Politechnika Świętokrzyska

Laboratorium

Programowanie w języku Python 2

Ćwiczenie 3

Wyrażenie generatorowe
Dodatkowe iteratory
Deskryptory

Cel ćwiczenia

Celem ćwiczenia jest zapoznanie studentów z narzędziami programistycznymi i bibliotekami języka Python.

dr inż Robert Kazała

Wyrażenie generatorowe

Wyrażenie generatora jest jak funkcja generatora bez funkcji

Przykład 1

```
unique_characters = {'E', 'D', 'M', 'O', 'N', 'S', 'R', 'Y'}
gen = (ord(c) for c in unique_characters)
gen
next(gen)
next(gen)
tuple(ord(c) for c in unique_characters)
```

Wyrażenie generatora jest jak anonimowa funkcja, która zwraca wartości. Samo wyrażenie wygląda jak comprehension list, ale jest obudowane nawiasami okrągłymi zamiast nawiasów kwadratowych. Wyrażenie generatora zwraca iterator Użycie generator expression zamiast list comprehension może zaoszczędzić zarówno C P U, jak i R AM. Jeśli lista jest budowana tylko po to, aby ją przekazać do innego obiektu (np. Przekazując do krotki() lub do set()), lepiej użyć zamiast tego wyrażenia generatorowego.

Funkcje tworzące iteratory - itertools

Ten moduł implementuje wiele bloków iteratorów inspirowanych konstrukcjami z APL, Haskell i SML. Każdy został przekształcony został do formy odpowiedniej dla Pythona. Moduł standaryzuje podstawowy zestaw szybkich, wydajnych pamięci narzędzi, które są użyteczne same lub w połączeniu. Razem tworzą „algebrę iteratora", która umożliwia zwięzłe i wydajne konstruowanie specjalistycznych narzędzi w czystym języku Python.

Iteratory nieskończone

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Iteratory kończące się na najkrótszej sekwencji wejściowej

Iterator	Arguments	Results	Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`
`chain()`	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
`dropwhile()`	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`
`filterfalse()`	pred, seq	elements of seq where pred(elem) is false	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
`groupby()`	iterable[, key]	sub-iterators grouped by value of key(v)	
`islice()`	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
`starmap()`	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
`takewhile()`	pred, seq	seq[0], seq[1], until pred fails	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
`tee()`	it, n	it1, it2, ... itn splits one iterator into n	
`zip_longest()`	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Iteratory kombinatoryczne

Iterator	Arguments	Results
`product()`	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements

Przykład 2

Obliczanie permutacji

```
import itertools
perms = itertools.permutations([1, 2, 3], 2)
next(perms)
next(perms)
next(perms)
next(perms)
next(perms)
next(perms)
next(perms)
```

Przykład 3

Obliczanie permutacji

```
import itertools
perms = itertools.permutations('ABC', 3)
next(perms)
next(perms)
next(perms)
next(perms)
next(perms)
next(perms)
next(perms)

list(itertools.permutations('ABC', 3))
```

Przykład 4

Obliczanie permutacji

```
import itertools
list(itertools.product('ABC', '123'))
list(itertools.combinations('ABC', 2))
```

Przykład 5

Obliczanie permutacji

```
names = list(open('examples/favorite-people.txt', encoding='utf-
8'))
names
names = [name.rstrip() for name in names]
names
```

```
names = sorted(names)
names

names = sorted(names, key=len)
names

import itertools
groups = itertools.groupby(names, len)
groups
list(groups)
groups = itertools.groupby(names, len)
for name_length, name_iter in groups:
     print('Names with {0:d} letters:'.format(name_length))
     for name in name_iter:
     print(name)
```

Biblioteka dodatkowych iteratorów - more-itertools

Dodatkowa biblioteka iteratorów rozszerzająca możliwości itertools.

Grouping	chunked, ichunked, sliced, distribute, divide, split_at, split_before, split_after, split_into, split_when, bucket, unzip, grouper, partition
Lookahead and lookback	spy, peekable, seekable
Windowing	windowed, substrings, substrings_indexes, stagger, pairwise
Augmenting	count_cycle, intersperse, padded, repeat_last, adjacent, groupby_transform, padnone, ncycles
Combining	collapse, sort_together, interleave, interleave_longest, zip_offset, dotproduct, flatten, roundrobin, prepend
Summarizing	ilen, unique_to_each, sample, consecutive_groups, run_length, map_reduce, exactly_n, all_equal, first_true, quantify
Selecting	islice_extended, first, last, one, only, strip, lstrip, rstrip, filter_except map_except nth_or_last, nth, take, tail, unique_everseen, unique_justseen
Combinatorics	distinct_permutations, distinct_combinations, circular_shifts, partitions, set_partitions, powerset, random_product, random_permutation, random_combination, random_combination_with_replacement, nth_combination
Wrapping	always_iterable, always_reversible, consumer, with_iter, iter_except
Others	locate, rlocate, replace, numeric_range, side_effect, iterate, difference, make_decorator, SequenceView, time_limited, consume, tabulate, repeatfunc

Deskryptory

Deskryptor to klasa, której można użyć do wywołania metody z prostym dostępem do atrybutów, Deskryptor implementuje co najmniej jedną z tych trzech metod:
__get __ (),
__set __ ()
__delete __ ().
Każda z tych metod ma listę niezbędnych parametrów i każdy z nich jest wywoływany przez inny rodzaj dostępu do atrybutu reprezentowanego przez deskryptor.

Przykład 6

```
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print('Retrieving', self.name)
        return self.val

    def __set__(self, obj, val):
        print('Updating', self.name)
        self.val = val

class MyClass(object):
     x = RevealAccess(10, 'var "x"')
     y = 5

m = MyClass()
m.x
m.x = 20
m.x
m.y
```

Przykład 7

```
class Descriptor(object):

     def __init__(self, name =''):
          self.name = name

     def __get__(self, obj, objtype):
          return "{}for{}".format(self.name, self.name)
```

```
     def __set__(self, obj, name):
          if isinstance(name, str):
               self.name = name
          else:
               raise TypeError("Name should be string")

class GFG(object):
     name = Descriptor()

g = GFG()
g.name = "Computer"
print(g.name)
```

Literatura

Zadania

1. Uruchomić, przeanalizować i zmodyfikować działanie wszystkich przykładów z instrukcji.
2. Na własnych przykładach zaprezentować tworzenie i wykorzystanie wyrażenia generatorowego.
3. Na własnych przykładach zaprezentować działanie wybranych iteratorów z biblioteki **itertools**.
4. Na własnych przykładach zaprezentować działanie wybranych iteratorów z biblioteki **more-itertools.**
5. Na samodzielnie zdefiniowanych klasach zaprezentować implementację i wykorzystanie deskryptorów.