Wizualizacja danych
Wykład 2

Operacje wejścia i wyjścia - dokończenie

input - operacja wejścia

```
1 num = input ("Wprowadź liczbę :")
2 print(num)
3 name1 = input("Wprowadź imię : ")
4 print(name1)
```

- zawsze przyjmuje napis, w razie potrzeby trzeba zrzutować

```
1 x = str(num)
2 y = int(num)
3 z = float(num)
```

print - instrukcja wyjścia

```
1 print(4.2)
2 # print('Mój wiek to' + 36)
3 print('Mój wiek to', 36)
4 a = 36
5 print('Mój wiek to', a)
6 print('hello', 'world')
7 print('hello', 'world', sep='')
8 print('hello', 'world', sep='\n')
```

```
4.2
Mój wiek to 36
Mój wiek to 36
hello world
helloworld
hello
world
```

```
1 print(*objects, sep=' ', end='\n', file=sys.stdout,
2       flush=False)
```

- `objects` - to co ma być wyświetlone
- `sep` - separator, domyślnie znak spacji
- `end` - co co ma być wyświetlone na końcu, domyślnie znak końca linii
- `file` - określa gdzie mają być `objects` wyświetlone, domyślnie `sys.stdout` (domyślny ekran)
- `flush`- określa czy “wyjście” ma być buforowane przed przekazaniem do `file`, domyślne `False`

```python
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4 ,sep='#', end='&')
```

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

```
1 print('x', 'y', 'z', sep='', end='')
2 print('a', 'b', 'c' , sep='', end='')
```

```
xyzabc
```

```
1 print('a', 'b', '\n', 'c')
```

```
a b
 c
```

\t - przesunięcie do następnego “tab”=8 spacji

```
1 print('sdf', 3456, -2, sep='\t')
```

```
sdf 3456    -2
```

Formatowanie napisów będzie później.

Operatory

Operacje arytmetyczne

Operator	Opis	Składnia
+	Dodawanie	x + y
–	Odejmowanie	x – y
*	Mnożenie	x * y
/	Dzielenie	x / y
//	Dzielenie całkowite	x // y
%	Dzielenie modulo	x % y
**	Potęgowanie	x ** y

```
1 print(5+3)
2 print(4*5.2)
3 print(9-7)
4 print(25%7)
```

```
8
20.8
2
4
```

```
1 print(4/5)
2 print(4//5)
3 print(4/5.0)
4 print(4//5.0)
```

```
0.8
0
0.8
0.0
```

```
1 print(3**0)
2 print(0**0)
```

```
1
1
```

```
1 print(4/0)
```

Daje info: `ZeroDivisionError: division by zero`.

Przypisanie z operacją arytmetyczną

Lista zawiera wybrane operacje.

Inna nazwa to złożone operatory przypisania.

Operator	Zapis	Dłuższa wersja
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5

```
a = 5
a += 1
print(a)
a **= 2
print(a)
```

```
6
36
```

Operatory porównania

Operator	Znaczenie	Przykład
>	Większe niż	x > y
<	Mniejsze niż	x < y
==	Równe	x == y
!=	Nie równa się	x != y
>=	Większe lub równe	x >= y
<=	Mniejsze lub równe	x <= y

Operatory logiczne

Operator	Znaczenie	Przykład
and	i	x and y
or	lub	x or y
not	negacja	not x

Operatory bitowe

Operator	Znaczenie	Przykład
&	i - logiczne	x & y
|	lub - logiczne	x | y
^	albo - logiczne	x ^ y
~	negacja - logiczne	~x
<<	przesunięcie w lewo	x << y
>>	przesunięcie w prawo	x >> y

Operator &

```
1 print(4&5)
```

```
4
```

x	100	4
y	101	5
x&y	100	4

Operator |

```
1 print(4|5)
```

```
5
```

x	100	4
y	101	5
x|y	101	5

Operator ^

```
1 print(4^5)
```

```
1
```

x	100	4
y	101	5
x^y	001	1

Operator `~`

- równoważnie `-(x+1)`

```
1 print(~4)
```

```
-5
```

`x`	`100`	4
`~x`	?	-5

Operator <<

a<<b - równoważnie a*pow(2,b)

```
1 print(3<<2)
```

```
12
```

x	0011	3
x<<2	1100	12

Operator >>

a>>b - równoważnie a//pow(2,b)

```
1 print(13>>2)
```

```
3
```

x	1101	13
x>>2	0011	3

Instrukcje warunkowe i pętle

Instrukcje warunkowe

Składnia

```
1 if <expr>:
2     <statement>
```

else

```
1 if <expr>:
2     <statement(s)>
3 else:
4     <statement(s)>
```

```
1 a = 5
2 if a > 0:
3     print("Liczba dodatnia")
4 else:
5     print("Liczna ujemna lub zero")
```

```
Liczba dodatnia
```

```
x = 0
y = 5
if x < y:
    print('yes1')

if y < x:
    print('yes2')

if x:
    print('yes3')

if y:
    print('yes4')

if x or y:
    print('yes5')

if x and y:
    print('yes6')
```

```
yes1
yes4
yes5
```

elif

```
if <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
elif <expr>:
    <statement(s)>
    ...
else:
    <statement(s)>
```

```
1 a = 5
2 if a > 0:
3     print("Liczba dodatnia")
4 elif a == 0:
5     print("Zero")
6 else:
7     print("Liczna ujemna")
```

```
Liczba dodatnia
```

Zagnieżdżone instrukcje warunkowe:

```
1 if <expr>:
2     <statement(s)>
3     if <expr>:
4         <statement(s)>
```

```
1  i = 21
2  if i > 0:
3      print("liczba jest dodatnia")
4      if i % 2 == 0:
5          print("Liczba jest dodatkowo parzysta")
```

```
liczba jest dodatnia
```

for - pętla

```
1 for i in <collection>:
2     <loop body>
```

Przykład:

```
1 for i in range(5):
2     print(i)
```

```
0
1
2
3
4
```

range

Generuje nam ciąg liczb (dedykowany typ `range`). Trzeba zamienić na listę “by podejrzeć”.

