Wprowadzenie do języka Python - Wykład 2

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

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

Przykład:

```
for i in range(5):
    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>
```

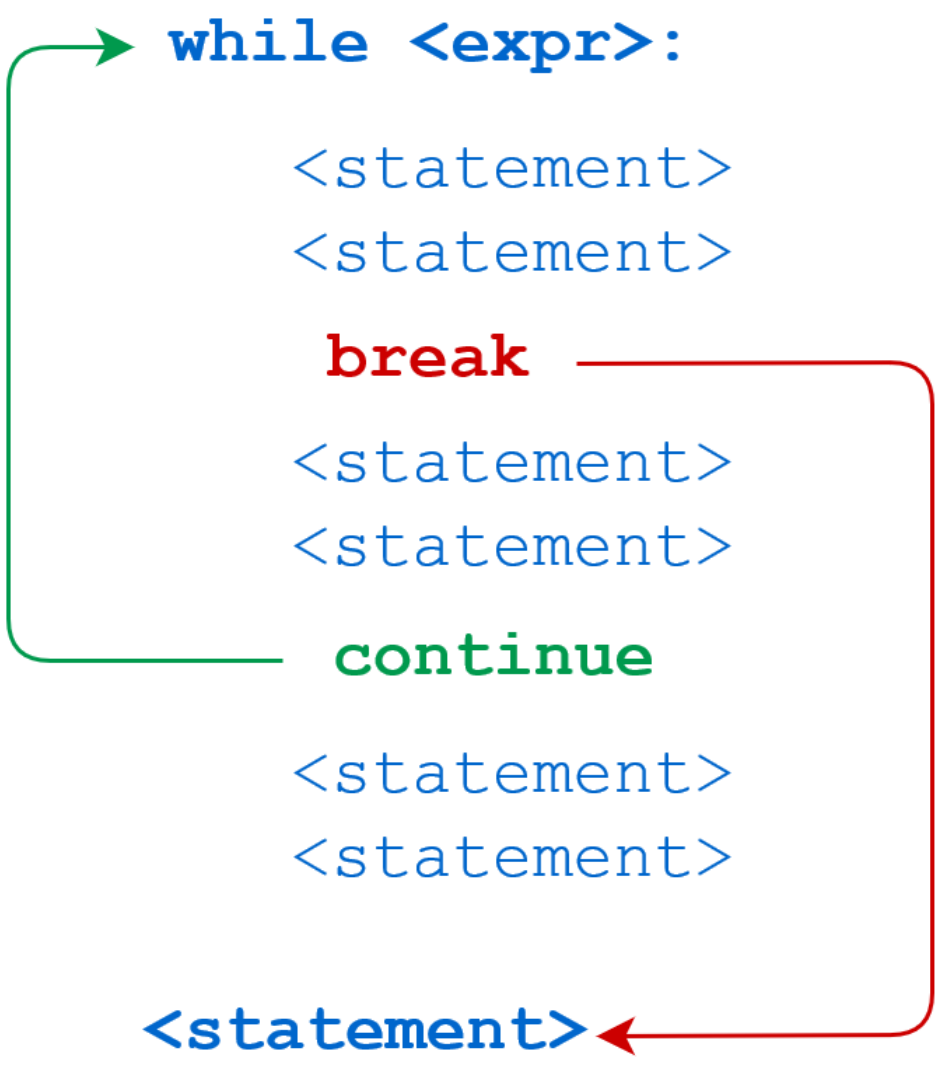
inna opcja:

```
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
```

Typowanie a pętle

Na ten moment nie jest obowiązkowe

https://peps.python.org/pep-0526/#id11

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

```
0
1
2
3
4
```

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
```

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]
```

Listy a typowanie

https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html

Wersja “stara” do Pythona 3.8:

```
1  from typing import List
2
3  x: List[int] = [1]
```

Wersja “nowa” do Pythona 3.9 (brak modułu) i 3.10 (operator |):

```
1  x: list[int | str] = [3, 5, "test", "fun"]
```

Słowo kluczowe `Optional`

```python
from typing import Optional

x: Optional[int] = 5
print(x)
x = None
print(x)
x = 7
print(x)
a: list[Optional[int]] = [5, None]
print(x)
```

```
5
None
7
7
```

Słowo kluczowe Any

```python
from typing import Any

x: Any = 5
print(x)
x = None
print(x)
x = "abc"
print(x)
a: list[Any] = [5, None, "abc"]
print(x)
```

```
5
None
abc
abc
```

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
```

```python
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)
```

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

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

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}
```

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

```
[3, 4, 5]
<generator object <genexpr> at 0x0000023A13C0D460>
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]
```

Bibliografia

- 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.