Politechnika Wrocławska

Programowanie Procduralne
Wykład #5(W#12) – Fizyka

Janusz Andrzejewski

Wstęp

- Generatory
  - Comprehensions
    - Listy składane
    - Zbiory składane
    - Słowniki składane
  - Wyrażenia generatorowe
  - Funkcje generatorowe
- Funkcje wyższego rzędu
  - min() oraz max()
  - sum()

Plan czyli czego się będziemy uczyli?

- W#1 – Podsumowanie
- – Typy mutowalne oraz niemutowalne a przekazywanie argumentów do funkcji.
- – Zasięg zmiennych — zmienne lokalne, nielokalne oraz globalne.
- – Argumenty funkcji: pozycyjne, nazwane, domyślne, rozpakowywanie, słownikowe
- – Funkcje: „first citizen object”.
- – Funkcje: funkcje wyższego rzędu.
- – Funkcje anonimowe.
- – Dopełnienia funkcji.
- – Zastosowania funkcji.
- Laboratoria: od teorii do praktyki czyli rozwiązywanie problemów związanych z omówionymi zagadnieniami na wykładzie.

Generatory:

- To obiekt tworzący sekwencje elementów
- „Tworzenie” elementów odbywa się tylko na żądanie (tzw. leniwe obliczanie)
- Typy:
  - Wyrażenia generatorowe
  - Funkcje generatorowe

Comprehensions:

- Po polsku znaczy: zrozumienie, pojmowanie
- W Pythonie przykłady:
  - Listy składane (list comprehemsions)
  - Zbiory składane (set comprehensions)
  - Słowniki składane (dictionary comprehensions)

Przykłady:

```
1 x = [1, 2, 3]
2 l_kwd = []
3 for i in x:
4     l_kwd.append(i**2)
5 l_kwd
```

```
[1, 4, 9]
```

```
1 x = [1, 2, 3]
2 li_kwd=[i**2 for i in x]
3 li_kwd
```

```
[1, 4, 9]
```

```
1 x = [1, 2, 3]
2 z_kwd = set()
3 for i in x:
4     z_kwd.add(i**2)
5 z_kwd
```

```
{1, 4, 9}
```

```
1 x = [1, 2, 3]
2 zb_kwd = {i**2 for i in x}
3 zb_kwd
```

```
{1, 4, 9}
```

```
1 x = [1, 2, 3]
2 s_kwd = {}
3 for i in x:
4     s_kwd[i]=i**2
5 s_kwd
```

```
{1: 1, 2: 4, 3: 9}
```

```
1 x = [1, 2, 3]
2 sl_kwd={i:i**2 for i in x}
3 sl_kwd
```

```
{1: 1, 2: 4, 3: 9}
```

Przykłady: użycie funkcji

```
1  def kwadrat(x):
2      return x*x
3
4  x = [1, 2, 3]
5  li_kwd=[kwadrat(i) for i in x]
6  li_kwd
```

```
[1, 4, 9]
```

```
1  potega = lambda x, n : x**n
2  pomnoz = lambda x, n: n*x
3  x = [1, 2, 3]
4  li_szesc = [potega(i, 3) for i in x]
5  s_pomn = {i:pomnoz(i, i*20) for i in x}
6  print("Lista szescianów:",li_szesc)
7  print("Slownik:", s_pomn)
```

```
Lista szescianów: [1, 8, 27]
Slownik: {1: 20, 2: 80, 3: 180}
```

```
1  wyklad = lambda x, n: n**x
2  x = [1, 2, 3]
3  zb_wykl={wyklad(i, i) for i in x}
4  zb_wykl
```

```
{1, 4, 27}
```

