Itertools Module

Python 2.x 2.3–2.7 , Python 3.x 3.0–3.6

Examples

itertools.dropwhile

itertools.dropwhile enables you to take items from a sequence after a condition first becomes False .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.dropwhile(is_even, lst))

print(result)
```

This outputs [13, 14, 22, 23, 44] .

(*This example is same as the example for takewhile but using dropwhile* .)

Note that, the first number that violates the predicate (i.e.: the function returning a Boolean value) is_even is, 13 . All the elements before that, are discarded.

The **output produced** by dropwhile is similar to the output generated from the code below.

```
def dropwhile(predicate, iterable):
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

The concatenation of results produced by takewhile and dropwhile produces the original iterable.

result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))

Combinations method in Itertools Module

itertools.combinations will return a generator of the *k* -combination sequence of a list.

In other words: It will return a generator of tuples of all the possible k-wise combinations of the input list.

For Example:

If you have a list:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 2))
print b
```

Output:

[(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)]

The above output is a generator converted to a list of tuples of all the possible *pair* -wise combinations of the input list a

You can also find all the 3-combinations:

```
a = [1,2,3,4,5]
b = list(itertools.combinations(a, 3))
print b
```

Output:

```
[(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4),
 (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5),
 (2, 4, 5), (3, 4, 5)]
```

Grouping items from an iterable object using a function

Start with an iterable which needs to be grouped

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
```

Generate the grouped generator, grouping by the second element in each tuple:

```
def testGroupBy(lst):
    groups = itertools.groupby(lst, key=lambda x: x[1])
    for key, group in groups:
        print(key, list(group))

testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
```

Only groups of consecutive elements are grouped. You may need to sort by the same key before calling groupby For E.g, (Last element is changed)

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 5, 6)]
testGroupBy(lst)

# 5 [('a', 5, 6)]
# 2 [('b', 2, 4), ('a', 2, 5)]
# 5 [('c', 5, 6)]
```

The group returned by groupby is an iterator that will be invalid before next iteration. E.g the following will not work if you want the groups to be sorted by key. Group 5 is empty below because when group 2 is fetched it invalidates 5

```
lst = [("a", 5, 6), ("b", 2, 4), ("a", 2, 5), ("c", 2, 6)]
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted(groups):
    print(key, list(group))

# 2 [('c', 2, 6)]
# 5 []
```

To correctly do sorting, create a list from the iterator before sorting

```
groups = itertools.groupby(lst, key=lambda x: x[1])
for key, group in sorted((key, list(group)) for key, group in groups):
    print(key, list(group))

# 2 [('b', 2, 4), ('a', 2, 5), ('c', 2, 6)]
# 5 [('a', 5, 6)]
```

Cycle through elements in an iterator

`cycle` is an infinite iterator.

```
>>> import itertools as it
>>> it.cycle('ABCD')
A B C D A B C D A B C D ...
```

Therefore, take care to give boundaries when using this to avoid an infinite loop. Example:

```
>>> # Iterate over each element in cycle for a fixed range
>>> cycle_iterator = it.cycle('abc123')
>>> [next(cycle_iterator) for i in range(0, 10)]
['a', 'b', 'c', '1', '2', '3', 'a', 'b', 'c', '1']
```

Get an accumulated sum of numbers in an iterable

Python 3.x ≥3.2

`accumulate` yields a cumulative sum (or product) of numbers.

```
>>> import itertools as it
>>> import operator

>>> list(it.accumulate([1,2,3,4,5]))
[1, 3, 6, 10, 15]
```

```
>>> list(it.accumulate([1,2,3,4,5], func=operator.mul))
[1, 2, 6, 24, 120]
```

itertools.count

Introduction:

This simple function generates infinite series of numbers. For example...

```
for number in itertools.count():
    if number > 20:
        break
    print(number)
```

Note that we must break or it prints forever!

Output:

```
0
1
2
3
4
5
6
7
8
9
10
```

Arguments:

`count()` takes two arguments, `start` and `step` :

```
for number in itertools.count(start=10, step=4):
    print(number)
    if number > 20:
        break
```

Output:

```
10
14
18
22
```

itertools.permutations

`itertools.permutations` returns a generator with successive r-length permutations of elements in the iterable.

```
a = [1,2,3]
list(itertools.permutations(a))
# [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]

list(itertools.permutations(a, 2))
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

if the list `a` has duplicate elements, the resulting permutations will have duplicate elements, you can use `set` to get unique permutations:

```
a = [1,2,1]
list(itertools.permutations(a))
# [(1, 2, 1), (1, 1, 2), (2, 1, 1), (2, 1, 1), (1, 1, 2), (1, 2, 1)]

set(itertools.permutations(a))
# {(1, 1, 2), (1, 2, 1), (2, 1, 1)}
```

itertools.product

This function lets you iterate over the Cartesian product of a list of iterables.

