Faculty of Geodesy and Cartography
WARSAW UNIVERSITY OF TECHNOLOGY

Informatyka Geodezyjna - wykłady/ćwiczenia, rok akad. 2020-2021

Wyk. 5: Python - pętla for

Kinga Węzka
kinga.wezka@pw.edu.pl
Katedra Geodezji i Astronomii Geodezyjnej

Warsaw University of Technology

Plan

1. Sterowanie wykonywaniem programu
   - Pętle w Pythonie
2. Pętla for
   - Obiekty iterowalne (iteratory)
3. Przykłady użycia pętli for
   - Przykłady użycia pętli for - iteracja po łańcuchu znaków
   - Przykłady użycia pętli for - iteracja po listach
   - Przykłady użycia pętli for - iteracja po słownikach
   - Przykłady użycia pętli for - zip - przechodzenie równoległe
   - Przykłady użycia pętli for - enumerate - generowanie wartości przesunięcia i elementów
4. Wynik wykonania pętli for – zasięg wyniku działania pętli
5. Pętle zagnieżdżone (ang. nested loops)
6. Listy i Słowniki składane (ang. list and dict comprehension)
7. Wyrażenie else dla pętli for
8. Instrukcje kontrolne w pętli for
9. Podsumowanie: funkcje często używane z iteratorami

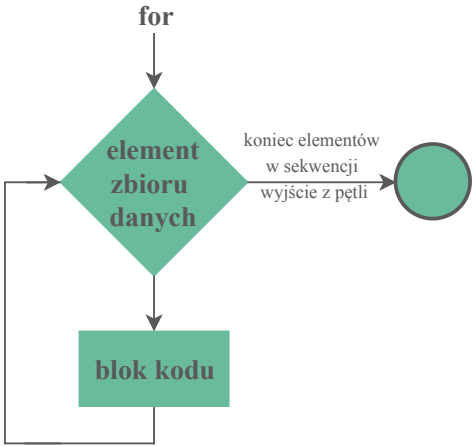
- Instrukcje warunkowe: **if/elif/else**
- Instrukcje kontrolne: **break**, **continue** i **pass**
- Pętle: **for while**

W Pythonie używane są dwa rodzaje pętli: for i while

- `for` : "przebiega" przez podany zbiór danych, element-po-elemencie. Pętli używamy jeżeli wiemy ile iteracji chcemy wykonać - znamy liczbę iteracji (nawet jeśli możesz to wcześniej zatrzymać)
- `while`: wykonuje się dopóki pewien warunek logiczny jest spełniony. Pętli używamy jeżeli nie wiemy kiedy się ona zakończy (nie znamy liczby iteracji) natomiast znamy jakiś specyficzny warunek który powinien zatrzymać iteracje.

Obie pętle są używane w celu przetwarzania powtarzającej się sekcji kodu. W przeciwieństwie do pętli **for** pętla **while** nie jest uruchamiana n-razy, ale dopóki zdefiniowany warunek nie będzie już spełniony. Jeśli warunek jest na początku (w pierwszym przejściu) fałszywy, blok kodu w pętli nie zostanie w ogóle wykonany.

Pętla `for` "przebiega" przez podany zbiór danych, który jest **obiektem iterowalnym** lub **iteratorem** (np. łańcuch znaków, lista, słownik itp.) oraz własnych typach obiektów, które można tworzyć za pomocą klas (więcej na wykładach na temat programowanie obiektowe)




```
for <element> in <obiekt iterowalny>:
    blok kodu: wykonanie dla elementu
```

In[1]:

```
for element in [1, 2, 3]:
    wynik = element + 5
    print(wynik)
```

Out[1]:

```
6
7
8
```

- Kiedy mamy kolekcję typów danych, to naturalne jest, że chcemy przeprowadzać iteracje przechowywanych tam danych.
- Typ danych **umożliwiający iterację** to taki, który może zwrócić pojedynczo każdy jego element. Każdy obiekt posiadający metodę `__iter__()` lub każda sekwencja (na przykład obiekt posiadający metodę `__getitem__()` pobierająca argumenty w postaci liczb całkowitych, począwszy od zera) pozwala na iterację i jest **iteratorem** (Summerfield, 2010, p.155).
- Kiedy używamy pętli `for element in iterator`, Python w rzeczywistości wywołuje funkcję `iter(iteracja)` w celu pobrania iteratora. Następnie w trakcie wykonywania każdej pętli iteracji zostaje wywołana metoda iteratora – `__next__()` , której celem jest pobranie kolejnego elementu. Po zgłoszeniu wyjątku `StopIteration` wykonywanie pętli zostaje przerwane.