Uwaga: wszystkie argumenty muszą być w typie całkowitym.

Jeden argument - to “koniec” - ciąg tworzą liczby naturalne od 0 do $n - 1$. Krok domyślny to 1.

Dwa argumenty - to “początek” i “koniec”. Krok domyślny to 1. Wtedy wyświetlone są liczby całkowite z przedziału lewostronnie domkniętego $[początek, koniec)$.

Trzy argumenty - to “początek”, “koniec” oraz krok.

```
print(list(range(5)))
print(list(range(1, 11)))
print(list(range(0, 30, 5)))
print(list(range(0, 10, 3)))
print(list(range(0, -10, -1)))
print(list(range(0)))
print(list(range(1, 0)))
```

```
[0, 1, 2, 3, 4]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 5, 10, 15, 20, 25]
[0, 3, 6, 9]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
[]
[]
```

While - pętla

```
while <expr>:
    <statement(s)>
```

Przykład:

```
i = 0
while i < 5:
    print(i)
    i = i + 1
```

```
0
1
2
3
4
```

Zagnieżdżone pętle

```
1 for i in <collection>:
2     <loop body>
3     for i in <collection>:
4         <loop body>
```

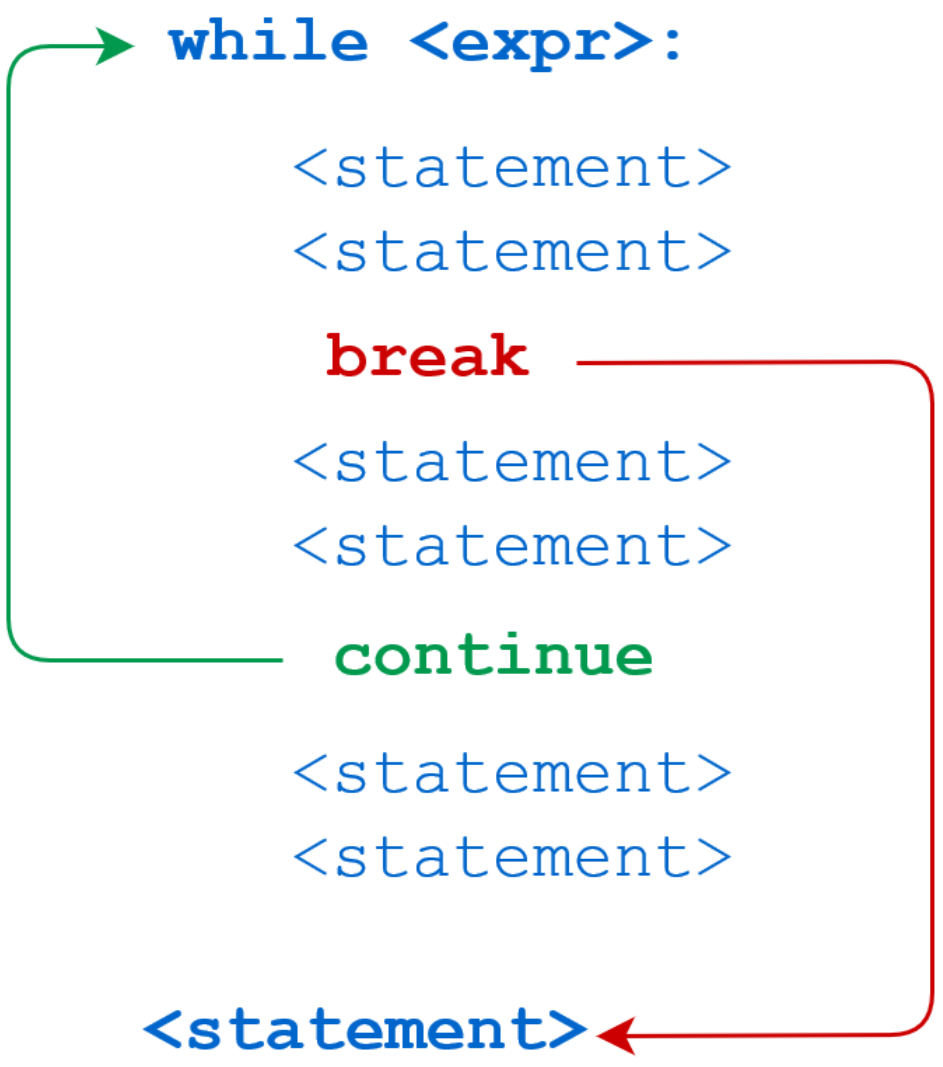
inna opcja:

```
1 while <expr>:
2     <statement(s)>
3     while <expr>:
4         <statement(s)>
```

```python
for i in range(3):
    for j in range(3):
        print(i, "*", j, "=", i * j)
```

```
0 * 0 = 0
0 * 1 = 0
0 * 2 = 0
1 * 0 = 0
1 * 1 = 1
1 * 2 = 2
2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
```

break/continue



break

```
1 n = 5
2 while n > 0:
3     n -= 1
4     if n == 2:
5         break
6     print(n)
```

```
4
3
```

continue

```
1 n = 5
2 while n > 0:
3     n -= 1
4     if n == 2:
5         continue
6     print(n)
```

```
4
3
1
0
```

Kolejność operaratorów

Kolejność operatorów

Od ostatniego:

- `lambda`
- `if - else`
- `or`
- `and`
- `not x`
- `in, not in, is, is not, <, <=, >, >=, !=, ==`
- `|`
- `^`
- `&`
- `<<, >>`
- `+, -`

- `*, @, /, //, %`
- `+x, -x, ~x`
- `**`
- `await x`
- `x[index], x[index:index], x(arguments...), x.attribute`
- `(expressions...), [expressions...], {key: value...}, {expressions...}`

Źródło: https://docs.python.org/3/reference/expressions.html#operator-precedence.

