Faculty of Geodesy
and Cartography
WARSAW UNIVERSITY OF TECHNOLOGY

Informatyka Geodezyjna - wykłady/ćwiczenia, rok akad. 2019-2020

Wyk. 11: Python – programowanie funkcyjne

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

Warsaw University
of Technology

1. Programowanie funkcyjne (funkcjonalne)

2. Programowanie funkcyjne w Pythonie
   - Wyrażenie lambda
   - Funkcja map()
   - Funkcja filter()
   - Funkcja reduce()

3. Comprehensing Python
   - Listy składane (list comprehension) - mapowanie, filtrowanie

4. Programowanie funkcyjne - podsumowanie

- Programowanie funkcyjne – filozofia programowania będąca odmianą programowania deklaratywnego, w której **funkcje należą do wartości podstawowych**, podstawa jest wartościowanie funkcji (często rekurencyjnych), a nie na wykonywanie poleceń.
- Języki funkcjonalne mają swoje źródło w logice matematycznej i **rachunku lambda** (opracowany w latach 30. XX wieku przez Alonzo Churcha), a imperatywne języki obejmują oparty na stanach model obliczeniowy opracowany przez Alana Turinga.
- Programowanie deklaratywne często traktuje programy jako pewne hipotezy wyrażone w logice formalnej, a wykonywanie obliczeń jako ich dowodzenie.
- W paradygmacie funkcyjnym opisywane są warunki jakie musi spełniać rozwiązanie (co chcemy osiągnąć), a w proceduralny opisywane są sekwencję kroków, które do niego prowadzą (jak to zrobić).

Programowanie proceduralne (imperatywne)

- co robić?
- w jakiej kolejności?
- instrukcje, pętle, procedury, obiekty

Programowanie funkcyjne (deklaratywne)

- jaki ma być efekt?
- jak skomponować program?
- funkcje, rekurencja

Python językiem funkcyjnym nie jest (jest językiem obiektowym), a już na pewno nie jest językiem czysto funkcyjnym, ale posiada pewne cechy charakterystyczne dla tych języków. Przede wszystkim umożliwia korzystanie z wyrażeń lambda oraz dostarcza wymienione powyżej funkcje map, filter, reduce.

- **Mapowanie** polega na pobieraniu funkcji oraz obiektu pozwalającego na iterację i wygenerowanie nowego elementu iterowanego, w którym element będzie wynikiem wywołania funkcji względem odpowiadającego mu elementu w początkowym obiekcie.
- **Redukcja** polega na pobieraniu funkcji i obiektu pozwalającego na iterację, a następnie wygenerowaniu pojedynczej wartości.
- **Filtrowanie** pozwala na iterację i wygenerowanie nowego iteratora, w którym każdy element pochodzi z początkowego iteratora - pod warunkiem, że funkcja wywołana względem tego iteratora zwróciła wartość `True`.

Python językiem funkcyjnym nie jest (jest językiem obiektowym), a już na pewno nie jest językiem czysto funkcyjnym, ale posiada pewne cechy charakterystyczne dla tych języków. Przede wszystkim umożliwia korzystanie z wyrażeń lambda oraz dostarcza wymienione powyżej funkcje map, filter, reduce.

- `lambda` – funkcja anonimowa (funkcja bez nazwy);
- `map` – odwzorowanie funkcji na sekwencję;
- `filter` – odfiltrowanie elementów sekwencji, dla których funkcja zwraca True;
- `reduce` – ciągle stosowanie funkcji dla sekwencji – zwraca jedną wartość;
- `List Comprehension` – generator list i słowników;
- Równie ważne są iteratory oraz generatory dostępne w modułach bibliotek standardowych takich jak `itertools` i `functools`.

Wyrażenie **lambda** lub funkcja **lambda** to sposób na tworzenie małych **anonimowych funkcji** (funkcji bez nazwy). Funkcje te są funkcjami „wyrzucania" (ang. *throw-away function*), tzn. że sa one potrzebne tylko tam, gdzie zostały utworzone i tam zwracają wynik. Funkcje Lambda są używane głównie w połączeniu z funkcjami `filter()`, `map()` i `reduce ()`.

Składnia:

```
lambda arg1, arg2,... argN : wyrażenie_wykorzystujące_argumenty
```

Przykład: funkcja podnoszenia do kwadratu

```
def f(x):
    return x**2
f(3)
```

```
9
```

```
g = lambda x: x**2
g(3)
```

```
9
```

```
(lambda x: x**2)(3)
```

```
9
```

- Odwzorowuje funkcję na każdy element kolekcji i otrzymuje w wyniku nową kolekcję.
- Funkcja **map** jako argumenty przyjmuje: funkcję (`funkacja`) oraz jedną lub większą liczbę sekwencji (`sekwencja`) i zbiera wynik wywołania funkcji z równoległymi elementami pobranymi z sekwencji. Tak jak zip, funkcja map jest w Pythonie 3.x generatorem wartości, dlatego konieczne jest przekazanie jej do list w celu zebrania wszystkich wyników naraz (rzutowanie na listę). (Lutz, 2011)

Składnia:

```
1 map(funkacja, sekwencja)
```

Przykład: odwzorowuje funkcję `abs` na każdy element sekwencji i zebranie wyników

```
1 map(abs, [1,-2, -9, -6])
```

```
1 <map at 0x7f0f8056ca90>
```

```
1 list(map(abs, [1,-2, -9, -6]))
```

```
1 [1, 2, 9, 6]
```