- Pętla po elementach łańcucha znaków (typ string):

In[2]:

```
s = "abc"
for c in s:
    print(c)
```

Out:

```
a
b
c
```

- Pętla po elementach łańcucha znaków z użyciem indeksów:

In[3]:

```
s = "abc"
for i in range(0, len(s)):
    print(s[i])
```

Out:

```
a
b
c
```

- Iteracja po elementach listy:

In[4]:

```
1 L = [1, 2, 3, 4]
2 for element in L:
3     print(element)
```

Out:

```
1 1
2 2
3 3
4 4
```

- Modyfikacja elementów listy :

In[5]:

```
1 L = [1, 2, 3, 4, 5]
2 for i in range(len(L)):
3     L[i] += 1
4 print(L)
```

Out:

```
1 [2, 3, 4, 5, 6]
```

- Iteracja po kluczach słownika:

In[6]:
```
D = {'a':1, 'b':2}
for key in D.keys():
    print('>', key)
```
Out:
```
> a
> b
```

- Iteracja po wartościach słownika:

In[7]:
```
D = {'a':1, 'b':2}
for val in D.values():
    print('>', val)
```
Out:
```
> 1
> 2
```

- Iteracja po kluczach i wartościach słownika:

In[8]:
```
D = {'a':1, 'b':2}
for key, val in D.items():
    print(key, '>', val)
```
Out:
```
a > 1
b > 2
```

- Iteracja po kluczach i odwołanie do wartości:

In[9]:
```
D = {'a':1, 'b':2}
for key in D:
    print(key, '>', D[key])
```
Out:
```
a > 1
b > 2
```

https://realpython.com/iterate-through-dictionary-python/?fbclid=

Funkcja `zip()` pozwala na wykorzystanie pętli for do równoległego przejścia większej liczby sekwencji.

In[10]:

```
A = [1, 2, 3, 4]
B = [8, 8, 8, 8]
C = [1, 1, 1, 1]
for a, b, c in zip(A, B, C):
    wynik = a + b + c
    print(a,b,c, '=', wynik)
```

Out:

```
1 8 1 = 10
2 8 1 = 11
3 8 1 = 12
4 8 1 = 13
```

In[11]:

```
A = [1, 2, 3]
B = [8, 8, 8, 8, 8, 8]
C = [1, 1, 1, 1, 1]
for a, b, c in zip(A, B, C):
    wynik = a + b + c
    print(a,b,c, '=', wynik)
```

Out:

```
1 8 1 = 10
2 8 1 = 11
3 8 1 = 12
```

Funkcja `zip(i1, i2, ..., iN)` - zwraca iterator krotek, używając iteratorów od i1 do iN

- Tworzenie słowników za pomocą funkcji zip

In[12]:

```
keys = ['mielonka', 'jajka', 'tost']
vals = [1, 3, 5]
D2 = {}
for (k, v) in zip(keys, vals):
    D2[k] = v
print(D2)
```

Out:

```
{'mielonka': 1, 'jajka': 3, 'tost': 5}
```

Funkcja `enumerate` zwraca obiekt *generatora* — rodzaj obiektu obsługujący protokół iteracji (więcej na kolejnych wykładach). W skrócie, obsługuje on metodę `__next__` wywoływaną za pomocą funkcji wbudowanej `next` i zwracającą za każdym przejściem pętli krotkę `(indeks, wartość)`.

Funkcja `enumerate(i, start)` używana jest w pętlach `for...in ..` w celu dostarczenia sekwencji `(indeks,element)` krotek o indeksie startującym od zera lub od wartości `start`.

