Python itertools

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

October 26, 2011

Outline

`itertools`

- New in version 2.3.
- This module implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.
- The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an iterator algebra making it possible to construct specialized tools succinctly and efficiently in pure Python.

Outline

itertools.chain(*iterables)

Description

- Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted.
- Used for treating consecutive sequences as a single sequence.

Example

```
>>> import itertools as it
>>> list(it.chain('abc', 'def'))
['a', 'b', 'c', 'd', 'e', 'f']
>>> list(it.chain(['abc', 'def']))
['abc', 'def']
>>> list(it.chain(['abc'], ['def']))
['abc', 'def']
>>>
```

itertools.chain(*iterables)

Equivalent Python code

```
def chain(*iterables):
    for it in iterables:
        for element in it:
            yield element
```

itertools.count([n])

Description

- Make an iterator that returns consecutive integers starting with `n`.
- If not specified `n` defaults to zero.
- Often used as an argument to `imap()` to generate consecutive data points. Also, used with izip() to add sequence numbers.

Example

```
import itertools as it
for i in it.count(2):
    if i > 8:
        break
    print i,
# print 2 3 4 5 6 7 8
```

itertools.count([n])

Equivalent python code

```
def count(n=0):
    while True:
        yield n
        n += 1
```

itertools.cycle(iterable)

Description

- Make an iterator returning elements from the iterable and saving a copy of each. When the iterable is exhausted, return elements from the saved copy
- This member of the toolkit may require significant auxiliary storage (depending on the length of the iterable).

Example

```
# cycle('ABCD') --> A B C D A B C D A B C D ...
```

itertools.cycle(iterable)

Equivalent python code

```
def cycle(iterable):
    saved = []
    for element in iterable:
        yield element
        saved.append(element)
    while saved:
        for element in saved:
              yield element
```

itertools.dropwhile(predicate, iterable)

Description

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element. Note, the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time.

itertools.dropwhile(predicate, iterable)

Equivalent python code

```
def dropwhile(predicate, iterable):
    # dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1
    iterable = iter(iterable)
    for x in iterable:
        if not predicate(x):
            yield x
            break
    for x in iterable:
        yield x
```

itertools.groupby(iterable[, key])

Description

Make an iterator that returns consecutive keys and groups from the `iterable`. The `key` is a function computing a key value for each element. If not specified or is `None`, `key` defaults to an identity function and returns the element unchanged. Generally, the `iterable` needs to already be sorted on the same key function.

itertools.groupby(iterable[, key])

```
class groupby(object):
    # [k for k, g in groupby('AAAABBBCCDAABBB')] --> A B C D A B
    # [list(g) for k, g in groupby('AAAABBBCCD')] --> AAAA BBB CC D
    def __init__(self, iterable, key=None):
        if key is None:
            key = lambda x: x
        self.keyfunc = key
        self.it = iter(iterable)
        self.tgtkey = self.currkey = self.currvalue = object()
    def __iter__(self):
        return self
    def next(self):
        while self.currkey == self.tgtkey:
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
        self.tgtkey = self.currkey
        return (self.currkey, self._grouper(self.tgtkey))
    def _grouper(self, tgtkey):
        while self.currkey == tgtkey:
            yield self.currvalue
            self.currvalue = next(self.it)    # Exit on StopIteration
            self.currkey = self.keyfunc(self.currvalue)
```

itertools.ifilter, itertools.ifilterfalse

Equivalent python code

```
def ifilter(predicate, iterable):
    # ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9
    if predicate is None:
        predicate = bool
    for x in iterable:
        if predicate(x):
            yield x
```

Equivalent python code

```
def ifilterfalse(predicate, iterable):
    # ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8
    if predicate is None:
        predicate = bool
    for x in iterable:
        if not predicate(x):
            yield x
```

itertools.imap(function, *iterables)

Description

- Make an iterator that computes the function using arguments from each of the iterables. If function is set to None, then `imap()` returns the arguments as a tuple.
- Like `map()` but stops when the shortest iterable is exhausted instead of filling in None for shorter iterables. The reason for the difference is that infinite iterator arguments are typically an error for `map()` (because the output is fully evaluated) but represent a common and useful way of supplying arguments to `imap()`.