Przykład: filtrowanie przy pomocy instrukcji `if`

```
1 x = [1, 2, "a", 3, (4,)]
2 li_calk=[i for i in x if isinstance(i, int)]
3 li_calk
```

```
[1, 2, 3]
```

```
1 x = [i for i in range(6)]
2 li_3_5 = [i for i in x if i%3==0 or i%5==0]
3 li_3_5
```

```
[0, 3, 5]
```

```
1 x = [i for i in range(6)]
2 li_kwP = [y for i in x if (y := i**2)%2 == 0]
3 li_kwP
```

```
[0, 4, 16]
```

Przykłady: filtrowanie cd.

```
1 x = [1, 2, "a", 3, (4,)]
2 li_calk=[i for i in x if isinstance(i, int)]
3 li_calk
```

```
[1, 2, 3]
```

```
1 x = [1, 2, "a", 3, (4,)]
2 li_calk=[i if isinstance(i, int) for i in x]
3 li_calk
```

```
  Cell In[36], line 2
    li_calk=[i if isinstance(i, int) for i in x]
             ^
SyntaxError: expected 'else' after 'if' expression
```

```
1 x = [1, 2, "a", 3, (4,)]
2 li_calk=[i if isinstance(i, int) else i for i in x]
3 li_calk
```

```
[1, 2, 'a', 3, (4,)]
```

```
1 x = [1, 2, "a", 3, (4,), 4.0]
2 li_kwd=[i**2 if isinstance(i, int) or isinstance(i, int)
3           else i for i in x]
4 li_kwd
```

```
[1, 4, 'a', 9, (4,), 4.0]
```

Przykłady: złożone filtrowanie

```
1 x = range(50)
2 li = [y for i in x if i % 4!= 0 if (y := i**2) % 2 == 0]
3 li
```

```
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]
```

```
1 x = range(50)
2 li = [y for i in x if i % 4!= 0 and (y := i**2) % 2 == 0]
3 li
```

```
[4, 36, 100, 196, 324, 484, 676, 900, 1156, 1444, 1764, 2116]
```

```
1 x = range(20)
2 li = [y for i in x if i % 4!= 0 or (y := i**2) % 2 == 0]
3 print(li)
```

```
[0, 0, 0, 0, 16, 16, 16, 16, 64, 64, 64, 64, 144, 144, 144, 144, 256, 256, 256, 256]
```

```
1 x = range(20)
2 li = [y for i in x if  (y := i**2) % 2  or i % 4!= 0 == 0]
3 print(li)
```

```
[1, 4, 9, 25, 36, 49, 81, 100, 121, 169, 196, 225, 289, 324, 361]
```

Uwagi:

```
1 x=[(1, 2) for _ in range(3)]
2 print(x)
```

```
[(1, 2), (1, 2), (1, 2)]
```

```
1 i = "to jest i"
2 x = [i for i in range(10)]
3 print("lista:",x)
4 print("A i to?:",i)
```

```
lista: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
A i to?: to jest i
```

```
1 i = "to jest i"
2 x = [i for j in range(10) if (i :=j**2)<10]
3 print("lista:",x)
4 print("A i to?:",i)
```

```
lista: [0, 1, 4, 9]
A i to?: 81
```

Wyrażenia generatorowe (generator comprehensions)

- Są bardzo podobne do list składanych – tylko zamiast nawiasów **[ ]** maja nawiasy **( )**
- Nie generują całej listy od razy zajmując pamięć, tylko element po elemencie w ramach potrzeby
- Można używać wszystkich konstrukcji pokazanych dla comprehensions

```
1 lista = [i for i in range(10)]
2 gen = (i for i in range(10))
3 print("lista:", lista)
4 print("gener:", gen)
```

```
lista: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
gener: <generator object <genexpr> at 0x7f7f42637030>
```

```
1 list_gen=list(gen)
2 print("Li_gen", list_gen)
```

```
Li_gen [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Uwagi (1/3):

```
1  gen = (i for i in range(5))
2  for i in gen:
3      print('i:',i)
4      if i >= 2:
5          break
6  print("Druga pętla")
7  for i in gen:
8      print("A teraz i",i)
```

```
i: 0
i: 1
i: 2
Druga pętla
A teraz i 3
A teraz i 4
```

UWAGA: Wyrażenia generatorowe możemy użyć tylko raz !!!