In[13]:

```
1 L = [3, 4, 5 ]
2 for (i, e) in enumerate(L):
3     print(i, e)
```

Out:

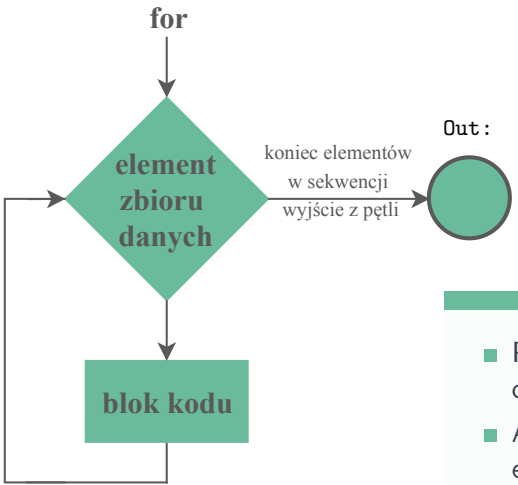
```
1 0 3
2 1 4
3 2 5
```

In[14]:

```
1 L = [3, 4, 5 ]
2 for (i, e) in enumerate(L, 10):
3     print(i, e)
```

Out:

```
1 10 3
2 11 4
3 12 5
```



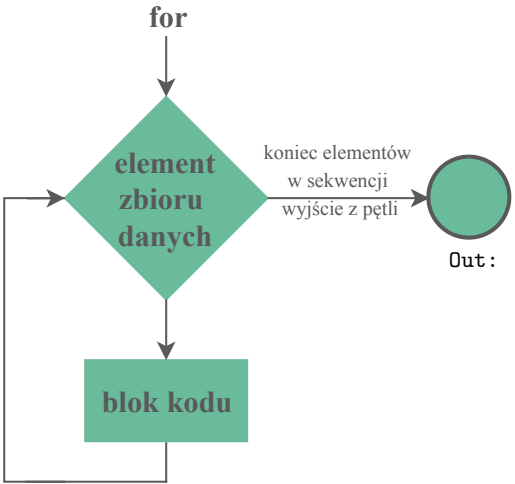
In[15]:

```
1 for element in [1, 2, 3]:
2     wynik = element + 5
3     print('wynik w pętli =', wynik)
4 print('wynik poza pętlą =', wynik)
```

Out:

```
1 wynik w pętli = 6
2 wynik w pętli = 7
3 wynik w pętli = 8
4 wynik poza pętlą = 8
```

- Po wyjściu z pętli mamy dostęp jedynie do ostatniego przetworzonego elementu: `wynik = 8`.
- Aby mieć dostęp do wyników przetwarzania każdego elementu, należy je "zebrać" do nowej kolekcji!

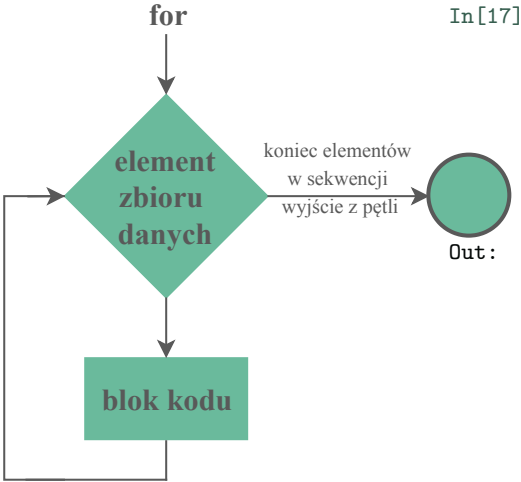


In[16]:

```
suma = 0
for element in [1, 2, 3, 4]:
    wynik = element + 5
    suma += wynik # suma=suma+wynik
    print('w pętli', suma)
print('suma skumulowana:', wynik)
```

Out:

```
w pętli 6
w pętli 13
w pętli 21
w pętli 30
suma skumulowana: 9
```

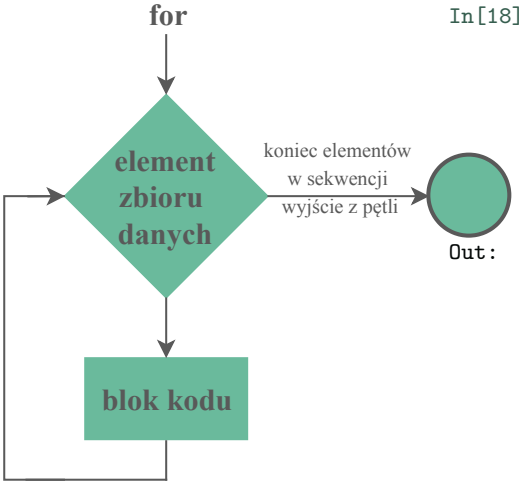


In[17]:

```
A = [] # utworzenie pustej listy
for element in [1, 2, 3, 4]:
    wynik = element + 5
    A.append(wynik)
print(A)
```

