zaj1-blok3

October 26, 2015

1 Python — dokończenie tematu

1.1 Variable arguments

W pythonie można tworzyć funkcjie przyjmujące dowolną listę argumentów. Zaznacza się to oznaczając argument gwiazdką.

Nie jest to jakoś mega-ważny temat, ale ponieważ na pewno spotkacie taki kod, a relatywnie trudno to wygooglać myślę że warto przedstawić temat

```
In [1]: def varargs_test(*args): # Przyjmuje dowolną liczbę argumentów
            return args

In [2]: type(varargs_test(1, 2, "kotek")) # Wszystkie argumenty będą spakowane do krotki.

Out[2]: tuple

In [3]: varargs_test(1, 2, "kotek")

Out[3]: (1, 2, 'kotek')
```

1.2 Variable arguments

Bardziej kompletny przykład:

```
In [4]: def sum(*args): # *oznacza dowolną ilość argumentów
            result = 0
            for a in args:
                result+=a
            return result

In [5]: sum(1, 2, 3, 4, 5, 6, 7, 8, 9)

Out[5]: 45
```

Rozpakowywanie argumentów

```
In [6]: sum(*range(10)) # Można Pythonowi kazać "rozpakować" dowolną iterable

Out[6]: 45
```

Powyższy przyklad oznacza:

> Drogi interpreterze mam tą ten obiekt, który zawiera argumenty pozycyjne. Spróbuj je dopasować do funkcji i wykonaj tą funkcję

```
In [7]: def test(a, b, c):
            print (a, c)
```

```
In [8]: krotka = 'a', 'b', 'c'
        test(*krotka) # Działą to też dla funkcji które nie są varargsowe

a c

In [9]: krotka = 'a', 'b'
        test(*krotka) # Ale w tym przypadku trzeba podać prawidłową liczbę argumentów!


        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-9-6ccca4783d2a> in <module>()
          1 krotka = 'a', 'b'
    ----> 2 test(*krotka) # Ale w tym przypadku trzeba podać prawidłową liczbę argumentów!


        TypeError: test() missing 1 required positional argument: 'c'
```

1.3 Variable arguments

Varargs mogą koegzystować z keyword arguments:

```
In [10]: from operator import add, mul

         def reduce(*args, op=add, zero=0):
             for a in args:
                 zero = op(zero, a)
             return zero

In [11]: reduce(*range(1, 10), zero=1, op=mul)

Out[11]: 362880
```

1.4 Variable keyword arguments

Można też przekazać dowolną liczbę keyword-arguments

```
In [12]: def func(**kwargs):
             return kwargs

In [13]: func(a=1, b=2, c=10) # Są one przekazywane jako słownik

Out[13]: {'a': 1, 'c': 10, 'b': 2}

In [14]: type(func())

Out[14]: dict

In [15]: dict = {'b': 2, 'a': 1, 'c': 10}

In [16]: func(**dict) # Mam nadzieję że zgadliście że tak można!

Out[16]: {'a': 1, 'b': 2, 'c': 10}
```

1.4.1 List comprehensions

Mamy listę A zawierającą jakieś obiekty, chcemy stworzyć inną listę, która zawiera elementy w jakiś sposób wyznaczane z A.

Na przykład mamy listę zwierząt a chcemy mieć listę ich imion.

```
In [17]: zoo = [
             {
                 "typ": "lis",
                 "imie": "Vitalis",
                 "masa": (5, 'kg')
             },
             {
                 "typ": "miś",
                 "imie": "Koralgol",
                 "masa": (5, 'kg')
             },
             {
                 "typ": "wiewiórka",
                 "imie": "Chip",
                 "masa": (5, 'g')
             },
             {
                 "typ": "miś",
                 "imie": "Puchatek",
                 "masa": (50, 'kg')
             },
         ]
```

1.5 List comprehensions

```
In [18]: imiona = []
         for z in zoo:
             imiona.append(z['imie'])

In [19]: imiona

Out[19]: ['Vitalis', 'Koralgol', 'Chip', 'Puchatek']
```

List comprehensions są ładniejszą metodą na zrobienie powyższej operacji

```
In [20]: imiona_compr = [z['imie'] for z in zoo]

In [21]: imiona_compr

Out[21]: ['Vitalis', 'Koralgol', 'Chip', 'Puchatek']
```

1.6 List comprehensions

```
In [22]: imiona_misiow = [z['imie'] for z in zoo if z['typ']=='miś']

In [23]: imiona_misiow

Out[23]: ['Koralgol', 'Puchatek']
```

2 Set comprehensions

Jak list comprehensions

```
In [24]: typy_zwierzat = {z['typ'] for z in zoo}

In [25]: typy_zwierzat

Out[25]: {'lis', 'miś', 'wiewiórka'}
```

2.1 Dictionary comprehensions

```
In [26]: imie_typ = {z['imie']:z['typ'] for z in zoo}

In [27]: imie_typ['Koralgol']

Out[27]: 'miś'
```

2.2 Generator comprehensions

Wszystkie comprehensions do tej pory tworzyły **kopie** danych. Nie jest to problem jeśli danych jest kilkadziesiąt mb. Jeśli jest ich więcej może powodować to spadek wydajności systemu.

```
In [28]: import random

         IMIONA = [
             'Kubuś', 'Prosiaczek', 'Tygrysek', "Kangurzyca", 'Kangurzątko', 'Królik', 'Sowa', 'Koralgol
             'Dumbo'
         ]

         RODZAJE = [
             'miś', 'świnia', 'tygrys', 'kangur', 'gołąb', 'wiewiórka', 'słoń'
         ]

         def random_animal():
             return {
                 'imie': random.choice(IMIONA),
                 'typ': random.choice(RODZAJE),
                 'masa': (random.uniform(0, 10), 'kg')
             }

In [29]: random_animal()

Out[29]: {'masa': (0.7146304687432181, 'kg'), 'imie': 'Tygrysek', 'typ': 'miś'}

In [30]: DATA = [random_animal() for ii in range(int(1E7))] #Miliard zwierzątek

In [31]: import pickle
         len(pickle.dumps(DATA)) / (1024*1024)

Out[31]: 333.8046598434448
```