```
1  gen = (i for i in range(5))
2  for i in gen:
3      print('i:',i)
4      if i >= 2:
5          break
6  print("Druga pętla - 2")
7  gen = (i for i in range(5))
8  for i in gen:
9      print("A teraz i",i)
```

```
i: 0
i: 1
i: 2
Druga pętla - 2
A teraz i 0
A teraz i 1
A teraz i 2
A teraz i 3
A teraz i 4
```

Uwagi (2/3) - preferowany sposób

```
1  for i in (i for i in range(5)):
2      print('i:',i)
3      if i >= 2:
4          break
5  print("Druga pętla - 2")
6  for i in (i for i in range(5)):
7      print("A teraz i",i)
```

```
i: 0
i: 1
i: 2
Druga pętla - 2
A teraz i 0
A teraz i 1
A teraz i 2
A teraz i 3
A teraz i 4
```

Uwagi (3/3):

```
1 gen = (i for i in range(5))
2 print("Suma:", sum(gen))
```

```
Suma: 10
```

```
1 gen = (i for i in range(5))
2 dl=len(gen)
3 print("Długość:",dl)
```

```
---------------------------------------------------------------
TypeError                                 Traceback
Cell In[11], line 2
      1 gen = (i for i in range(5))
----> 2 dl=len(gen)
      3 print("Długość:",dl)

TypeError: object of type 'generator' has no len()
```

```
1 gen = (i for i in range(5))
2 dl=len(list(gen))
3 print("Długość:",dl)
```

```
Długość: 5
```

```
1 gen = (i for i in range(5))
2 dl=sum(1 for _ in gen)
3 print("Długość:",dl)
```

```
Długość: 5
```

Generator range:

```
1  gen=range(4)
2  print("gen:", gen," typ:", type(gen))
```

```
gen: range(0, 4)  typ: <class 'range'>
```

```
1  gen=range(4)
2  print("gen=",gen)
3  print("Suma liczb do 4=",sum(gen))
4  for i in gen:
5      print("i=",i)
```

```
gen= range(0, 4)
Suma liczb do 4= 6
i= 0
i= 1
i= 2
i= 3
```

```
1  dl=len(gen)
2  print("Dlugosc:", dl)
```

```
Dlugosc: 4
```

Uwaga: generator range się nie wyczerpuje, można go używać wiele razy oraz działa funkcja **len**

Funkcje generatorowe (1/5):

- Różni od zwykłej funkcji tym, że wynik zwraca za pomocą instrukcji **`yield`**, a nie **`return`**
- Każda instrukcja **`yield`** jest „aktywna”
- Wymagają „specjalnego” wywołania

```
1  def MojGenerator():
2      print('Pierwszy raz')
3      yield 11
4
5      print('DrugiRaz')
6      yield 22
7
8      print('Trzeci Raz')
9      yield 33
10 print(type(MojGenerator))
11 print(MojGenerator())
```

```
<class 'function'>
<generator object MojGenerator at
```

```
1  def MojaFunkcja():
2      print('Pierwszy raz')
3      return 11
4
5      print('DrugiRaz')
6      return 22
7
8      print('Trzeci Raz')
9      return 33
10 print(type(MojaFunkcja))
11 print(MojaFunkcja())
```

```
<class 'function'>
Pierwszy raz
11
```

Funkcje generatorowe (2/5):

```
1  def MojGenerator():
2      print('Pierwszy raz')
3      yield 11
4
5      print('DrugiRaz')
6      yield 22
7
8      print('Trzeci Raz')
9      yield 33
```

```
1  def MojaFunkcja():
2      print('Pierwszy raz')
3      return 11
4
5      print('DrugiRaz')
6      return 22
7
8      print('Trzeci Raz')
9      return 33
```

