itertools – Iterator functions for efficient looping

Purpose:	The itertools module includes a set of functions for working with iterable (sequence-like) data sets.
Available In:	2.3

The functions provided are inspired by similar features of the “lazy functional programming language” Haskell and SML. They are intended to be fast and use memory efficiently, but also to be hooked together to express more complicated iteration-based algorithms.

Iterator-based code may be preferred over code which uses lists for several reasons. Since data is not produced from the iterator until it is needed, all of the data is not stored in memory at the same time. Reducing memory usage can reduce swapping and other side-effects of large data sets, increasing performance.

Merging and Splitting Iterators

The `chain()` function takes several iterators as arguments and returns a single iterator that produces the contents of all of them as though they came from a single sequence.

```
from itertools import *

for i in chain([1, 2, 3], ['a', 'b', 'c']):
    print i
$ python itertools_chain.py

1
2
3
a
b
c
```

`izip()` returns an iterator that combines the elements of several iterators into tuples. It works like the built-in function `zip()`, except that it returns an iterator instead of a list.

```
from itertools import *

for i in izip([1, 2, 3], ['a', 'b', 'c']):
    print i
$ python itertools_izip.py

(1, 'a')
(2, 'b')
```

```
(3, 'c')
```

The `islice()` function returns an iterator which returns selected items from the input iterator, by index. It takes the same arguments as the slice operator for lists: start, stop, and step. The start and step arguments are optional.

```
from itertools import *

print 'Stop at 5:'
for i in islice(count(), 5):
    print i

print 'Start at 5, Stop at 10:'
for i in islice(count(), 5, 10):
    print i

print 'By tens to 100:'
for i in islice(count(), 0, 100, 10):
    print i
$ python itertools_islice.py

Stop at 5:
0
1
2
3
4
Start at 5, Stop at 10:
5
6
7
8
9
By tens to 100:
0
10
20
30
40
50
60
70
80
90
```

The `tee()` function returns several independent iterators (defaults to 2) based on a single original input. It has semantics similar to the Unix tee utility, which repeats the values it reads from its input and writes them to a named file and standard output.

```
from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)
```

```
for i in i1:
    print 'i1:', i
for i in i2:
    print 'i2:', i
$ python itertools_tee.py

i1: 0
i1: 1
i1: 2
i1: 3
i1: 4
i2: 0
i2: 1
i2: 2
i2: 3
i2: 4
```

Since the new iterators created by `tee()` share the input, you should not use the original iterator any more. If you do consume values from the original input, the new iterators will not produce those values:

```
from itertools import *

r = islice(count(), 5)
i1, i2 = tee(r)

for i in r:
    print 'r:', i
    if i > 1:
        break

for i in i1:
    print 'i1:', i
for i in i2:
    print 'i2:', i
$ python itertools_tee_error.py

r: 0
r: 1
r: 2
i1: 3
i1: 4
i2: 3
i2: 4
```

Converting Inputs

The `imap()` function returns an iterator that calls a function on the values in the input iterators, and returns the results. It works like the built-in `map()`, except that it stops when any input iterator is exhausted (instead of inserting `None` values to completely consume all of the inputs).

In the first example, the lambda function multiplies the input values by 2. In a second example, the lambda function multiplies 2 arguments, taken from separate iterators, and returns a tuple with the original arguments and the computed value.

```
from itertools import *

print 'Doubles:'
for i in imap(lambda x:2*x, xrange(5)):
    print i

print 'Multiples:'
for i in imap(lambda x,y:(x, y, x*y), xrange(5), xrange(5,10)):
    print '%d * %d = %d' % i
$ python itertools_imap.py

Doubles:
0
2
4
6
8
Multiples:
0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

The `starmap()` function is similar to `imap()`, but instead of constructing a tuple from multiple iterators it splits up the items in a single iterator as arguments to the mapping function using the `*` syntax. Where the mapping function to imap() is called f(i1, i2), the mapping function to starmap() is called `f(*i)`.

```
from itertools import *

values = [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
for i in starmap(lambda x,y:(x, y, x*y), values):
    print '%d * %d = %d' % i
$ python itertools_starmap.py

0 * 5 = 0
1 * 6 = 6
2 * 7 = 14
3 * 8 = 24
4 * 9 = 36
```

Producing New Values

The `count()` function returns an interator that produces consecutive integers, indefinitely. The first number can be passed as an argument, the default is zero. There is no upper bound argument (see the built-in `xrange()` for more control over the result set). In this example, the iteration stops because the list argument is consumed.

```
from itertools import *

for i in izip(count(1), ['a', 'b', 'c']):
    print i
$ python itertools_count.py

(1, 'a')
(2, 'b')
(3, 'c')
```

The `cycle()` function returns an iterator that repeats the contents of the arguments it is given indefinitely. Since it has to remember the entire contents of the input iterator, it may consume quite a bit of memory if the iterator is long. In this example, a counter variable is used to break out of the loop after a few cycles.

```
from itertools import *

i = 0
for item in cycle(['a', 'b', 'c']):
    i += 1
    if i == 10:
        break
    print (i, item)
$ python itertools_cycle.py

(1, 'a')
(2, 'b')
(3, 'c')
(4, 'a')
(5, 'b')
(6, 'c')
(7, 'a')
(8, 'b')
(9, 'c')
```

The `repeat()` function returns an iterator that produces the same value each time it is accessed. It keeps going forever, unless the optional times argument is provided to limit it.