Out:

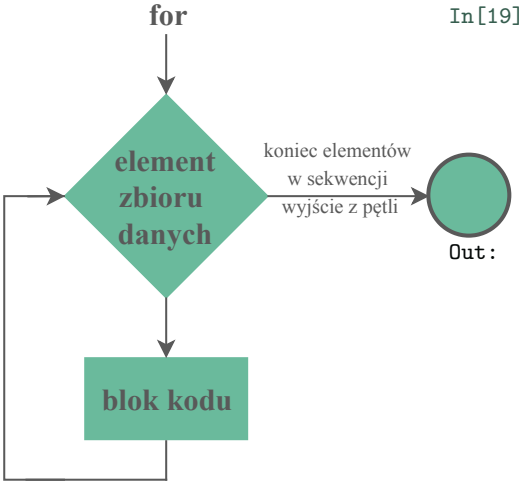
```
[6, 7, 8, 9]
```



In[18]:

```
A = [] # utworzenie pustej listy
for element in [1, 2, 3, 4]:
    wynik = element + 5
    A.extend([wynik]) #iterowalny!
print(A)
```

```
[6, 7, 8, 9]
```

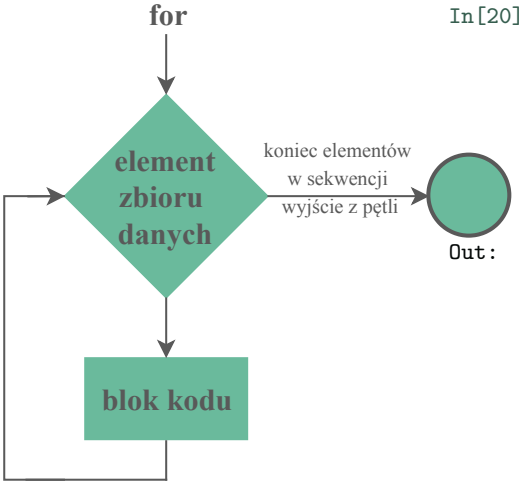


In[19]:

```
1 A = [] # utworzenie pustej listy
2 for element in [1, 2, 3, 4]:
3     wynik = element + 5
4     A.append([element, wynik])
5 print(A)
```

Out:

```
1 [[1, 6], [2, 7], [3, 8], [4, 9]]
```

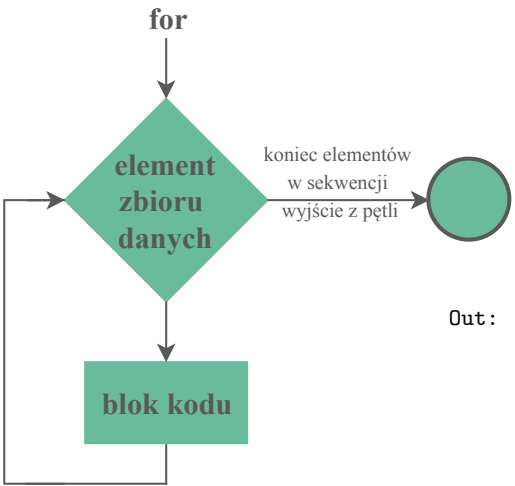


In[20]:

```
A = [] # utworzenie pustej listy
for element in [1, 2, 3, 4]:
    wynik = element + 5
    A.extend([element, wynik])
print(A)
```

Out:

```
[1, 6, 2, 7, 3, 8, 4, 9]
```

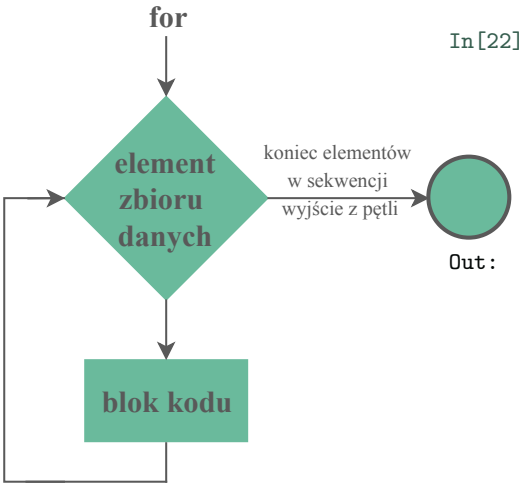


In[21]:

```
A = 4*['nan'] # inicjalizacja listy
print('przed:', A)
i = 0 # inicjalizacja indeksu
for element in [1, 2, 3, 4]:
    wynik = element + 5
    print('idx', i, 'wynik', wynik)
    A[i] = wynik
    i+=1  # aktualizacja indeksu
print('po :', A)
```