```
1  for x in MojGenerator():
2      print("x=",x)
```

```
Pierwszy raz
x= 11
DrugiRaz
x= 22
Trzeci Raz
x= 33
```

```
1  for x in MojaFunkcja():
2      print("x=",x)
```

```
Pierwszy raz
---------------------------------------
TypeError
Cell In[27], line 1
----> 1 for x in MojaFunkcja():
      2     print("x=",x)

TypeError: 'int' object is not iterable
```

Funkcje generatorowe (3/5):

```python
def GenLiczb(*,start=0,koniec=1, krok=0.1):
    i=start
    while i < koniec:
        yield i
        i+=krok

lista=[x for x in GenLiczb()]
print(lista)
```

```
[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 0.9999999999999999]
```

```python
gen=GenLiczb()
print("Generuje liczby:",end=" ")
for i in gen:
    print(i, end=", ")
print()
s1=sum(gen)
print("a suma tych liczb wynosi:",s1)
```

```
Generuje liczby: 0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6, 0.7, 0.7999999999999999, 0.8999999999999999, 0.9999999999999999,
a suma tych liczb wynosi: 0
```

Funkcje generatorowe (4/5):

```python
def GenLiczb1(*,start=0, koniec=1, krok=0.1):
    x=start
    i=0
    while x < koniec:
        yield x
        i+=1
        x=start+i*krok

lista=[x for x in GenLiczb1()]
print(lista)
```

```
[0, 0.1, 0.2, 0.30000000000000004, 0.4, 0.5, 0.6000000000000001, 0.7000000000000001, 0.8, 0.9]
```

```python
gen=GenLiczb1(krok=0.3)
print("Generuje liczby:",end=" ")
for i in gen:
    print(i, end=", ")
print()
gen=GenLiczb1(krok=0.3) #odnów generator
s1=sum(gen)
print("a suma tych liczb wynosi:",s1)
```

```
Generuje liczby: 0, 0.3, 0.6, 0.8999999999999999,
a suma tych liczb wynosi: 1.7999999999999998
```

Funkcje generatorowe (5/5):

```python
def GenNieskonczony():
    i = 0
    while True:
        yield i
        i+= 1

print("Pętla nieskonczina")
print("Wyświetlam wszystkie liczby naturalne")
print('Program "trochę" niebezpieczny')
for i in GenNieskonczony():
    print("i = ",i)
    #if i > 10: #odkomentuj jeśli chcesz
    #    break  #nieskończonej petli
```

```
Pętla nieskonczina
Wyświetlam wszystkie liczby naturalne
Program "trochę" niebezpieczny
i =  0
i =  1
i =  2
i =  3
i =  4
i =  5
i =  6
i =  7
i =  8
i =  9
i =  10
i =  11
```

Wbudowane funkcje wyższego rzędu (1/8) — funkcje `min()` oraz `max()`:

- Funkcje znajdują ekstremalny „element” w obiekcie iterowalnym (krotka, lista, słownik, ciąg znaków, zbiory, generatory) albo w dowolnej ilości parametrów
- Posiadają dwa opcjonalne nazwane parametry:
  - **`default=`** – zwracana wartość domyślną
  - **`key=`** – funkcja jednej zmiennej służąca do porównywania elementów

Wbudowane funkcje wyższego rzędu (2/8) – funkcje min() oraz max():

```
1 MIN=min([1, -2, 3, 4, -5])
2 MAX=max([1, -2, 3, 4, -5])
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: -5
Max: 4
```

```
1 MIN=min("AbraKadabra")
2 MAX=max("AbraKadabra")
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: A
Max: r
```

```
1 MIN=min(1, 2, 3, 4, -5, -6)
2 MAX=max(1, 2, 3, 4, -5, -6)
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: -6
Max: 4
```

```
1 MIN=min({1, 2, 3, -4})
2 MAX=max({1, 2, 3, -4})
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: -4
Max: 3
```