Example

```
>>> import itertools as it
>>> it.imap(pow, (2,3,10), (5,2,3))
<itertools.imap object at 0xb7c682ac>
>>> list(it.imap(pow, (2,3,10), (5,2,3)))
[32, 9, 1000]
```

itertools.imap(function, *iterables)

Equivalent python code

```
def imap(function, *iterables):
    iterables = map(iter, iterables)
    while True:
        args = [next(it) for it in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)
```

itertools.imap(function, *iterables)

Example

```
>>> import itertools as it
>>> import operator
>>>
>>> it.imap(operator.add, [1, 'a', 1+2j, [1]], [2, 'b', 3+4j ,[2]])
<itertools.imap object at 0xb7ca2a0c>
>>> list(it.imap(operator.add, [1, 'a', 1+2j, [1]], [2, 'b', 3+4j ,
[3, 'ab', (4+6j), [1, 2]]
>>>
>>> list(it.imap(pow, xrange(10), it.count()))
[1, 1, 4, 27, 256, 3125, 46656, 823543, 16777216, 387420489]
>>> list(it.imap(operator.add, xrange(10), it.count()))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>>
```

itertools.islice(iterable[,start],stop[,step])

Description

- Make an iterator that returns selected elements from the iterable. If start is non-zero, then elements from the iterable are skipped until start is reached. Afterward, elements are returned consecutively unless step is set higher than one which results in items being skipped. If stop is None, then iteration continues until the iterator is exhausted, if at all; otherwise, it stops at the specified position.
- Unlike regular slicing, `islice()` does not support negative values for start, stop, or step. Can be used to extract related fields from data where the internal structure has been flattened (for example, a multi-line report may list a name field on every third line).

itertools.islice(iterable[,start],stop[,step])

Example

```
>>> import itertools as it
>>>
>>> list(it.islice('abcdefg', 2))
['a', 'b']
>>>
>>> list(it.islice('abcdefg', 2, 4))
['c', 'd']
>>>
>>> list(it.islice('abcdefg', 2, None))
['c', 'd', 'e', 'f', 'g']
>>>
>>> list(it.islice('abcdefg', 0, None, 2))
['a', 'c', 'e', 'g']
>>>
```

itertools.islice(iterable[,start],stop[,step])

Equivalent python code

```
def islice(iterable, *args):
    s = slice(*args)
    it = iter(xrange(s.start or 0,
                     s.stop  or  sys.maxint,
                     s.step  or 1))
    nexti = next(it)
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
            nexti = next(it)
```

itertools.izip(*iterables)

Description

- Make an iterator that aggregates elements from each of the iterables. Like `zip()` except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time.
- The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into *n*-length groups using `izip(*[iter(s)]*n)`.
- `izip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `izip_longest()` instead.

Example

```
>>> import itertools
>>> for x in itertools.izip('ABCD', 'xy'): print x
...
('A', 'x')
('B', 'y')
>>>
```

itertools.izip(*iterables)

Equivalent python code

```
def izip(*iterables):
    # izip('ABCD', 'xy') --> Ax By
    iterables = map(iter, iterables)
    while iterables:
        yield tuple(map(next, iterables))
```

itertools.izip_longest(*iterables[, fillvalue])

Description

- Make an iterator that aggregates elements from each of the iterables. If the iterables are of uneven length, missing values are filled-in with fillvalue. Iteration continues until the longest iterable is exhausted.
- If one of the iterables is potentially infinite, then the `izip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`). If not specified, `fillvalue` defaults to `None`.