In[1]:

```
# funkcja
def addition(n):
    return n + n

numbers = [1, 2, 3, 4]
result = map(addition, numbers)
print(list(result))
```

Out1:

```
[2, 4, 6, 8]
```

lub z wyrażeniem `lambda`

In[2]:

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
```

Out2:

```
[1, 4, 9, 16, 25]
```

- Uruchamia funkcję filtrującą dla każdego elementu sekwencji i zwraca w wyniku nową sekwencje **zawierającą tylko te elementy dla których *użyta funkcja* zwraca logiczną prawdę `True`**.
- Przypomina pętlę `for`, ale jest funkcją wbudowaną i wydajnieszą w działaniu (szybszą).

Składnia:

```
1 filter(funkcja, sekwencja)
```

Przykład: odfiltrowanie elementów nieparzystych z listy

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

```
1 <filter at 0x7f0f8056ca90>
```

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

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

In[3]:

```
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False

sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']
filtered = filter(fun, sequence)
```

Out:

```
['e', 'e']
```

In[4]:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
```

In[5]:

```
number_list = range(-5, 5)
less_than_zero = []
for x in number_list:
    if  x < 0 :
        number_list.append(list)
print(less_than_zero)
```

Out:

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

```
fibonacci      = [0,1,1,2,3,5,8,13,21,34,55]
odd_numbers    = list(filter(lambda x: x % 2, fibonacci))
even_numbers   = list(filter(lambda x: x % 2 -1, fibonacci))
```

- Uruchamia funkcje (`function`) **kumulując wyniki jej działania dla każdego kolejnego elementu** sekwencji (`sequence`), jako wynik zwraca jedną wartość.
- Od Pythona 3.x funkcja `reduce()` nie jest już funkcją wbudowaną (jak w Python 2) jest natomiast funkcją dostępną w bibliotece standardowej `funkools`, dostępna jako: (`funkools.reduce()`)

Składnia:

```
1 reduce(function, sequence)
```

Przykład: poniżej zapisana funkcja wykonuje następujące obliczenia $((((1+2)+3)+4)+5)$:

```
1 from functools import reduce
2 reduce(lambda x, y: x+y, [1, 2, 3, 4, 5])
```

```
1 15
```

In[6]:

```
from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
print(product)
```

Out1:

```
24
```

In[7]:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num
print(product)
```

Out1:

```
24
```

- Elementy programowania funkcyjnego takie jak: `map()`, i `filter()` są nadal częścią bibliotek wbudowanych (ang. *build-in*) Pythona, jednak `reduce()` została przeniesiona do biblioteki standardowej `functools`.
- Python zachowuje elementy programowania funkcyjnego dla programistów korzystających ze stylu funkcyjnego.
- Jednak w Pythonie istnieje inna alternatywa dla `lambda`, `map()`, `filter()` i `reduce()` tj. **List Comprehension**
- Zrozumienie **List Comprehension** jest bardziej widoczne i łatwiejsze do zrozumienia
- Posiadanie zarówno **List Comprehension**, jak i `lambda`, `map()`, `filter()` i `reduce()` jest niezgodne z motto Pythona: *Powinien istnieć jeden oczywisty sposób na rozwiązanie problemu*,

Comprehensing (generatory) – w Pythonie Python's listy/słowniki/zbory składane są po prostu jednolinijkowym zapisem pętli `for` - bardziej zwięzła i zwarta składnia.

- **Listy składane** (ang. *list comprehension*)

```
1 [wyrażenie for element in kolekcja]
```

- **Słowniki składane** (ang. *Dict comprehension*)

```
1 {klucz:wartosc for element in kolekcja}
```

- **Zbiory składane** (ang. *Set comprehension*)

```
1 {wyrazenie for element in kolekcja}
```

- **Listy składane – mapowanie** – Tworzy nową (anonimową) sekwencję, w której dla każdego elementu (`element`) z kolekcji (`kolekcja`) znajdzie się wynik wyrażenia (`wyrażenie`).

```
1 [wyrażenie for element in kolekcja]
```

- Przykład:

In[8]:
```
1 a = [x**2 for x in range(0, 10)]
```

In[9]:
```
1 a = list(map(lambda x: x**2, range(0, 10)))
```

In[10]:
```
1 a = []
2 for x in range(0, 10):
3     wynik = x**2
4     a.append(wynik)
```

Out:
```
1 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- **Listy składane – filtrowanie** – Tworzy nową (anonimową) sekwencję, w której dla każdego elementu (`element`) z kolekcji (`kolekcja`) znajdzie się wynik wyrażenia (`wyrażenie`) dla którego zostanie spełniony warunek (`warunek`)

```
[wyrażenie for element in kolekcja if warunek]
```

- Przykład:

In[11]:
```
[x**2 for x in range(0, 10) if x**2 > 50]
```

In[12]:
```
a=filter(lambda x : x>50, map(lambda x: x**2, range(0,10)))
```

In[13]:
```
a = []
for x in range(0, 10):
    wynik = x**2
    if wynik > 50:
        a.append(wynik)
```

Out:
```
[64, 81]
```

- Więcej na temat wydajności programowania funkcyjnego: (Lutz, 2011, p.536)
- *The fate of reduce() in Python 3000* by Guido van Rossum: https://www.artima.com/weblogs/viewpost.jsp?thread=98196
- https://www.python-course.eu/python3_lambda.php
- Listy składane: https://realpython.com/list-comprehension-python/

https://stackoverflow.com/questions/1903980/why-list-comprehension-is-called-so-in-python

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

Dziękuje za uwagę

Kinga Węzka kinga.wezka@pw.edu.pl