Jakieś 250 MB danych o zwierzątkach

2.3 Generator comprehensions

Mamy dużą ilość danych o zwierzątkach, powiedzmy że chcemy zobaczyć jaka jest średnia masa zwierzątka. Możemy to zrobić za pomocą pętli `for`:

```
In [32]: sum = 0
         for z in DATA:
             sum+=z['masa'][0]
         result = sum/len(DATA)
         result

Out[32]: 5.000520312316514
```

Można byłoby zrobić to za pomocą list comprehension:

```
In [33]: del sum # Sum nie jest słowem kluczowym a funkcją wbudowaną
         # Jej nazwę można przykryć (jak w tym przykkładzie przykrywa ją zmienna sum z
         # przykładu wyżej)

In [34]: result_list = sum([z['masa'][0] for z in DATA])/len(DATA)
```

Ale lepiej tak:

```
In [35]: result_gen = sum((z['masa'][0] for z in DATA))
```

Powyższa składnia definiuje generator, czyli coś po czym można iterować tylko raz.

2.4 Generator comprehension

Generator jest obiektem który wspiera tylko jeden interfejs dostępu do danych: jest iterablą.

- Można go wykorzystać do iterowana w pętli `for`
- **Nie zajmuje miejsca w pamięci**
- Nie ma określonej długości
- Nie można pobrać n-tego elementu

2.5 Iteracja po generatorze

```
In [66]: masy_lista = [z['masa'][0] for z in DATA]
         masy_genetator = (z['masa'][0] for z in DATA)

In [67]: type(masy_lista)

Out[67]: list

In [68]: type(masy_genetator)

Out[68]: generator

In [69]: masy_lista[0]

Out[69]: 8.773148227299254

In [70]: masy_genetator[0]
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-70-db10dfe0d07b> in <module>()
----> 1 masy_genetator[0]


    TypeError: 'generator' object is not subscriptable
```

```
In [71]: len(masy_lista)
```

```
Out[71]: 10000000
```

```
In [72]: len(gen)
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-72-cd0a70738f47> in <module>()
----> 1 len(gen)


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

2.6 Czasówki

```
In [73]: %%timeit -r 10
         sum = 0
         for z in DATA:
             sum+=z['masa'][0]
         result = sum/len(DATA)
```

```
1 loops, best of 10: 648 ms per loop
```

```
In [74]: %%timeit -r 10
         result_list = sum([z['masa'][0] for z in DATA])/len(DATA)
```

```
1 loops, best of 10: 712 ms per loop
```

```
In [75]: %%timeit -r 10
         result_gen = sum((z['masa'][0] for z in DATA))
```

```
1 loops, best of 10: 811 ms per loop
```

3 Generator function

Utrudnijmy sobie życie dodając jednostkę do masy zwierzątka

```
In [39]: import random

         IMIONA = [
             'Kubuś', 'Prosiaczek', 'Tygrysek', "Kangurzyca", 'Kangurzątko', 'Królik', 'Sowa', 'Koralgol
```

```
    'Dumbo'
]

RODZAJE = [
    'miś', 'świnia', 'tygrys', 'kangur', 'gołąb', 'wiewiórka', 'słoń'
]

MASY = [
    'mg', 'g', 'kg', 'Mg'
]

def random_animal():
    return {
        'imie': random.choice(IMIONA),
        'typ': random.choice(RODZAJE),
        'masa': (random.uniform(0, 10), random.choice(MASY))
    }
```

```
In [40]: random_animal()
```

```
Out[40]: {'masa': (5.640900786770713, 'Mg'), 'imie': 'Tygrysek', 'typ': 'gołąb'}
```

```
In [41]: DATA2 = [random_animal() for ii in range(int(1E7))] #Miliard zwierzątek
```

3.1 Generator function

Chciałbym (jak ostatnio) wyznaczyć średnią masę zwierzątek w moim zoo, ale tym razem obliczenie masy zwierzątka jest trudne. Jak ostatnio chciałbym mieć iterablę która zwraca mi masę zwierzątka w kilogramach.

```
In [42]: def mass_generator(data):
             for d in data:
                 mass, unit = d['masa']
                 if unit == 'kg':
                     yield mass
                 if unit == 'g':
                     yield mass*1E-3
                 if unit == 'mg':
                     yield mass*1E-6
                 if unit == 'Mg':
                     yield mass*1000
```

```
In [43]: sum(mass_generator(DATA2))/len(DATA2)
```

```
Out[43]: 1252100.083902385
```

```
In [44]: %%timeit
         sum(mass_generator(DATA2))/len(DATA2)
```

```
1 loops, best of 3: 2.32 s per loop
```

4 Generator function

Zatem co właściwie się stało? Otóż funkcja która zawiera słowo `yield` zwraca generator, który zwraca kolejne wartości które pokazują się po kolejnych wywołaniach yield. Taka funkcja nie zwraca jednej wartości a ich ciąg.

```
In [46]: type(mass_generator(DATA2))
```

```
Out[46]: generator
```

```
In [45]: print(list(zip(range(10), mass_generator(DATA2))))
```

```
[(0, 0.00010521436756124381), (1, 4.096475860945121), (2, 851.5196342480069), (3, 7107.939997762918), (4
```