For example,

```
for x, y in itertools.product(xrange(10), xrange(10)):
    print x, y
```

is equivalent to

```
for x in xrange(10):
    for y in xrange(10):
        print x, y
```

Like all python functions that accept a variable number of arguments, we can pass a list to itertools.product for unpacking, with the * operator.

Thus,

```
its = [xrange(10)] * 2
for x,y in itertools.product(*its):
    print x, y
```

produces the same results as both of the previous examples.

```
>>> from itertools import product
>>> a=[1,2,3,4]
>>> b=['a','b','c']
>>> product(a,b)
<itertools.product object at 0x0000000002712F78>
>>> for i in product(a,b):
...     print i
...
(1, 'a')
(1, 'b')
(1, 'c')
(2, 'a')
(2, 'b')
(2, 'c')
(3, 'a')
(3, 'b')
(3, 'c')
(4, 'a')
(4, 'b')
(4, 'c')
```

---

itertools.repeat

Repeat something n times:

```
>>> import itertools
>>> for i in itertools.repeat('over-and-over', 3):
...    print(i)
over-and-over
over-and-over
over-and-over
```

---

itertools.takewhile

itertools.takewhile enables you to take items from a sequence until a condition first becomes `False` .

```
def is_even(x):
    return x % 2 == 0


lst = [0, 2, 4, 12, 18, 13, 14, 22, 23, 44]
result = list(itertools.takewhile(is_even, lst))

print(result)
```

This outputs `[0, 2, 4, 12, 18]` .

Note that, the first number that violates the predicate (i.e.: the function returning a Boolean value) `is_even` is, `13` . Once `takewhile` encounters a value that produces `False` for the given predicate, it breaks out.

The **output produced** by `takewhile` is similar to the output generated from the code below.

```
def takewhile(predicate, iterable):
```

```
    for x in iterable:
        if predicate(x):
            yield x
        else:
            break
```

Note: The concatenation of results produced by `takewhile` and `dropwhile` produces the original iterable.

`result = list(itertools.takewhile(is_even, lst)) + list(itertools.dropwhile(is_even, lst))`

Take a slice of a generator

Itertools "islice" allows you to slice a generator:

```
results = fetch_paged_results()  # returns a generator
limit = 20  # Only want the first 20 results
for data in itertools.islice(results, limit):
    print(data)
```

Normally you cannot slice a generator:

```
def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in gen()[:3]:
    print(part)
```

Will give

```
Traceback (most recent call last):
  File "gen.py", line 6, in <module>
    for part in gen()[:3]:
TypeError: 'generator' object is not subscriptable
```

However, this works:

```
import itertools

def gen():
    n = 0
    while n < 20:
        n += 1
        yield n

for part in itertools.islice(gen(), 3):
    print(part)
```

Note that like a regular slice, you can also use `start` , `stop` and `step` arguments:

```
itertools.islice(iterable, 1, 30, 3)
```

Zipping two iterators until they are both exhausted

Similar to the built-in function `zip()` , `itertools.zip_longest` will continue iterating beyond the end of the shorter of two iterables.

```
from itertools import zip_longest
a = [i for i in range(5)] # Length is 5
b = ['a', 'b', 'c', 'd', 'e', 'f', 'g'] # Length is 7
for i in zip_longest(a, b):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

An optional `fillvalue` argument can be passed (defaults to `''`) like so:

```
for i in zip_longest(a, b, fillvalue='Hogwash!'):
    x, y = i  # Note that zip longest returns the values as a tuple
    print(x, y)
```

In Python 2.6 and 2.7, this function is called `itertools.izip_longest` .

Chaining multiple iterators together

Use itertools.chain to create a single generator which will yield the values from several generators in sequence.

```
from itertools import chain
a = (x for x in ['1', '2', '3', '4'])
b = (x for x in ['x', 'y', 'z'])
' '.join(chain(a, b))
```

Results in:

```
'1 2 3 4 x y z'
```

As an alternate constructor, you can use the classmethod chain.from_iterable which takes as its single parameter an iterable of iterables. To get the same result as above:

```
' '.join(chain.from_iterable([a,b])
```

While chain can take an arbitrary number of arguments, chain.from_iterable is the only way to chain an *infinite* number of iterables.

Syntax

```
import itertools
```

Parameters

Remarks