Pytanie do przemyślenia

Co oznacza w Pythonie, że wartości przekazywane są przez referencję?

```
1 a = 5
2 b = a
3 b += 2
4 print(a)
5 print(b)
```

```
5
7
```

Typy sekwencyjne

Listy

Lista w Pythonie to tzw. typ sekwencyjny umożliwia przechowywanie elementów różnych typów.

Cechy:

- zmienny (`mutable`) - umożliwia przypisanie wartości pojedynczym elementom
- do zapisu używamy nawiasów kwadratowych
- poszczególne elementy rozdzielamy przecinkami
- każdy element listy ma przyporządkowany indeks
- elementy listy są numerowane od zera
- listy dopuszczają porządek (o ile elementy są porównywalne)
- listy są dynamiczne (mogą mieć różną długość)
- listy mogą być zagnieżdżone

Uwaga!

Listy w języku Python są specyficzną strukturą danych nie zawsze dostępną w innych językach programowania. Pojęcie listy w całej informatyce “szersze”. Wyróżnia się np. listy jednokierunkowe, które nie muszą mieć indeksu. Nie będziemy takich przypadków analizować.

```
1 nazwa = [element1, element2, ..., elementN]
```

Pusta lista:

```
1 a = []
2 b = list()
```

Lista z liczbami:

```
1 a = [2, 3, 4.5, 5, -3.2]
```

Lista mieszana:

```
1 b = ['abcd', 25+3j, True, 1]
```

Kolejność ma znaczenie

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

```
False
```

Elementy na liście nie muszą być unikalne

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

```
[1, 2, 3, 4, 2]
False
```

Indeks - dostęp do elementów listy (od zera)

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(a[1])
3 print(a[4])
4 print(a[0])
5 #print(a[7])
```

```
3
-2.3
1
```

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(a[-1])
3 print(a[-5])
4 print(a[-7])
```

```
9.3
abc
1
```

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(a[1:4])
3 print(a[-5:-2])
4 print(a[:4])
```

```
[3, 'abc', False]
['abc', False, -2.3]
[1, 3, 'abc', False]
```

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(a[2:])
3 print(a[0:6:2])
4 print(a[1:6:2])
```

```
['abc', False, -2.3, 'XYZ', 9.3]
[1, 'abc', -2.3]
[3, False, 'XYZ']
```

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(a[6:0:-2])
3 print(a[::-1])
4 print(a[:])
```

```
[9.3, -2.3, 'abc']
[9.3, 'XYZ', -2.3, False, 'abc', 3, 1]
[1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
```

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(a[::2])
3 print(a[::-2])
```

```
[1, 'abc', -2.3, 9.3]
[9.3, -2.3, 'abc', 1]
```

Specjalne funkcje

Długość (rozmiar listy)

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 print(len(a))
```

```
7
```

Implementacja samodzielna długości:

```
 1 def dlugosc(lista):
 2     x = 0
 3     for i in lista:
 4         x += 1
 5 
 6     return x
 7 
 8 
 9 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
10 print(dlugosc(a))
```

```
7
```

Maksimum i minimum?

Działa wtedy gdy mamy porządek

- liczby `<=`
- napisy - porządek leksykograficzny (omówimy przy napisach)

```
a = [4,-5,3.4,-11.2]
print(min(a))
print(max(a))
b = ['abc', 'ABCd', 'krt', 'abcd']
print(min(b))
print(max(b))
```

```
-11.2
4
ABCd
krt
```

Modyfikacja i wstawianie

```
1 a = [4,-5,3.4,-11.2]
2 a[2] = 'a'
3 print(a)
```

```
[4, -5, 'a', -11.2]
```

```
1 a = [4,-5,3.4,-11.2]
2 a[2] = ['a','b']
3 print(a)
```

```
[4, -5, ['a', 'b'], -11.2]
```

```
1 a = [4,-5,3.4,-11.2]
2 a[1:2] = ['a','b']
3 print(a)
```

```
[4, 'a', 'b', 3.4, -11.2]
```

```
1 a = [4,-5,3.4,-11.2]
2 a[1:3] = ['a','b']
3 print(a)
```

```
[4, 'a', 'b', -11.2]
```

Dodawanie list

```
1 a = [4,-5,3.4,-11.2]
2 b = ['a','b','c']
3 print(a+b)
```

```
[4, -5, 3.4, -11.2, 'a', 'b', 'c']
```

Mnożenie listy przez liczbę całkowitą (`int`)

```
1 a = [4,-5,3.4,-11.2]
2 print(a*2)
3 print(2*a)
```

```
[4, -5, 3.4, -11.2, 4, -5, 3.4, -11.2]
[4, -5, 3.4, -11.2, 4, -5, 3.4, -11.2]
```

Usuwanie elementów z listy

```
1 a = [1, 3, 'abc', False, -2.3, 'XYZ', 9.3]
2 del a[2]
3 print(a)
4 del a[:]
5 print(a)
```

```
[1, 3, False, -2.3, 'XYZ', 9.3]
[]
```

Do poczytania

- https://docs.python.org/3.10/tutorial/introduction.html#lists
- https://docs.python.org/3.10/tutorial/datastructures.html#more-on-lists

`list.append(x)` - dodaje element na końcu listy.

Równoważnie `a[len(a):] = [x]`

```
1 a = [1, 3, 'abc', False]
2 a.append(5.3)
3 print(a)
```

```
[1, 3, 'abc', False, 5.3]
```

`list.extend(iterable)` - dodaje elementy z argumenty na koniec listy. Równoważnie: `a[len(a):] = iterable`

```
1 a = [1, 3, 'abc', False]
2 b = [3, -2]
3 a.extend(b)
4 print(a)
```

```
[1, 3, 'abc', False, 3, -2]
```

Różnice?