Example

```
>>> import itertools
>>> for x in itertools.izip_longest('ABCD', 'xy', fillvalue='-'): print x
...
('A', 'x')
('B', 'y')
('C', '-')
('D', '-')
>>>
```

itertools.izip_longest(*iterables[, fillvalue])

Equivalent python code

```
def izip_longest(*args, **kwds):
    # izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
    fillvalue = kwds.get('fillvalue')
    def sentinel(counter = ([fillvalue]*(len(args)-1)).pop):
        yield counter() # yields the fillvalue, or raises IndexErro
    fillers = repeat(fillvalue)
    iters = [chain(it, sentinel(), fillers) for it in args]
    try:
        for tup in izip(*iters):
            yield tup
    except IndexError:
        pass
```

itertools.tee(iterable[, n=2])

Description

- Return *n* independent iterators from a single iterable.
- Once `tee()` has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed.
- This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use `list()` instead of `tee()`.

Example

```
import itertools
r = itertools.islice(itertools.count(), 5)
i1, i2 = itertools.tee(r)
for i in i1:
    print i # 0 1 2 3 4
for i in i2:
    print i # 0 1 2 3 4
```

itertools.tee(iterable[, n=2])

Equivalent python code

```
def tee(iterable, n=2):
    it = iter(iterable)
    deques = [collections.deque() for i in range(n)]
    def gen(mydeque):
        while True:
            if not mydeque:         # when the local deque is empty
                newval = next(it) # fetch a new value and
                for d in deques:  # load it to all the deques
                    d.append(newval)
            yield mydeque.popleft()
    return tuple(gen(d) for d in deques)
```

itertools.tee(iterable[, n=2])

Another example

```
import itertools

def pairwise(iterable):
    a, b = itertools.tee(iterable)
    b.next()
    return itertools.izip(a, b)

r = itertools.islice(itertools.count(), 4)
for a, b in pairwise(r):
    # [ 0 , 1 ] [ 1 , 2 ] [ 2 , 3 ]
    print '[', a, ',', b, ']',
```

itertools.combinations

Description

- Return r length subsequences of elements from the input iterable.
- Combinations are emitted in lexicographic sort order. So, if the input iterable is sorted, the combination tuples will be produced in sorted order.
- Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

itertools.combinations

Equivalent python code

```
def combinations(iterable, r):
    # combinations('ABCD', 2) --> AB AC AD BC BD CD
    # combinations(range(4), 3) --> 012 013 023 123
    pool = tuple(iterable)
    n = len(pool)
    if r > n:
        return
    indices = range(r)
    yield tuple(pool[i] for i in indices)
    while True:
        for i in reversed(range(r)):
            if indices[i] != i + n - r:
                break
        else:
            return
        indices[i] += 1
        for j in range(i+1, r):
            indices[j] = indices[j-1] + 1
        yield tuple(pool[i] for i in indices)
```

Outline

Recipes 1/3

Toolset

```
def take(n, iterable):
    "Return first n items of the iterable as a list"
    return list(islice(iterable, n))

def enumerate(iterable, start=0):
    return izip(count(start), iterable)

def tabulate(function, start=0):
    "Return function(0), function(1), ..."
    return imap(function, count(start))

def consume(iterator, n):
    "Advance the iterator n-steps ahead. If n is none, consume entirely."
    collections.deque(islice(iterator, n), maxlen=0)

def nth(iterable, n, default=None):
    "Returns the nth item or a default value"
    return next(islice(iterable, n, None), default)
```

Recipes 2/3

Toolset

```
def quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.
    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

def ncycles(iterable, n):
    "Returns the sequence elements n times"
    return chain.from_iterable(repeat(iterable, n))

def dotproduct(vec1, vec2):
    return sum(imap(operator.mul, vec1, vec2))

def flatten(listOfLists):
    return list(chain.from_iterable(listOfLists))
```

Recipes 3/3

Toolset

```
def repeatfunc(func, times=None, *args):
    """Repeat calls to func with specified arguments.
    Example:  repeatfunc(random.random)
    """
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

def roundrobin(*iterables):
    "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
    # Recipe credited to George Sakkis
    pending = len(iterables)
    nexts = cycle(iter(it).next for it in iterables)
    while pending:
        try:
            for next in nexts:
                yield next()
        except StopIteration:
            pending -= 1
            nexts = cycle(islice(nexts, pending))
```