

Kurs rozszerzony języka Python

Wykład 3.

Marcin Młotkowski

26 października 2021

Plan wykładu

- 1 Programowanie funkcjonalne
- 2 Funkcje
 - Listy i funkcje
 - Operacje na kolekcjach (iterable)
 - Generatory
 - itertools i przykład
- 3 funkcyjne/wyjście
 - Pliki tekstowe

Plan wykładu

- 1 Programowanie funkcjonalne
- 2 Funkcje
 - Listy i funkcje
 - Operacje na kolekcjach (iterable)
 - Generatory
 - itertools i przykład
- 3 funkcyjne/wyjście
 - Pliki tekstowe

Python **nie** jest językiem funkcjonalnym.

Python zawiera elementy wspierające programowanie funkcjonalne:

- funkcje są obiektami pierwszej klasy;
- z funkcji można zrobić inne funkcje;
- programowanie leniwe.

Plan wykładu

- 1 Programowanie funkcjonalne
- 2 Funkcje
 - Listy i funkcje
 - Operacje na kolekcjach (iterable)
 - Generatory
 - itertools i przykład
- 3 funkcyjne/wyjście
 - Pliki tekstowe

Funkcje są wartościami pierwszej klasy, tj. mogą być argumentami innych funkcji czy metod.

Funkcje

```
def calka_oznaczona(f, a, b):  
    krok, suma, x = .1, 0, a  
    while x + krok < b:  
        suma += f(x)*krok  
        x += krok  
    return suma
```

```
def fun(x): (x + 1)/x  
print(calca(fun, 0, 5))
```


Funkcje, cd

Inne przykłady

```
def square(n): return n*n

def double(n): return 2 * n

funList = [ square, double ]
for f in funList:
    print(f(10))
```

Lambda funkcje

```
double = lambda x: 2*x
```

Lambda funkcje

```
double = lambda x: 2*x
```

```
square = lambda x: x*x
```

Lambda funkcje

```
double = lambda x: 2*x  
  
square = lambda x: x*x  
  
funList = [ double, square ]  
print(calka(square, 0, 10))
```

Lambda funkcje, cd

```
funList = [ lambda x: 2*x, lambda x: x*x ]  
  
print(calka(lambda x: x*x, 0, 10))
```

Dwuargumentowe funkcje lambda

```
f = lambda x, y: 2*x + y
```

Operacje na listach

```
print(list(filter(fun, lista)))  
print(list(map(lambda x: 2*x, lista)))  
print(list(reduce(lambda x, y: x + y, lista, 0)))
```

Listy składane

Przykłady

```
lista = range(10)  
[ 2 * x for x in lista ]
```


Listy składane

Przykłady

```
lista = range(10)
[ 2 * x for x in lista ]

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Listy składane

Przykłady

```
lista = range(10)
[ 2 * x for x in lista ]

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
[ (x, x*x*x) for x in lista if x % 3 == 0 ]
```

Listy składane

Przykłady

```
lista = range(10)  
[ 2 * x for x in lista ]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
[ (x, x*x*x) for x in lista if x % 3 == 0 ]
```

```
[(0, 0), (3, 27), (6, 216), (9, 729)]
```

Listy składane, dalsze przykłady

Przetwarzanie list stringów

```
lista = [ "mOnty", "pyTHon's", "FlyinG", "circus" ]
```

Listy składane, dalsze przykłady

Przetwarzanie list stringów

```
lista = [ "mOnty", "pyTHon's", "FlyinG", "circus" ]
```

```
lista = [ e[0].upper() + e[1:].lower() for e in lista ]
```

Listy składane zagnieżdżone

Talia kart

```
kolory = [ 'Kier', 'Karo', 'Trefl', 'Pik']  
figury = ['K', 'D', 'W'] + list(range(2, 11)) + ['A']  
  
[ (kolor, fig) for kolor in kolory for fig in figury ]])
```

Ważne

- filter
- map
- reduce

Ważne

- filter
- map
- reduce

Python 2.*

Funkcje zwracają listę

Python 3.*

Funkcje zwracają iterator

Programowanie funkcjonalne

Operatory (moduł operator)

```
operator.add(x,y)  
operator.mul(x,y)  
operator.pow(x,y)  
...
```

Programowanie funkcjonalne

Operatory (moduł operator)

```
operator.add(x,y)  
operator.mul(x,y)  
operator.pow(x,y)  
...
```

Iloczyn skalarny

```
sum(map(operator.mul, vector1, vector2))
```

Iterator a lista

Python 3.*

```
list(filter(lambda x : x > 2, [1,2,3,4]))
```

Biblioteka *itertools*

Mnóstwo funkcji produkujących generatory

Biblioteka *itertools*

Mnóstwo funkcji produkujących generatory

Kolejne potęgi 2

```
it = map(lambda x : 2**x, itertools.count())  
next(it)  
next(it)  
next(it)  
...
```

Definicje

Generator

Generator to funkcja, która zwraca iterator.

Definicje

Generator

Generator to funkcja, która zwraca iterator.

Wyrażenie generatorowe

Wyrażenie generatorowe to wyrażenie, która zwraca iterator.

Jak implementować funkcje generatorowe

yield

Wykorzystanie **yield**

Implementacja nieskończonej listy potęg 2