```
1 a = [1, 3, 'abc', False]
2 b = [3, -2]
3 a.append(b)
4 print(a)
```

```
[1, 3, 'abc', False, [3, -2]]
```

`list.insert(i, x)` - wstawia element `x` na pozycji `i`

```
1 a = [1, 3, 'abc', False]
2 a.insert(0,'w')
3 print(a)
4 a.insert(4,9.0)
5 print(a)
```

```
['w', 1, 3, 'abc', False]
['w', 1, 3, 'abc', 9.0, False]
```

`list.remove(x)` - usuwa element z listy (pierwszy od początku)

```
1 a = [1, 3, 'abc', False]
2 a.remove(False)
3 print(a)
4 b = [3, 4, 5, 3]
5 b.remove(3)
6 print(b)
```

```
[1, 3, 'abc']
[4, 5, 3]
```

`list.pop()` - usuwa i zwraca ostatni element

`list.pop(i)` - usuwa i zwraca element na pozycji `i`

```
1 a = [1, 3, 'abc', False]
2 print(a.pop())
3 print(a)
4 b = [3, -4, 6.2, 7]
5 print(b.pop(3))
6 print(b)
```

```
False
[1, 3, 'abc']
7
[3, -4, 6.2]
```

`list.clear()` - usuwa wszystkie elementy z listy.

Równoważnie: `del a[:]`

```
1 a = [1, 3, 'abc', False]
2 a.clear()
3 print(a)
```

```
[]
```

`list.index(x)` - zwraca indeks elementu x (o ile istnieje, inaczej błąd), w przypadku duplikatów pierwszy z lewej
`list.index(x,start)` - zwraca indeks elementu x (o ile istnieje, inaczej błąd) zaczynając od pozycji `start`, w przypadku duplikatów pierwszy z lewej
`list.index(x,start,end)` - zwraca indeks elementu x (o ile istnieje, inaczej błąd) zaczynając od pozycji `start` a kończąc na `end-1`, w przypadku duplikatów pierwszy z lewej

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

```
1
6
1
```

```
a = ['abc','xyz','abc','efg']
print(a.index('abc'))
print(a.index('abc', 2))
print(a.index('abc', 1, 4))
```

```
0
2
2
```

`list.count(x)` - zwraca ile razy występuję element `x` na liście

```
1 a = ['abc','xyz','abc','efg']
2 print(a.count('abc'))
3 print(a.count(4))
```

```
2
0
```

`list.sort()` - sortuje listę (o ile elementy można posortować)

```
1 a = ['abc','xyz','abc','efg']
2 a.sort()
3 print(a)
```

```
['abc', 'abc', 'efg', 'xyz']
```

`list.reverse()` - odwraca kolejność elementów na liście (nie ma nic związku z sortowaniem!)

```
1 a = [4, 5, -2, 7.3, 9, -22, 23]
2 a.reverse()
3 print(a)
```

```
[23, -22, 9, 7.3, -2, 5, 4]
```

`list.copy()` - tworzy kopię listy

Spójrzmy na przykład jak działa operator przypisania dla list.

```
1 a = [4, 5, -2, 7.3, 9, -22, 23]
2 b = a
3 b[2] = 100
4 print(b)
5 print(a)
```

```
[4, 5, 100, 7.3, 9, -22, 23]
[4, 5, 100, 7.3, 9, -22, 23]
```

Różnica z użyciem copy.

```
1 a = [4, 5, -2, 7.3, 9, -22, 23]
2 b = a.copy()
3 b[2] = 100
4 print(b)
5 print(a)
```

```
[4, 5, 100, 7.3, 9, -22, 23]
[4, 5, -2, 7.3, 9, -22, 23]
```

Lista jako stos

```
1 stack = [3, 4, 5, 8, 9]
2 stack.append(6)
3 stack.append(7)
4 print(stack)
5 print(stack.pop())
6 print(stack)
```

```
[3, 4, 5, 8, 9, 6, 7]
7
[3, 4, 5, 8, 9, 6]
```

Lista jako kolejka

```
1 from collections import deque
2
3 queue = deque(["aw", "tg", "kj"])
4 queue.append("gg")
5 print(queue)
6 print(queue.popleft())
7 print(queue)
```

```
deque(['aw', 'tg', 'kj', 'gg'])
aw
deque(['tg', 'kj', 'gg'])
```

List Comprehensions

```
squares = []
for x in range(5):
    squares.append(x ** 2)

print(squares)
```

```
[0, 1, 4, 9, 16]
```

```
squares = [x**2 for x in range(5)]
print(squares)
```

```
[0, 1, 4, 9, 16]
```

Krotka - `tuple`

```
1  krotka = 123, 'abc', True
2  krotka2 = (123, 'abc', True)
3  print(krotka[2])
```

```
True
```

```
1  krotka[0] = 1
```

TypeError: ‘tuple’ object does not support item assignment

https://docs.python.org/3.10/library/stdtypes.html#tuple

Zbiór - set

```
1 cyfry = {'raz', 'dwa', 'raz', 'trzy', 'raz', 'osiem'}
2 print(cyfry)
```

```
{'osiem', 'raz', 'dwa', 'trzy'}
```

https://docs.python.org/3.10/library/stdtypes.html#set

