Python - praktyczne podstawy 2

Rafał Rakowski

Agenda

1. Zbiór
2. Iterator
3. Generator

Zbiór (Set)

1. tworzenie
2. modyfikowanie zbioru
3. zbiór niezmienny (FrozenSet)
4. operacje na zbiorach

Zbiór (Set)

Kolekcja unikalnych elementów, obsługująca działania odpowiadające matematycznej teorii zbiorów.

Jest nieuporządkowany i niehaszowalny.

Tworzenie zbioru

set(‘aabbccddee’)
{'a', 'b', 'c', 'd', 'e'}

a = set(['Ala', 'ma', 'kota', 'Ala'])
{'Ala', 'kota', 'ma'}

brak kolejności

a[1]
TypeError: 'set' object does not support indexing

Zbiór > lista

```
a = set([5, 2, 2, 7, 1, 1])
{7, 2, 5, 1}
b = list(a)
[7, 2, 5, 1]
b.sort()
[1, 2, 5, 7]
sorted(a)
[1, 2, 5, 7]
```

Zbiór jest niehaszowalny

```
a = set([1, 2, 3])
b = {a: 123}
TypeError: unhashable type: 'set'
```

Modyfikowanie zbioru

```
a = set([1, 2, 3])

# dodawanie elementu
a.add(4)
```
{1, 2, 3, 4}

```
# łączenie zbiorów
a.update(set([5, 6]))
```
{1, 2, 3, 4, 5, 6}

Modyfikowanie zbioru

```
# kopiowanie zbioru (shallow copy)
a.copy()
{1, 2, 3, 4, 5, 6}

# usuwanie elementu
a.remove(1)
{2, 3, 4, 5, 6}

# czyszczenie zbioru
a.clear()
set()
```

Zbiór niezmienny (FrozenSet)

```
a = frozenset()
d = {a: ‘testadmin123’}
{frozenset(): 'testadmin123'}
d[a]
'testadmin123'
```

Operacje na zbiorach (wyrażenia i metody)

```
# różnica zbiorów
a = set([1, 2, 3, 4, 5])
b = set([4, 5, 6, 7])
a - b
a.difference(b)
```

```
{1, 2, 3}
```

Operacje na zbiorach (wyrażenia i metody)

```
# symetryczna różnica zbiorów
a = set([1, 2, 3, 4, 5])
b = set([4, 5, 6, 7])
a ^ b
a.symmetric_difference(b)
```

```
{1, 2, 3, 6, 7}
```

Operacje na zbiorach (wyrażenia i metody)

```
# suma zbiorów
a = set([1, 2, 3, 4, 5])
b = set([4, 5, 6, 7])
a | b
a.union(b)
```

```
{1, 2, 3, 4, 5, 6, 7}
```

Operacje na zbiorach (wyrażenia i metody)

```
# nadzbiór
a = set([1, 2, 3, 4, 5])
b = set([2, 3, 4])
a > b
a.issuperset(b)
True
b > a
False
```

Operacje na zbiorach (wyrażenia i metody)

```
# podzbiór
a = set([1, 2, 3, 4, 5])
b = set([2, 3, 4])
a < b
a.issubset(b)
False
b < a
True
```

Iterator

1. lista składana
2. range(), map(), zip(), filter()
3. własny iterator

Iterator

Jest to obiekt przechowujący fizyczną sekwencję lub
zwracający jeden wynik naraz w kontekście narzędzia iteracyjnego (for).

Obiekt, by być iteratorem musi zawierać przynajmniej 2 metody:

1. __iter__()
2. __next__()

Wywołaniami iteratora można sterować ręcznie (__next__()).

Lista składana

Tworzenie listy jedna linijką kodu, czytelniejsze, szybsze działanie, przykład:

```
# sposób na piechotę’
txt = 'Lorem ipsum dolor sit amet consectetur adipiscing elit'
words_len = []
for word in txt.split():
    words_len.append(len(word))
```

```
[5, 5, 5, 3, 4, 11, 10, 4]
```

Lista składana

```
# użycie listy składanej
txt = 'Lorem ipsum dolor sit amet consectetur adipiscing elit'
words_len = [len(word) for word in txt.split()]
```

```
[5, 5, 5, 3, 4, 11, 10, 4]
```

range()

```
# syntax: range(start, stop, step)

[x for x in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

[x for x in range(4, 10, 2)]
[4, 6, 8]


[x for x in range(10, 4, -1)]
[10, 9, 8, 7, 6, 5]
```

map()

```
# syntax: map(function, iterable)

data = (1, 2, 3, 4)
map(lambda x: x**2, data)
```

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

zip()

```
zip agreguje elementy w formie iteratora krotek
# syntax: zip(*iterables)
```

kierowca_a = (15.25, 16.22, 17.33)
kierowca_b = (14.15, 15.86, 17.10)
kierowca_c = (12.10, 13.20, 12.70)

zestawienie_okrazen = zip(kierowca_a, kierowca_b, kierowca_c)

[(15.25, 14.15, 12.1), (16.22, 15.86, 13.2), (17.33, 17.1, 12.7)]

filter()

```
# syntax: filter(function, iterable)

data = range(-3, 3)

less_than_zero = filter(lambda x: x < 0, data)
```

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

Własny iterator

```
class Fib:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        self.a = 0
        self.b = 1
        return self

    def __next__(self):
        fib = self.a
        if fib > self.max:
            raise StopIteration
        self.a, self.b = self.b, self.a + self.b
        return fib

[x for x in Fib(5)]
[0, 1, 1, 2, 3, 5]
```

Generator

1. funkcja generatora
2. wyrażenie generatora

Generator

Mechanizm do zwracania wyników cząstkowych:

- implementuje protokół iteracyjny (__next__, StopIteration)
- zwraca wynik cząstkowy na żądanie (lazy evaluation)
- oszczędza pamięć
- obciążenie procesora jest rozłożone w czasie

funkcja generatora

- zwykła funkcja
- do zwracania wyników yeld zamiast return
- yeld kompilowany jest do obiektu obsługującego interfejs iteracji
- można użyć return do zakończenia działania generatora

funkcja generatora

```
def sum_inc(max=5):
    a = 0;
    for i in range(max):
        a += i;
        yield a


[x for x in sum_inc()]
```

```
[0, 1, 3, 6, 10]
```

wyrażenie generatora

- konstrukcja podobna do listy składanej
- zamiast [ ] używa się ( )
- w odróżnieniu od listy składanej nie buduje całego wyniku w pamięci, zwraca obiekt obsługujący protokół iteratorów

wyrażenie generatora

```
# lista składana

[x for x in range(4)]
```

```
[0, 1, 2, 3]
```

```
# wyrażenie generatora

(x for x in range(4))
```

```
<generator object <genexpr> at 0x7f2a64153cd0>
```

dzonybt@gmail.com