Wypisywanie

```
print(‘aaaa’)
print( Wiek: {}, waga {} kg'.format(17, 55))
liczba = input(‘podaj liczbe’)
```

Zmienne – poniżej typy niezmienne**:**

```
print(type(‘aaa’))
print(type(4)) # nieograniczony niczym oprócz pamięci komputera rozmiar !!!!
print(type(4.0)) # zakres zależy od kompilatora C użytego do kompilacji Pythona
napis = ‘napis’
napis[3] = ‘a’ # error - bo string jest niezmienny tak jak int, float
```

Dynamiczna kontrola typów zmiennych – typ ustalany w momencie przypisania wartości do zmiennej
Konwersja typów: typ(element), możliwa zmiana typu zmiennej w trakcie programu

```
a = ‘aaa’
a = 4

A = ‘jeden’
B = ‘dwa’
C = A
B = A

print(A)
print(B)
print(C)
```

Uwaga: wszystko jest w Pythonie obiektem, również np. int.

Wszystko jest obiektem, del usuwa odniesienie do obiektu, ale nie usuwa samego obiektu – usuwa go automatyczny Garbage Collection
Brak odniesień do obiektu – Python sam go usuwa

Nazwa musi zaczynać się od litery lub _ , w nazwie mogą być litery, cyfry, _

Komentarze w Pythonie oznacza się symbolem **„#**” - brak komentarzy blokowych.

WYRAŻENIA WARUNKOWE

```
If wyrazenie:
        kod
elif wyrazenie:
        kod
else:
        kod
```

UWAGA: W Pythonie w instrukcjach warunkowych if, pętlach for itd. nie stosuje się klamer (w przeciwieństwie do C++). Zamiast nich służą wcięcia (tabulacje).

Fałsz, gdy False, None, pusta kolekcja, liczba równa zero, w. p. p. prawda

A is None # sprawdź, czy A jest „Nullem „

Porównania, standard != == łączenie ich; kontrola typu jest!!!

a = 5
0 <= a < 9

ASERCJE
assert len(args) > 0, 'Brak argumentów'

PĘTLE
for x in range(5):
    print(i) # 0, 1, 2, 3, 4

while warunek:
    zrob_cos

break, continue – jak w C++

brak do while !!!

OPERATORY ARYTMETYCZNE
+ - * / (dzielenie daje float)
x//y całkowite, x % y, mod, x**y – potęgowanie, abs(x) – moduł
są też +=, … %=, **= - tworzy to nowy obiekt, do niego odnosi się zmienn

STRUKTURY DANYCH

Struktury danych: LISTA, KROTKA, SŁOWNIK –też są to obiekty tak jak int, z tym że zmienne (mutable)

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

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

a = [] # stworzenie pustej listy
B = [1, 'napis', 4.0] # listy mogą mieć mieszanego typu obiekty
a.append(3) # dodanie elementu
a.insert(index, element) # umieszczenie elementu na pozycji index
a.index(element) # pierwsze wystąpienie
a.reverse()
a.sort()
a.remove(element) # usuwa ostatnie wystąpienie
'napis' in a # czy jest element w liście

a[0] # weź element pierwszy
a[i] = ‘nowy’ # zastąpi element
len(lista)

+ na listach, A=[1,2,3] , A*3 = ?, string to samo
for x in lista
a is not None

KOPIE GŁĘBOKIE
lista1 = [1, 2, 3]
lista2 = a
lista1 [0] = 0
print(lista2) # płytka kopia

Rozwiązanie:
import copy
lista2 = copy.deepcopy(lista1)

Krotka (tupla) to uporządkowana i niezmienna sekwencja zera lub większej liczby odniesień do obiektów. Krotki definiujemy w nawiasach () lub za pomocą funkcji tuple. Konwersja na listę: list (krotka)

krotka = (1, 2, ‘aaa’)
t[0] = 0 # terror
len(krotka)

Słownik

S = {} # uwaga: klucze musza być unikalne
S2 = {‘wiek’: 20, ‘Plec’: ’M’}
S2[‘wiek’] # 20
S2[‘nowy_klucz’] = 4 # dodaje wartość z nieistniejącym kluczem
del S2[key] # skasowanie klucza i wartości

FUNKCJE
def nazwa_funkcji(argument_wymagany, argument_domyslny=None):
    Kod

Zawsze zwraca, domyślnie None ewentualnie return wartość

WYRAŻENIE LAMBDA – odpowiednik funkcji @ w Matlabie
parametry: wyrażenie
pole_trapezu = lambda a,b,h: 0.5*(a+b)*h
pole_trapezu(2,5,6)

KLASY

```
class KlasaTestowa(KlasaBazowa):
        __init__(self):
            super().__init__() # wywołanie konstruktora klasy bazowej (Python 3!!!)

        def metoda(self):
            pass # nic nie rób – "pusta" metoda

nowy = Klasa_testowa()
nowy.pole
nowy.metoda()
```

dziedziczenie wielokrotne , brak private/public (metody private– 2 podkreślenia na początku w praktyce dla odróżnienia)

PAKIETY

instalacja w konsoli pip install nazwa_pakietu
przydatne pakiety: os (wczytywanie plików), pickle (zapisywanie obiektów)

IMPORT MODUŁÓW

```
import math
print(math.sqrt(4))
```

NUMPY

```
Import numpy as np

M = np.array([[1,2,3],[4, 5, 6],[7, 8, 9]])
print(M.dtype) # typ danych – musi być jeden dla całej macierzy
print(M.shape) # wymiar macierzy
A = np.array([1.0, 2.5, 3.7], dtype=int)

A+B, A+10,  A-10

A*B # jak z kropką

A.dot(B) lub np.dot(A, B) # iloczyn normalny

np.min(A), np. max(A)  # axis=0  w kolumnie, axis=1 – w wierszu

np.sum(A) # suma wszystkich elementów, można również wierszami lub kolumnami jak wyżej

print(A>=3) – macierz logiczna

A.T – transpozycja # lub np.transpose(A)
```

Macierz jednowymiarowa indeksowana jak lista !!, inne krotką np. A[1,2], A[:, -1] # -1 – ostatnia kolumna

A[A>8] # macierz logiczna

np.vstack((A, B)), np.hstack((A, B)) # łączenie macierzy w pionie lub poziomie

A = B.reshape(3,4) # zmiana wymiaru na 3x4 (musi być odpowiednia liczba elementów)

X = A.flatten() # „spłaszcznie” do jednego wymiaru

np.argwhere(A>1) # zwróci indeksy elementów większych od 1

```
a = np.zeros(5)
b = np.zeros((2,4))
c = np.zeros_like(M) # jak macierz m rozmiar i typ
a = np.ones(4)
b = np.ones((2,5))
c = np.ones_like(M)

a = np.random.rand(5)
x = np.random.rand(3,4)
```

ZAKRESY

```
a = np.arange(5)
a = np.arange(4.0, 1.0, -0.5)
>>> array([4. , 3.5, 3. , 2.5, 2. , 1.5])
b = np.linspace(0, 2, 5)
>>> array([0. , 0.5, 1. , 1.5, 2. ])
```

LOGGER – stosowany do debuggowania, znacznie szybszy niż print

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG) # ustawienie
#formatu I poziomu komunikatów np. level=logging.WARN sprawi, że nie pokażą się komunikaty z
#poziomu niższego czyli DEBUG i INFO
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warn('This is a warn message')
```