```
1 x = {2, 3, 4, -3, 5, 2}
2 y = {5, 6, 7}
3 z = {'a', 'b'}
4 w = {3, 4}
5 print(x)
6 print(len(x))
7 print(0 in x)
8 print(0 not in x)
```

```
{2, 3, 4, 5, -3}
5
False
True
```

```
x = {2, 3, 4, -3, 5, 2}
y = {5, 6, 7}
z = {'a', 'b'}
w = {3, 4}
print(x.isdisjoint(y))
print(x.isdisjoint(z))
print(w.issubset(x))
print(x.issubset(w))
print(w <= x)
print(w < x)
print(w.issuperset(x))
print(x.issuperset(w))
print(w >= x)
print(w > x)
```

```
False
True
True
False
True
True
False
True
False
False
```

```python
x = {2, 3, 4, -3, 5, 2}
y = {5, 6, 7}
z = {'a', 'b'}
w = {3, 4}
print(x.union(y))
print(x | y)
print(x.intersection(y))
print(x & y)
print(x.difference(y))
print(x - y)
print(x.symmetric_difference(y))
print(x ^ y)
print(x.copy())  # płytka kopia
```

```
{2, 3, 4, 5, 6, 7, -3}
{2, 3, 4, 5, 6, 7, -3}
{5}
{5}
{2, 3, 4, -3}
{2, 3, 4, -3}
{2, 3, 4, 6, 7, -3}
{2, 3, 4, 6, 7, -3}
{2, 3, 4, 5, -3}
```

```
x = {2, 3, 4, -3, 5, 2}
x.update({11})
print(x)
x |= {12}
print(x)
x = {2, 3, 4, -3, 5, 2}
x.intersection_update({5})
print(x)
x = {2, 3, 4, -3, 5, 2}
x &= {5}
print(x)
x = {2, 3, 4, -3, 5, 2}
```

```
{2, 3, 4, 5, 11, -3}
{2, 3, 4, 5, 11, 12, -3}
{5}
{5}
```

```
x = {2, 3, 4, -3, 5, 2}
x.difference_update({4})
print(x)
x = {2, 3, 4, -3, 5, 2}
x -= {4}
print(x)
x = {2, 3, 4, -3, 5, 2}
x.symmetric_difference_update({4, 11})
print(x)
x = {2, 3, 4, -3, 5, 2}
x ^= {4, 11}
print(x)
```

```
{2, 3, 5, -3}
{2, 3, 5, -3}
{2, 3, 5, 11, -3}
{2, 3, 5, 11, -3}
```

```python
x = {2, 3, 4, -3, 5, 2}
print(x)
x.add(11)
print(x)
x.remove(-3)  # usuwa gdy jest, inaczej KeyError
print(x)
x.discard(12)
print(x)
```

```
{2, 3, 4, 5, -3}
{2, 3, 4, 5, 11, -3}
{2, 3, 4, 5, 11}
{2, 3, 4, 5, 11}
```

```
x = {2, 3, 4, -3, 5, 2}
print(x)
x.discard(2)
print(x)
x = {2, 3, 4, -3, 5, 2}
print(x.pop())
print(x)
x.clear()
print(x)
```

```
{2, 3, 4, 5, -3}
{3, 4, 5, -3}
2
{3, 4, 5, -3}
set()
```

Słownik

```
1 tel = {'jack': 4098, 'sape': 4139}
2 tel['guido'] = 4127
3 print(tel)
4 tel['jack']
5 del tel['sape']
6 tel['irv'] = 4127
7 print(tel)
```

```
{'jack': 4098, 'sape': 4139, 'guido': 4127}
{'jack': 4098, 'guido': 4127, 'irv': 4127}
```

https://docs.python.org/3.10/library/stdtypes.html#mapping-types-dict

Słownik a typowanie

```
1 x: dict[str, float] = {"field": 2.0}
```

po staremu:

```
1 from typing import Dict
2
3 x: Dict[str, float] = {"field": 2.0}
```

```python
d = {"one": 1, "two": 2, "three": 3, "four": 4}
print(d)
print(list(d))
print(list(d.values()))
d["one"] = 42
print(d)
del d["two"]
print(d)
d["two"] = None
print(d)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
['one', 'two', 'three', 'four']
[1, 2, 3, 4]
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
{'one': 42, 'three': 3, 'four': 4}
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

```
1 d = {"one": 1, "two": 2, "three": 3, "four": 4}
2 print(d)
3 print(list(reversed(d)))
4 print(list(reversed(d.values())))
5 print(list(reversed(d.items())))
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
['four', 'three', 'two', 'one']
[4, 3, 2, 1]
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

```python
d = {"one": 1, "two": 2, "three": 3, "four": 4}
print("one" in d)
print(1 in d)
print("one" not in d)
print(1 not in d)
print(iter(d))
for elem in iter(d):
    print(elem)
```

```
True
False
False
True
<dict_keyiterator object at 0x000001BCDAF03AB0>
one
two
three
four
```

```python
d = {"one": 1, "two": 2, "three": 3, "four": 4}
d.clear()
print(d)
d = {"one": 1, "two": 2, "three": 3, "four": 4}
e = d.copy() # płyta kopia
print(e)
```

```
{}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
```

```
x = ('key1', 'key2', 'key3')
y = 0
d1 = dict.fromkeys(x, y)
print(d1)
z = (3, 4, 5)
d2 = dict.fromkeys(x, z)
print(d2)
```

```
{'key1': 0, 'key2': 0, 'key3': 0}
{'key1': (3, 4, 5), 'key2': (3, 4, 5), 'key3': (3, 4, 5)}
```