Out:

```
przed: ['nan', 'nan', 'nan', 'nan']
idx 0 wynik 6
idx 1 wynik 7
idx 2 wynik 8
idx 3 wynik 9
po  : [6, 7, 8, 9]
```

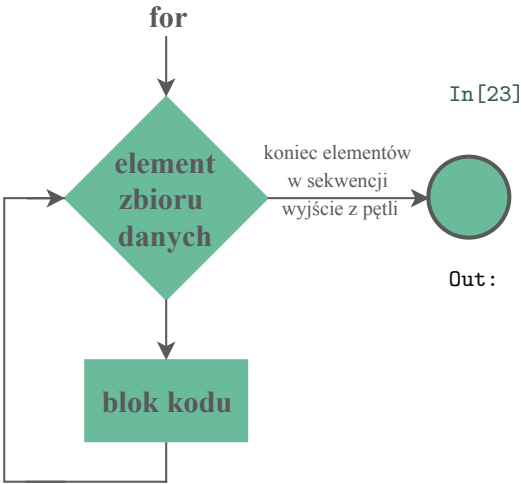


In[22]:

```
A = set() # pusty zbioru
for element in set({1,2,3,4}):
    wynik = element + 5
    A.update([wynik]) #iterowalny!
print(A)
```

Out:

```
{8, 9, 6, 7}
```

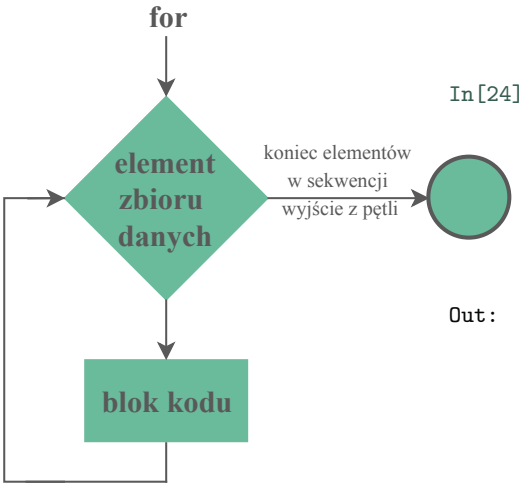


In[23]:

```
1 A = '' # pusty łańcuch znaków
2 for element in 'łańcuch znaków':
3     A += ' ' + element
4 print(A)
```

Out:

```
1 ł a ń c u c h  z n a k ów
```

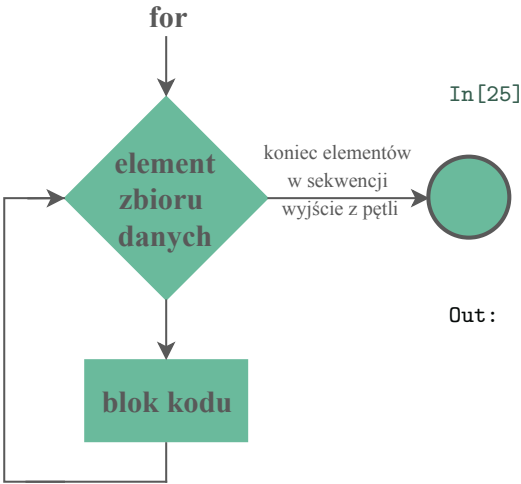


In[24]:

```
A = {} # pusty słownik
for i in range(0,5):
    A.setdefault('result', [])
    A['result'].append(i)
print(A)
```

Out:

```
{'result': [0, 1, 2, 3, 4]}
```



In[25]:

```
A = {} # pusty słownik
for i in range(4):
    result = 'key' + str(i)
    A[result] = i
print(A)
```

Out:

```
{'key0': 0, 'key1': 1, 'key2': 2,
    'key3': 3}
```

Pętle zagnieżdżone (ang. *nested loops*), tzw. pętle w pętli), wykorzystywane są do przebiegu po kolekcjach wielopoziomowych np. listy list, słownika którego wartości są sekwencją etc.