```
1 MIN=min(1, 2, 3, 4, -5, -6, "a")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent ca
Cell In[22], line 1
----> 1 MIN=min(1, 2, 3, 4, -5, -6, "a")

TypeError: '<' not supported between instances of 'str' and 'int'
```

„Elementy” w funkcjach muszą dać się porównywać !!!

Wbudowane funkcje wyższego rzędu (3/8) – funkcje min() oraz max():

```
1 MIN=min({}) #Pusty słownik
```

```
---------------------------------------------
ValueError                                Tr
Cell In[14], line 1
----> 1 MIN=min({}) #Pusty słownik

ValueError: min() arg is an empty sequence
```

```
1 MIN=min({}, default="Pusty słownik")
2 print("Min:", MIN)
```

```
Min: Pusty słownik
```

```
1 MAX=max((), default="Pusta krotka")
2 print("Max:", MAX)
```

```
Max: Pusta krotka
```

Wbudowane funkcje wyższego rzędu (4/8) – funkcje min() oraz max():

Co jest tym magicznym „elementem” ?

```
1 MIN=min([(1, 2, 3), (-1, 2), (3,)])
2 MAX=max([(1, 2, 3), (-1, 2), (3,)])
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: (-1, 2)
Max: (3,)
```

```
1 MIN=min({"a":1, "b":3, "A":23})
2 MAX=max({"a":1, "b":3, "A":23})
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: A
Max: b
```

Wbudowane funkcje wyższego rzędu (5/8) – funkcje min() oraz max():

Szukamy po „długości” krotki w liście

```
1 MIN=min([(1, 2, 3), (-1, 2), (3,)], key=lambda x: len(x))
2 MAX=max([(1, 2, 3), (-1, 2), (3,)], key=lambda x: len(x))
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: (3,)
Max: (1, 2, 3)
```

```
1 MIN=min(1, 2, 3, 4, -5, -6, "a", "1", "X", key=lambda x: str(x))
2 MAX=max(1, 2, 3, 4, -5, -6, "a", "1", "X", key=lambda x: str(x))
3 print("Min:", MIN, "typ el:",type(MIN))
4 print("Max:", MAX, "typ el:",type(MAX))
```

```
Min: -5 typ el: <class 'int'>
Max: a typ el: <class 'str'>
```

```
1 MIN=min("A","N", "m", "a", "1", key=str.lower)
2 MAX=max("A","N", "m", "a", "1", key=str.lower)
3 print("Min:", MIN)
4 print("Max:", MAX)
```

```
Min: 1
Max: N
```

Wbudowane funkcje wyższego rzędu (6/8) — funkcje `min()` oraz `max()`:

```
1  def iloczyn(x):
2      ilo=1
3      for i in x:
4          ilo*=i
5      return ilo
6
7  MIN=min([(1, 2, 3), (-1, 2), (3,)], key=iloczyn)
8  MAX=max([(1, 2, 3), (-1, 2), (3,)], key=iloczyn)
9  print("Min:", MIN)
10 print("Max:", MAX)
```

```
Min: (-1, 2)
Max: (1, 2, 3)
```

Wbudowane funkcje wyższego rzędu (7/8) – funkcje min() oraz max():

```
1  cennik={"jabłka":3.20, "banany":4.20,
2          "marchewka":1.20, "kapusta":2.20}
3  MIN=min(cennik)
4  print("Najtaniej:",MIN)
```

```
Najtaniej: banany
```

```
1  MIN=min(cennik.values())
2  print("Najtaniej:",MIN)
```

```
Najtaniej: 1.2
```

```
1  MIN=min(cennik.items())
2  print("Najtaniej:",MIN)
```

```
Najtaniej: ('banany', 4.2)
```

```
1  MIN=min(cennik.items(),
2          key=lambda x: x[1])
3  print("Najtaniej:",MIN)
```