```python
d = {"one": 1, "two": 2, "three": 3, "four": 4}
print(d.get("two"))
print(d.items())
print(d.keys())
print(d.pop("three"))
print(d)
print(d.popitem())
print(d)
```

```
2
dict_items([('one', 1), ('two', 2), ('three', 3), ('four', 4)])
dict_keys(['one', 'two', 'three', 'four'])
3
{'one': 1, 'two': 2, 'four': 4}
('four', 4)
{'one': 1, 'two': 2}
```

```
d = {"one": 1, "two": 2, "three": 3, "four": 4}
d.update(red=1, blue=2)
print(d)
print(d.values())
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'red': 1, 'blue': 2}
dict_values([1, 2, 3, 4, 1, 2])
```

```python
d = {"one": 1, "two": 2, "three": 3, "four": 4}
d2 = {"a": 11, "b": 12}
d3 = d | d2
print(d3)
print(d)
d |= d2
print(d)
```

```
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'a': 11, 'b': 12}
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'a': 11, 'b': 12}
```

Python comprehension

```
1 a = [3, 4, 5]
2 print(a)
3 a2 = [i**2 for i in a]
4 print(a2)
5 a3 = {i**2 for i in a}
6 print(a3)
7 a4 = {i: i**2 for i in a}
8 print(a4)
```

```
[3, 4, 5]
[9, 16, 25]
{16, 9, 25}
{3: 9, 4: 16, 5: 25}
```

```python
a = [3, 4, 5]
print(a)
a2 = (i**2 for i in a)
print(a2)
for i in a2:
    print(i)
```

```
[3, 4, 5]
<generator object <genexpr> at 0x000001BCDAEEB5A0>
9
16
25
```

```python
a = [3, 4, "w", 5]
print(a)
w = [i*i for i in a if isinstance(i, int)]
print(w)
```

```
[3, 4, 'w', 5]
[9, 16, 25]
```

```python
a = [3, 4, "w", 5]
print(a)
w = [i*i if isinstance(i, int) else i for i in a]
print(w)
```

```
[3, 4, 'w', 5]
[9, 16, 'w', 25]
```

Napisy

- trochę podobne do listy
- typ sekwencyjny do przechowywania znaków, ale w odróżnieniu od listy jest niezmienny
- w języku Python nie ma oddzielnego typu znakowego
- apostrofy i cudzysłów można stosować zamiennie, ale konsekwentnie

Inne nazwy: - string, napisy, łańcuchy znaków

Abstrakcyjnie:

- na końcu każdego napisu jest znak “zerowy” - będzie widać lepiej w C/C++

Tablica znaków ASCII https://upload.wikimedia.org/wikipedia/commons/5/5c/ASCII-Table-wide.pdf

```
1 a = "Olsztyn"
2 print(a)
3 print(a[3])
4 #a[2] = 'w'
```

```
Olsztyn
z
```

```
1 a = "Olsztyn"
2 b = "Gdańsk"
3 print(a + b)
4 print(a * 2)
5 print(2 * a)
```

```
OlsztynGdańsk
OlsztynOlsztyn
OlsztynOlsztyn
```

Specjalne funkcje

- `chr()` zamienia liczbę całkowitą na znak
- `ord()` zamienia znak na liczbę całkowitą odpowiadającą pozycji w tabeli znaków
- `len()` - długość napisu
- `str()` - rzutuje argument na napis

Porządek leksykograficzny

Mądra definicja z wikipedii:

Relację leksykograficzną $\preccurlyeq$ między ciągami $\alpha, \beta \in X^*$ ustala się następująco:

- jeśli istnieje wskaźnik j taki, że $\alpha(j) \neq \beta(j)$, to znajdujemy najmniejszy i o tej własności. Wówczas
  - $\alpha \preccurlyeq \beta$ gdy $\alpha(i) \preccurlyeq \beta(i)$ lub $\beta \preccurlyeq \alpha$ gdy $\beta(i) \preccurlyeq \alpha(i)$ (tzn. relacja między ciągami jest zgodna z relacją między odpowiednimi elementami)
- jeśli taki j nie istnieje, to
  - jeśli oba są skończone i tej samej długości, to $\alpha = \beta$
  - jeśli oba ciągi są nieskończone, to $\alpha = \beta$
  - jeśli są różnej długość np. β jest dłuższy od α (w szczególności β może być nieskończony), to $\alpha \prec \beta$

Przykłady:

```
1 print("A" < "a")
2 print("Abc" < "aTw")
3 print("vccx" < "123")
4 print("ABC" < "AB")
5 print("AB" < "ABC")
```

```
True
True
False
False
True
```

Fomatowanie napisów

- trzy różne konwencje
- niektóre rzeczy nie działają w każdej wersji 3.x
- warto zastanowić się czy warto używać tych konstrukcji? czasem może lepiej skorzystać z funkcji print?

styl `printf`

Zaczerpnięty z języka C - stare.

https://docs.python.org/3.10/library/stdtypes.html#old-string-formatting

```
1  a = "abc"
2  str = "a to %s" % a
3  print(str)
4  b = 4
5  c = 5
6  str2 = "%d + %d = %d" % (b, c, b + c)
7  print(str2)
```

```
a to abc
4 + 5 = 9
```

Dodatkowe:

https://gist.github.com/pjastr/02d01dba3d5f5c3e60ed74cb32c913
https://gist.github.com/pjastr/cbe8418eb4798b92d7fcba4f48d328
https://gist.github.com/pjastr/e7d5fcbebd578c1df122d307e00517
https://gist.github.com/pjastr/00c1df223918f975d678d8455b4f5b

styl format

https://docs.python.org/3.10/library/string.html#formatstrings