In[26]:

```
A = [['a', 'b'], [1, 2]]
for lista in A:
    print('lista', lista)
    for e in lista:
        print(e)
```

Out:

```
lista ['a', 'b']
a
b
lista [1, 2]
1
2
```

In[27]:

```
A = {'a':[1,2], 'b': [11,22]}
for key, val in A.items():
    print('key, val', key, val)
    for e in val:
        print(e)
```

Out:

```
key, val a [1, 2]
1
2
key, val b [11, 22]
11
22
```

```
1 lista = [działanie for element in iterator]
```

In[28]:

```
1 lista = []
2 for e in range(4):
3     wynik = e+3
4     lista.append(wynik)
5 print(lista)
```

Out[28]:

```
1 [3, 4, 5, 6]
```

In[29]:

```
1 lista = [e + 3 for e in range(4)]
2 print(lista)
```

Out:

```
1 [3, 4, 5, 6]
```

Zapis w komórce In[28] jest równoważny z zapisem w komórce In[29].

```
lista = [działanie for element in iteracja if warunek]
```

In[30]:

```
lista = []
for e in range(4):
    if e >= 2:
        wynik = e+3
        lista.append(wynik)
print(lista)
```

Out:

```
[5, 6]
```

In[31]:

```
lista = [e + 3 for e in range(4) if e >= 2]
print(lista)
```

Out:

```
[5, 6]
```

```
s = {wyrażenia_klucz:wyrażenie_wartość for klucz, wartość in iteracja}
```

In[32]:

```
import os
file_size = {}
for nazwa in os.listdir("."):
    file_size[nazwa] = os.path.getsize(nazwa) # w bajtach
print(file_size)
```

Out:

```
{'zadania': 4096, 'numbers.py': 1350, 'utc2sow_leapsec.py': 825,
    'slad_stosu.py': 237, 'plik1.txt': 17}
```

In[33]:

```
f_s = {nazwa:os.path.getsize(nazwa) for nazwa in os.listdir(".")}
```

Out:

```
{'zadania': 4096, 'numbers.py': 1350, 'utc2sow_leapsec.py': 825,
    'slad_stosu.py': 237, 'plik1.txt': 17}
```

```
s = {wyraż_klucz:wyraż_wart. for klucz, wartosc in iteracja if warunek}
```

In[34]:

```
file_size = {}
for nazwa in os.listdir("."):
    if os.path.isfile(nazwa):
        file_size[nazwa] = os.path.getsize(nazwa) # w bajtach
print(file_size)
```

Out:

```
{'types_numbers.py': 1350, 'utc2sow_leapsec.py': 825,
    'slad_stosu.py': 237, 'plik1.txt': 17}
```

In[35]:

```
f_s = {nazwa:os.path.getsize(nazwa) for nazwa in os.listdir(".")
    if os.path.isfile(nazwa)}
```

Out:

```
{'types_numbers.py': 1350, 'utc2sow_leapsec.py': 825,
    'slad_stosu.py': 237, 'plik1.txt': 17}
```

Po pętli `for` można umieścić instrukcje `else`. Kod pod instrukcją `else` wykonywany jest tylko gdy pętla jest "wyczerpana". Instrukcja `else` nie wykona się gdy przerwiemy przejścia instrukcją `break` lub gdy zostanie zgłoszony wyjątek (ang. *exception raised*).

In[36]:

```
for i in range(4):
    print(i)
else:
    print("zrobione")
```

Out:

```
0
1
2
3
zrobione
```

In[37]:

```
for i in range(4):
    print(i)
    if i == 2:
        break
else:
    print("zrobione")
```

Out:

```
0
1
2
```

continue

Kończy przebieg aktualnej iteracji pętli i rozpoczyna kolejną iterację.

In[38]:

```
for i in range(10):
    if i %2 != 0:
        continue
    print('even', i)
```

Out:

```
even 0
even 2
even 4
even 6
even 8
```

break

Powoduje zakończenie pętli, kod po tej instrukcji w pętli nie zostanie wykonany.