```
Najtaniej: ('marchewka', 1.2)
```

Wbudowane funkcje wyższego rzędu (8/8) – funkcje min() oraz max():

```
1 MIN=min((3-i for i in range(6)))
2 MAX=max((3-i for i in range(6)))
3 print("MIN=", MIN, "MAX=",MAX)
```

```
MIN= -2 MAX= 3
```

```
1  def GenLiczb1(*,start=0, koniec=1, krok=0.1):
2      x=start
3      i=0
4      while x < koniec:
5          yield x
6          i+=1
7          x=start+i*krok
8
9  MIN=min(GenLiczb1())
10 MAX=max(GenLiczb1())
11 print("MIN=", MIN, "MAX=",MAX)
```

```
MIN= 0 MAX= 0.9
```

Wbudowane funkcje wyższego rzędu (1/5) – funkcja sum():

- Funkcja oblicza „sumę” elementów dla iteratora zaczynając od zera
- Posiada jeden opcjonalny nazwany argument **start=0**

```
1 SUMA=sum((1, 2, 3, -4))
2 print("Suma: ", SUMA)
```

```
Suma:  2
```

```
1 SUMA=sum(1, 2, 3, -4)
2 print("Suma: ", SUMA)
```

```
---------------------------------------------------------------
TypeError                                 Traceback (
Cell In[48], line 1
----> 1 SUMA=sum(1, 2, 3, -4)
      2 print("Suma: ", SUMA)

TypeError: sum() takes at most 2 arguments (4 given)
```

Wbudowane funkcje wyższego rzędu (2/5) – funkcja sum():

```
1 SUMA=sum(("a", "b", "c"))
2 print("Suma:", SUMA)
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most rece
Cell In[60], line 1
----> 1 SUMA=sum(("a", "b", "c"))
      2 print("Suma:", SUMA)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
1 SUMA=sum(("a", "b", "c"), start="")
2 print("Suma:", SUMA)
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recer
Cell In[59], line 1
----> 1 SUMA=sum(("a", "b", "c"), start="")
      2 print("Suma:", SUMA)

TypeError: sum() can't sum strings [use ''.join(seq) instead]
```

Wbudowane funkcje wyższego rzędu (3/5) – funkcja sum():

```
1 SUMA=sum(((1, 2), (2, 3, 4)), start=())
2 print("Suma:", SUMA)
```

```
Suma: (1, 2, 2, 3, 4)
```

```
1 SUMA=sum([[1, 2], [2, 3, 4]], start=[])
2 print("Suma:", SUMA)
```

```
Suma: [1, 2, 2, 3, 4]
```

Wbudowane funkcje wyższego rzędu (4/5) – funkcja sum( ):

```
1 SUMA=sum(({"a":1, "b":2}, {"c":3}), start={})
2 print("Suma:", SUMA)
```

```
TypeError                                 Traceback (most recent
Cell In[77], line 1
----> 1 SUMA=sum(({"a":1, "b":2}, {"c":3}), start={})
      2 print("Suma:", SUMA)

TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

```
1 SUMA=sum(({1, 2}, {3}), start=set())
2 print("Suma:", SUMA)
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recer
Cell In[76], line 1
----> 1 SUMA=sum(({1, 2}, {3}), start=set())
      2 print("Suma:", SUMA)

TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Wbudowane funkcje wyższego rzędu (5/5) – funkcja sum():

```
 1  def GenLiczb1(*,start=0, koniec=1, krok=0.1):
 2      x=start
 3      i=0
 4      while x < koniec:
 5          yield x
 6          i+=1
 7          x=start+i*krok
 8
 9  sum1=sum(GenLiczb1())
10  print('"Domyślna" suma = ',sum1)
11  suma=sum(GenLiczb1(start=-5, koniec=10, krok=0.5))
12  print("Suma = ",suma)
```

```
"Domyślna" suma =  4.500000000000001
Suma =  67.5
```

Dziękuję za uwagę