```
1 a = "abc"
2 str = "a to {}".format(a)
3 print(str)
4 b = 4.2
5 c = 5
6 str2 = "{0} + {1} = {2}".format(b, c, b + c)
7 print(str2)
```

```
a to abc
4.2 + 5 = 9.2
```

```
1 b = 4.2
2 c = 5
3 str2 = "{0:f} + {1:d} = {2:e}".format(b, c, b + c)
4 print(str2)
```

```
4.200000 + 5 = 9.200000e+00
```

https://pyformat.info/

https://gist.github.com/pjastr/27fe6c13cd8bcba63be561a05af030a0

https://gist.github.com/pjastr/d40fe6eaf21a9595bcf6b43e7b020fbd

https://gist.github.com/pjastr/8a630b4a575e6a389a7dc3c5e8dc65a0

https://gist.github.com/pjastr/d42d937c7b00b80b5dfe309b4ac0e854

https://gist.github.com/pjastr/dbc9a0c1e8bf612b68e0bc7789daf53b

https://gist.github.com/pjastr/adfd7371dcbe4e4034bfb12dcfe30129

https://gist.github.com/pjastr/2980fb68a484dcb5ff595774eec4195c

Dodatkowe przykłady:

https://docs.python.org/3.9/library/string.html#format-examples
https://gist.github.com/pjastr/d42d937c7b00b80b5dfe309b4ac0e8
https://gist.github.com/pjastr/90f1accf7f54d8c74ac036d59a24a9d
https://gist.github.com/pjastr/adfd7371dcbe4e4034bfb12dcfe3012
https://gist.github.com/pjastr/2980fb68a484dcb5ff595774eec4195

f-Strings

https://docs.python.org/3.10/reference/lexical_analysis.html#f-strings

```
1 a = "abc"
2 str = f"a to {a}"
3 print(str)
4 b = 4.2
5 c = 5
6 str2 = f"{b} + {c} = {b+c}"
7 print(str2)
```

```
a to abc
4.2 + 5 = 9.2
```

```
1 b = 4.2
2 c = 5
3 str2 = f"{b:f} + {c:d} = {b+c:e}"
4 print(str2)
```

```
4.200000 + 5 = 9.200000e+00
```

Dodatkowe

- podział stałych https://docs.python.org/3.10/library/string.html?highlight=string#module-string
- funkcje wbudowane dot. napisów https://docs.python.org/3.10/library/stdtypes.html#string-methods

Funkcje

Funkcje

```
1 def functionname( parameters ):
2    "function_docstring"
3    function_suite
4    return [expression]
```

```
1  def printme(str):
2      """Funkcja wyświetlająca string"""
3      print(str)
4      return
5
6
7  printme("abc")
8  print(printme.__doc__)
```

```
abc
Funkcja wyświetlająca string
```

Przekazywanie przez referencję

```
 1 def changeme(lista):
 2     print("Przed zmianą: ", lista)
 3     lista[2] = 50
 4     print("Po zmianie: ", lista)
 5     return
 6
 7
 8 mylist = [10, 20, 30]
 9 changeme(mylist)
10 print("Poza funkcją: ", mylist)
```

```
Przed zmianą:  [10, 20, 30]
Po zmianie:  [10, 20, 50]
Poza funkcją:  [10, 20, 50]
```

```
1  def changeme(lista):
2      lista = [2, 3, 4]
3      print("Wewnątrz funkcji: ", lista)
4      return
5
6
7  lista = [10, 20, 30]
8  changeme(lista)
9  print("Poza funkcją: ", lista)
```

```
Wewnątrz funkcji:  [2, 3, 4]
Poza funkcją:  [10, 20, 30]
```

```
1  def changeme():
2      global lista
3      lista = [2, 3, 4]
4      print("Wewnątrz funkcji: ", lista)
5      return
6
7
8  changeme()
9  print("Poza funkcją: ", lista)
```

```
Wewnątrz funkcji:  [2, 3, 4]
Poza funkcją:  [2, 3, 4]
```

Obowiązkowy argument (ang. positional argument)

```
1 def printme(str):
2     print(str)
3     return
4
5
6 printme()
```

```
## TypeError: printme() missing 1 required
positional argument: 'str'
```

Keyword argument

```
1 def kwadrat(a):
2     return a*a
3
4
5 print(kwadrat(a=4))
```

```
16
```

Domyślny argument

```
1 def sumsub(a, b, c=0, d=0):
2     return a - b + c - d
3
4
5 print(sumsub(12, 4))
6 print(sumsub(3, 4, 5, 7))
```

```
8
-3
```

```python
def srednia(first, *values):
    return (first + sum(values)) / (1 + len(values))


print(srednia(2, 3, 4, 6))
print(srednia(45))
```

```
3.75
45.0
```

```python
def f(**kwargs):
    print(kwargs)


f()
f(pl="Polish", en="English")
```

```
{}
{'pl': 'Polish', 'en': 'English'}
```

Inne symbole

- symbol / oznacza, że wcześniejsze argumenty są pozycyjne
- symbol * oznacza, że późniejszej argumenty są typu keyword

```
1 def f(a, b, /, c, d, *, e, f):
2     print(a, b, c, d, e, f)
3
4
5 f(10, 20, 30, d=40, e=50, f=60)
```

```
10 20 30 40 50 60
```

Funkcje matematyczne

Link do dokumentacji

https://docs.python.org/3/library/math.html

```
1 import math
2
3 a=0
4 b=math.sin(2*math.pi)
5 print(b)
6 print(math.isclose(a,b, rel_tol=1e-09, abs_tol=1e-09))
```

```
-2.4492935982947064e-16
True
```

Wyrażenia lambda

Skąd nazwa? https://pl.wikipedia.org/wiki/Rachunek_lambda

Wyrażenie