In[39]:

```
for i in range(10):
    if i %2 != 0:
        break
    print('even', i)
```

Out:

```
even 0
```

In[40]:

```
1 def dodawanie(x,y):
2     pass
```

Out[40]:

```
1 ...
```

- Instrukcja `break` powoduje natychmiastowe wyjście z pętli (znajdziemy się poniżej całej instrukcji pętli), natomiast `continue` przeskakuje z powrotem na górę pętli (znajdziemy się tuż przed kolejnym elementem pobieranym w `for`).
- Część `else` w pętlach `for` zostanie wykonana raz, kiedy pętla się kończy — o ile kończy się normalnie, bez trafienia na instrukcję `break`. Instrukcja `break` powoduje natychmiastowe wyjście z pętli i pominięcie części `else` (o ile jest ona w ogóle obecna).
- Funkcję wbudowaną `range` można wykorzystać w pętli `for` w celu zaimplementowania ustalonej liczby powtórzeń, przejścia po wartościach przesunięć zamiast po elementach znajdujących się na tych pozycjach, pominięcia kolejnych elementów w miarę przechodzenia, a także modyfikacji listy w trakcie przechodzenia jej. Żadna z tych ról nie wymaga `range` i w większości przypadków istnieją alternatywy — przejście samych elementów, wycinki z trzema wartościami granicznymi, a także listy składane są obecnie nieraz lepszymi rozwiązaniami (pomimo naturalnych upodobań byłych programistów języka C, którzy chcą wszystko zliczać) (Lutz, 2011).
- Słowniki i listy składane są elementami paradygmatu programowania funkcyjnego oferowanego przez Python.

Na podstawie (Summerfield, 2010, tab. 3.4):

- `a + b` – zwraca sekwencje, która jest konkatenacją sekwencji s i t
- `s * n` – zwraca sekwencje, która wielokrotnością (n) sekwencji s
- `x in i` – zwraca wartość True, jeśli element x znajduje się w iteracji. Do odwrócenia sprawdzenia należy użyć operatora `not in`
- `all(i)` – zwraca wartość `True`, jeśli każdy element iteracii i przyjmuje wartość `True`.
- `any(i)` – zwraca wartość `True`, jeśli dowolny element iteracii i przyjmuje wartość `True`
- `enumerate(i, start)` – funkcja używana w pętlach `for...in ..` w celu dostarczenia sekwencji (indeks, element) krotek o indeksie rozpoczynającym się od zera lub wartości start.
- `len(x)` – zwraca wielkość x. Jeżeli x jest kolekcją, wówczas oznacza to liczbę elementów. Jeżeli x jest ciągiem tekstowym, oznacza to liczbę znaków

- `max(i, klucz)` – zwraca największy element w iteracji i lub element z największą wartością `klucz(element)`, jeśli podano `klucz`
- `min(i, klucz)` – zwraca najmniejszy element w iteracji i lub element z największą wartością `klucz(element)`, jeśli podano `klucz`
- `range(poczatek, koniec, interwal)` – zwraca liczbę całkowitą iteratora. Jeżeli podano jeden argument (koniec), iterator bedzie w przedziale od 0 do koniec 1. W przypadku dwóch argumentów (poczqtek, koniec) iterator będzie w przedziale od poczqtek do koniec. Po podaniu trzech argumentów iterator bedzie w przedziale od początek do koniec w krokach o wartości krok.
- `reversed(i)` – zwraca iterator, który zwraca elementy iteratora i w odwrotnej kolejności.
- `sorted(i, key, reverse)` - zwraca listę elementów iteratora i w kolejności sortowania. Argument key służy do podania wzorca DSU (Decorate, Sort, Undecorate). Jeżeli argument reverse ma wartość `True`, sortowanie będzie przeprowadzone odwrotnie

- `sum(i, start)` - zwraca sumę elementów w iteracji i plus początek (którego wartość domyślna wynosi 0). Iterator i nie może zawierać ciągów tekstowych.
- `zip(i1, i2, ..., iN)` - zwraca iterator krotek, używając iteratorów od i1 do iN.

M. Lutz. *Python. Wprowadzenie*. Helion, 2011.

M. Summerfield. *Python 3. Kompletne wprowadzenie do programowania. Wydanie II*. 2010.

Dziękuje za uwagę

Kinga Węzka kinga.wezka@pw.edu.pl