 do 9
```

`range` generuje zakres, tzn. listę złożoną z liczb naturalnych tworzących szereg arytmetyczny. Są 3 warianty `range`:
```
range(n)          # n > 0, lista [0, 1, 2, ..., n-1]
range(m, n)       # m < n, lista [m, m+1, …, n-1]; jeśli m >= n lista pusta
range(m, n, step) # jak wyżej, ale co step wartości
np.:
range(0, 12, 2)   # [0, 2, 4, 6, 8, 10]
range(5, 1, -1)   # [5, 4, 3, 2]
```

Iterowanie po liście:
```
for i in lista:
  print (i)
```

Od pierwszego elementu:
```
for i in lista[1:]:
  print (i)
```

Iterowanie po stringu:
```
s = "Witaj!"
for i in s:
  print (i)
```

Z pętli można wyjść przez break (jak w C):
```
for i in range(20):
  print (i)
  if i % 5 == 4:
    break
```

Pętla while

```
x = -1
while x < 3:
  print (x)
  x += 1
print ("koniec")
```

Nie ma pętli do..while. Można ją zasymulować za pomocą break:

```
while True:
  ...
  if warunek_wyjścia:
     break
# ciąg dalszy programu
```

Słowniki

W słowniku dostęp do dowolnej wartości przechowywanej w słowniku możliwy jest poprzez podanie klucza do niej.
Słownik składa się zatem ze zbioru kluczy i zbioru wartości, gdzie każdemu kluczowi przypisana jest pojedyncza wartość. Klucz nie musi być liczbą, wystarczy, że jest typu niezmiennego. Można powiedzieć więc, że o ile lista czy krotka odwzorowuje liczby całkowite (indeksy) na obiekty dowolnego typu, o tyle słownik odwzorowuje obiekty dowolnego typu niezmiennego na obiekty dowolnego typu.

```
tel = {"policja":997, "straz":998, "pogotowie":999}
```

`len(tel)  #3` – zwraca liczbę kluczy
`tel ["policja"]  #997` – zwraca wartość klucza
`tel ["taxi"] =222` – dadawanie nowego klucza
`tel2 = tel` – tworzy alias
`del tel ["taxi"]` – usuwa element ze słownika
`tel.keys() # ['policja','straz','pogotowie'] -` zwraca listę kluczy
`tel.values()# [997,998,999] -` zwraca listę wartości

Zbiory

Zbiór ma dwie ważne cechy: obiekty w zbiorze nie mogą się powtarzać oraz dostęp do konkretnych elementów zbioru (znalezienie elementu w zbiorze lub stwierdzenie, że go nie ma) jest bardzo szybki.

```
zbiorPusty = set()
zbior = {1, 3, 5}
print(zbiorPusty)
## set()
print(zbior)
## {1, 3, 5}
```

```
print(1 in zbiorPusty)
## False
print(1 in zbior)
## True

zbior.add(2)- dodanie element do zbioru
zbior.discard(2) – usunięcie element ze zbioru
{1, 5}.issubset({1, 5, 9}) – czy zbiór jest podzbiorem innego zbioru
```

Opreracje na zbiorach:

```
print({1,5,8} | {1,5,9}) # suma
## {1, 5, 8, 9}
print({1, 5, 8} - {1, 5, 9}) # różnica
## {8}
print({1, 5, 8} & {1, 5, 9}) # przecięcie
## {1, 5}
```

Własne funkcje

Funkcja to podprogram, zwykle zawierający parametry (argumenty), który może coś zwrócić. Składnia:

```
def dodaj(a,b):
  return a + b
```

Taka funkcja zadziała dla argumentów typu *int*, ale też dla stringów (wynikiem będzie konkatenacja stringów):

```
x, y = 3, -10
print (dodaj(x,y))

s1 = "Ala "
s2 = dodaj(s1, "ma kota")
print (s2)
```

Funkcja może coś zwracać (za pomocą instrukcji return), ale nie musi.

W Pythonie funkcja może być argumentem jakieś funkcji, może być wynikiem (wartością zwracaną) funkcji, można ją podstawiać pod zmienne etc.
Łatwo sprawdzić, co jest funkcją, a co nie – wbudowana funkcja callable():

```
callable(list), callable(int), callable(len), callable(-2)
```

Podstawiając funkcję pod zmienną można nadać funkcjom (wbudowanym w język albo wziętym z jakichś „gotowych” modułów, albo naszym własnym) inne nazwy, aliasy:

```
import math
pierwiastek = math.sqrt   # bez nawiasów!
print (pierwiastek(2))
print (math.sqrt(2))  # oczywiście „po staremu” też działa
długość = len
```

```
print (długość("Ala ma kota"))
```

Zmiana argumentów wewnątrz funkcji:
Stringi, krotki i liczby są niemutowalne, więc nie zostaną zmienione, listy – mogą być zmieniane.

Funkcja może zwrócić wiele wartości (zwrócenie krotki), np.

```
def suma_roznica_iloczyn(x, y):
  return x+y, x-y, x*y
```

Gdy nie jest znana liczba argumentów możemy napisać:

```
def sumuj(powitanie, *ell):
# gwiazdka oznacza: bierz argumenty od tego miejsca do końca
  print (powitanie)
  s = 0
  for i in ell:
    s += i
  return s

print (sumuj("napis powitalny :-)", 4, 2, 3, -1, 8))
print (sumuj("jeszcze raz", 4, 1))
```

Pliki

```
f = open("c:/topsecret/mypassword.txt", "rt")
text = f.read()
```

Zmienna text przechowuje zawartość wskazanego pliku jako napis (string). Gdyby f został otwarty w trybie binarnym (‘b’ zamiast ‘t’), to też funkcja read() zwróciłaby string.

Obiekty plikowe mają 3 atrybuty: `f.name, f.mode, f.closed`.

Jeżeli chcemy plik „na raz” wczytać do pamięci, nie trzeba nawet tworzyć zmiennej plikowej:

```
words = open("c:/test.txt").read().split()
print (words[:30])
words.count("and")
```

`readlines()` – zwraca listę wierszy

Zapis do pliku – metoda `write()` (argumentem jest string).

```
g = open("c:/output.txt", "w")
f.write("Oto kilka początkowych liczb pierwszych:\n")
for i in [2, 3, 5, 7, 11]:
```

```
  f.write(str(i)+" ")
f.close()
```

Można też jednym wywołaniem metody *writelines()* zapisać do pliku listę stringów:

```
lista = ["napis1 ", "napis 2 ", "napis 3 "]
f.writelines(lista)
```

Foldery

```
import os

os.path.exists(sciezka_do_pliku)- sprawdza, czy plik istnieje
os.listdir("C:\folder\") – wypisuje wszystkie pliku z folderu
os.path.join("folder1", "folder2", "plik.txt") – skleja ścieżkę
#folder1/folder2/plik.txt – zgodnie z systemem operacyjnym
os.remove(sciezka_do_pliku) – usuwa plik
```