```
1 def identity(x):
2     return x
```

zapisujemy jako

```
lambda x: x
```

Jeśli uruchomimy to w konsoli, to możemy wywołać

```
1 _(5)
```

To tzw. przykład funkcji anonimowej.

By uruchomić to też w skrypcie, to potrzebujemy argumentu:

```
1 print((lambda x: x + 1)(2))
```

lub

```
1 add_one = lambda x: x + 1
2 print(add_one(2))
```

Możemy też tworzyć złożenia funkcji (funkcje wyższych rzędów):

```
1 high_ord_func = lambda x, func: x + func(x)
2 print(high_ord_func(2, lambda x: x * x))
3 print(high_ord_func(2, lambda x: x + 3))
```

```
6
7
```

Możemy również skorzystać z różnych opcji argumentów jak dla funkcji:

```
print((lambda x, y, z: x + y + z)(1, 2, 3))
print((lambda x, y, z=3: x + y + z)(1, 2))
print((lambda x, y, z=3: x + y + z)(1, y=2))
print((lambda *args: sum(args))(1,2,3))
print((lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3))
print((lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3))
```

```
6
6
6
6
6
6
```

Typ

```
1 string = 'abc'
2 print(lambda string: string)
```

```
<function <lambda> at 0x000001BCDAEAE7A0>
```

Różnice względem zwykłej funkcji:

```
1 def cube(y):
2     print(f"Finding cube of number:{y}")
3     return y * y * y
4
5 lambda_cube = lambda num: num ** 3
6 print("invoking function defined with def keyword:")
7 print(cube(30))
8 print("invoking lambda function:", lambda_cube(30))
```

```
invoking function defined with def keyword:
Finding cube of number:30
27000
invoking lambda function: 27000
```

Przykłady praktyczne

```
1 r = lambda a: a + 15
2 print(r(10))
3 r = lambda x, y: x * y
4 print(r(12, 4))
```

```
25
48
```

```python
subject_marks = [('English', 88), ('Science', 90), ('Maths', 97), ('Social
print("Original list of tuples:")
print(subject_marks)
subject_marks.sort(key=lambda x: x[1])
print("\nSorting the List of Tuples:")
print(subject_marks)
```

```
Original list of tuples:
[('English', 88), ('Science', 90), ('Maths', 97), ('Social sciences', 82)]

Sorting the List of Tuples:
[('Social sciences', 82), ('English', 88), ('Science', 90), ('Maths', 97)]
```

```python
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print("Original list of integers:")
print(nums)
print("\nEven numbers from the said list:")
even_nums = list(filter(lambda x: x % 2 == 0, nums))
print(even_nums)
print("\nOdd numbers from the said list:")
odd_nums = list(filter(lambda x: x % 2 != 0, nums))
print(odd_nums)
```

```
Original list of integers:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Even numbers from the said list:
[2, 4, 6, 8, 10]

Odd numbers from the said list:
[1, 3, 5, 7, 9]
```

```
array_nums1 = [1, 2, 3, 5, 7, 8, 9, 10]
array_nums2 = [1, 2, 4, 8, 9]
print("Original arrays:")
print(array_nums1)
print(array_nums2)
result = list(filter(lambda x: x in array_nums1, array_nums2))
print("\nIntersection of the said arrays: ", result)
```

```
Original arrays:
[1, 2, 3, 5, 7, 8, 9, 10]
[1, 2, 4, 8, 9]

Intersection of the said arrays:  [1, 2, 8, 9]
```

“Obiektowość”

Programowanie obiektowe w Pythonie

```
class Employee:
    """Common base class for all employees"""
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print("Total Employee %d" % Employee.empCount)

    def displayEmployee(self):
        print("Name : ", self.name, ", Salary: ",
              self.salary)


emp1 = Employee("John", 2000)
```

```
Name :  John , Salary:  2000
Name :  Anna , Salary:  5000
```

Bibliografia

Bibliografia

- https://pl.wikipedia.org/wiki/Python, dostęp online 20.02.2023.
- https://bulldogjob.pl/news/264-java-php-ruby-jak-wlasciwie-wymawiac-nazwy-technologii. dostęp online 12.02.2019.
- https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html, dostęp online 14.02.2019.
- K. Ropiak, Wprowadzenie do języka Python, http://wmii.uwm.edu.pl/\~kropiak/wd/Wprowadzenie%20do%20j%C4%99zyka%20Python.p dostęp online 14.02.2019.
- B. Slatkin, Efektywny Python. 59 sposobów na lepszy kod, Helion 2015.
- https://www.python.org/dev/peps/pep-0008/, dostęp online 20.02.2023.

Bibliografia - cd2

- https://www.flynerd.pl/2017/05/python-4-typy-i-zmienne.html, dostęp online 14.02.2019.
- http://pytolearn.csd.auth.gr/p0-py/01/print.html, dostęp online 15.02.2019.
- https://www.tutorialspoint.com/python3/python_lists.htm, dostęp online 17.02.2019.
- https://realpython.com/python-data-types/, dostęp online 5.01.2022
- https://www.w3schools.com/python/python_variables.asp, dostęp online 5.01.2022
- https://www.w3schools.com/python/python_variables_multiple.asp, dostęp online 5.01.2022

Bibliografia - cd3

- https://realpython.com/python-print/, dostęp online 5.01.2022
- https://www.programiz.com/python-programming/operators, dostęp online 5.01.2022
- https://realpython.com/python-conditional-statements/, dostęp online 5.01.2022
- https://realpython.com/python-for-loop/, dostęp online 5.01.2022
- https://realpython.com/python-while-loop/, dostęp online 5.01.2022
- https://towardsdatascience.com/a-guide-to-python-comprehensions-4d16af68c97e, dostęp online 20.02.2023.