```
def power2():  
    power = 1  
    while True:  
        yield power  
        power = power * 2
```

```
it = power2()  
for x in range(4):  
    print (next(it))
```

Wykorzystanie **yield**

Implementacja nieskończonej listy potęg 2

```
def power2():  
    power = 1  
    while True:  
        yield power  
        power = power * 2
```

```
it = power2()  
for x in range(4):  
    print (next(it))
```

Nieskończona pętla

```
for i in power2(): print (i)
```

Wyrażenia generatorowe

Instrukcja

```
wyr_generatorowe = (i** 2 for i in range(5))
```

jest równoważna

```
def wyr_generatorowe():  
    for i in range(5):  
        yield i**2
```

Listy składane a wyrażenia generatotorowe

```
[i**2 for i in range(5)]  
(i**2 for i in range(5))
```

Listy składane a wyrażenia generatorowe

```
[i**2 for i in range(5)]  
(i**2 for i in range(5))
```

Jak zrobić z wyrażenia generatorowego listę

```
list(i**2 for i in range(5))
```

Zastosowanie

String szesnastkowo

```
":".join(":02x".format(ord(c)) for c in s)
```

yield from

Zamiast

Example

```
for item in iterable: yield item:
```

można pisać

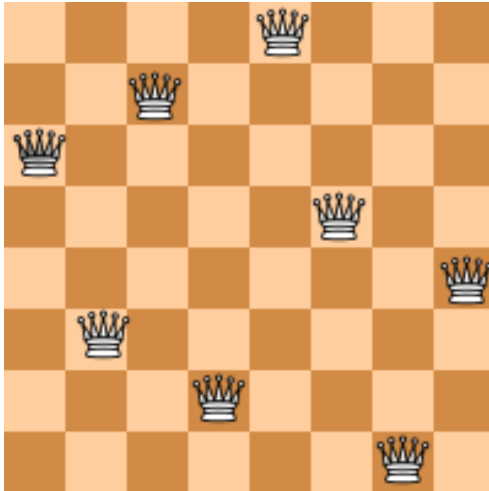
Example

```
yield from iterable
```

Moduł itertools

Generatory permutacji, kombinacji, iloczynów kartezjańskich

Problem 8 hetmanów



Źródło: Wikipedia

Plan wykładu

- 1 Programowanie funkcjonalne
- 2 Funkcje
 - Listy i funkcje
 - Operacje na kolekcjach (iterable)
 - Generatory
 - itertools i przykład
- 3 funkcyjne/wyjście
 - Pliki tekstowe

Operacje na plikach

Otwarcie i zamknięcie pliku

```
fh = open('plik', 'r')
```

```
...
```

```
fh.close()
```

Atrybuty otwarcia

'r'	odczyt
'w'	zapis
'a'	dopisanie
'r+'	odczyt i zapis
'rb', 'wb', 'ab'	odczyt i zapis binarny

Metody czytania pliku

Odczyt całego pliku

```
fh.read()
```

Odczyt tylko size znaków

```
fh.read(size)
```

Odczyt wiersza, wraz ze znakiem '\n'

```
fh.readline()
```

Zwraca listę odczytanych wierszy

```
fh.readlines()
```

Tryby odczytu/zapisu

Tryb tekstowy

`fh.read()` zwraca string w kodowaniu takie jak ustawiono przy otwarciu pliku: `open(fname, 'r', encoding='utf8')`.

Tryb binarny `open(fname, 'rb')`

`fh.read()` zwraca ciąg binarny.

Odczyt pliku

Przykład

```
fh = open('test.py', 'r')
while True:
    wiersz = fh.readline()
    if len(wiersz) == 0: break
    print(wiersz)
fh.close()
```

Odczyt pliku

Przykład

```
fh = open('test.py', 'r')
while True:
    wiersz = fh.readline()
    if len(wiersz) == 0: break
    print(wiersz)
fh.close()
```

Inny przykład

```
fh = open('test.py', 'r')
for wiersz in fh:
    print(wiersz)
```

Zapis do pliku

```
fh.write('dane zapisywane do pliku\n')  
fh.writelines(['to\n', 'są\n', 'kolejne\n', 'wiersze\n'])
```


Zamykanie pliku

Uwaga

Zawsze należy zamykać pliki.

Przykład

```
try:
    fh = open('nieistniejący', 'r')
    data = fh.read()
finally:
    fh.close()
```

Zamykanie pliku

Uwaga

Zawsze należy zamykać pliki.

Przykład

```
try:  
    fh = open('nieistniejacy', 'r')  
    data = fh.read()  
finally:  
    fh.close()
```

Alternatywne zamykanie pliku

```
del fh
```

Zamykanie pliku

Porada

```
with open('nieistniejący', 'r') as fh:  
    data = fh.read()
```

Formaty danych

- Pliki tekstowe
- pickle
- Pliki z rekordami
- Pliki CSV
- Pliki *.ini
- XML
- ...