```
from itertools import *

for i in repeat('over-and-over', 5):
    print i
$ python itertools_repeat.py

over-and-over
over-and-over
over-and-over
over-and-over
over-and-over
```

It is useful to combine `repeat()` with `izip()` or `imap()` when invariant values need to be included with the values from the other iterators.

```
from itertools import *

for i, s in izip(count(), repeat('over-and-over', 5)):
    print i, s
$ python itertools_repeat_izip.py

0 over-and-over
1 over-and-over
2 over-and-over
3 over-and-over
4 over-and-over
from itertools import *

for i in imap(lambda x,y:(x, y, x*y), repeat(2), xrange(5)):
    print '%d * %d = %d' % i
$ python itertools_repeat_imap.py

2 * 0 = 0
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
```

Filtering

The `dropwhile()` function returns an iterator that returns elements of the input iterator after a condition becomes false for the first time. It does not filter every item of the input; after the condition is false the first time, all of the remaining items in the input are returned.

```
from itertools import *

def should_drop(x):
    print 'Testing:', x
    return (x<1)

for i in dropwhile(should_drop, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
$ python itertools_dropwhile.py

Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Yielding: 2
Yielding: 3
Yielding: 4
Yielding: 1
Yielding: -2
```

The opposite of `dropwhile()`, `takewhile()` returns an iterator that returns items from the input iterator as long as the test function returns true.

```
from itertools import *

def should_take(x):
    print 'Testing:', x
    return (x<2)

for i in takewhile(should_take, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
$ python itertools_takewhile.py

Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Yielding: 1
Testing: 2
```

`ifilter()` returns an iterator that works like the built-in `filter()` does for lists, including only items for which the test function returns true. It is different from `dropwhile()` in that every item is tested before it is returned.

```
from itertools import *

def check_item(x):
    print 'Testing:', x
    return (x<1)

for i in ifilter(check_item, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
$ python itertools_ifilter.py

Testing: -1
Yielding: -1
Testing: 0
Yielding: 0
Testing: 1
Testing: 2
Testing: 3
Testing: 4
Testing: 1
Testing: -2
Yielding: -2
```

The opposite of `ifilter()`, `ifilterfalse()` returns an iterator that includes only items where the test function returns false.

```
from itertools import *

def check_item(x):
    print 'Testing:', x
    return (x<1)

for i in ifilterfalse(check_item, [ -1, 0, 1, 2, 3, 4, 1, -2 ]):
    print 'Yielding:', i
```

```
$ python itertools_ifilterfalse.py

Testing: -1
Testing: 0
Testing: 1
Yielding: 1
Testing: 2
Yielding: 2
Testing: 3
Yielding: 3
Testing: 4
Yielding: 4
Testing: 1
Yielding: 1
Testing: -2
```

Grouping Data

The `groupby()` function returns an iterator that produces sets of values grouped by a common key.

This example from the standard library documentation shows how to group keys in a dictionary which have the same value:

```
from itertools import *
from operator import itemgetter

d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
di = sorted(d.iteritems(), key=itemgetter(1))
for k, g in groupby(di, key=itemgetter(1)):
    print k, map(itemgetter(0), g)
$ python itertools_groupby.py

1 ['a', 'c', 'e']
2 ['b', 'd', 'f']
3 ['g']
```

This more complicated example illustrates grouping related values based on some attribute. Notice that the input sequence needs to be sorted on the key in order for the groupings to work out as expected.

```
from itertools import *

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Point(%s, %s)' % (self.x, self.y)
    def __cmp__(self, other):
        return cmp((self.x, self.y), (other.x, other.y))

# Create a dataset of Point instances
```

```
data = list(imap(Point,
                 cycle(islice(count(), 3)),
                 islice(count(), 10),
                 )
            )
print 'Data:', data
print

# Try to group the unsorted data based on X values
print 'Grouped, unsorted:'
for k, g in groupby(data, lambda o:o.x):
    print k, list(g)
print

# Sort the data
data.sort()
print 'Sorted:', data
print

# Group the sorted data based on X values
print 'Grouped, sorted:'
for k, g in groupby(data, lambda o:o.x):
    print k, list(g)
print
$ python itertools_groupby_seq.py

Data: [Point(0, 0), Point(1, 1), Point(2, 2), Point(0, 3), Point(1, 4),
Point(2, 5), Point(0, 6), Point(1, 7), Point(2, 8), Point(0, 9)]

Grouped, unsorted:
0 [Point(0, 0)]
1 [Point(1, 1)]
2 [Point(2, 2)]
0 [Point(0, 3)]
1 [Point(1, 4)]
2 [Point(2, 5)]
0 [Point(0, 6)]
1 [Point(1, 7)]
2 [Point(2, 8)]
0 [Point(0, 9)]

Sorted: [Point(0, 0), Point(0, 3), Point(0, 6), Point(0, 9), Point(1,
1), Point(1, 4), Point(1, 7), Point(2, 2), Point(2, 5), Point(2, 8)]

Grouped, sorted:
0 [Point(0, 0), Point(0, 3), Point(0, 6), Point(0, 9)]
1 [Point(1, 1), Point(1, 4), Point(1, 7)]
2 [Point(2, 2), Point(2, 5), Point(2, 8)]
```