API Reference

New Routines

more_itertools.**chunked**(*iterable*, *n*)

Break an iterable into lists of a given length:

```
>>> list(chunked([1, 2, 3, 4, 5, 6, 7], 3))
[[1, 2, 3], [4, 5, 6], [7]]
```

If the length of `iterable` is not evenly divisible by `n`, the last returned list will be shorter.

This is useful for splitting up a computation on a large number of keys into batches, to be pickled and sent off to worker processes. One example is operations on rows in MySQL, which does not implement server-side cursors properly and would otherwise load the entire dataset into RAM on the client.

more_itertools.**collate**(**iterables*, *key=lambda a: a*, *reverse=False*)

Return a sorted merge of the items from each of several already-sorted `iterables`.

```
>>> list(collate('ACDZ', 'AZ', 'JKL'))
['A', 'A', 'C', 'D', 'J', 'K', 'L', 'Z', 'Z']
```

Works lazily, keeping only the next value from each iterable in memory. Use `collate()` to, for example, perform a n-way mergesort of items that don't fit in memory.

Parameters:

- **key** – A function that returns a comparison value for an item. Defaults to the identity function.
- **reverse** – If `reverse=True`, yield results in descending order rather than ascending. `iterables` must also yield their elements in descending order.

If the elements of the passed-in iterables are out of order, you might get unexpected results.

more_itertools.**consumer**(*func*)

Decorator that automatically advances a PEP-342-style “reverse iterator” to its first yield point so you don’t have to call `next()` on it manually.

```
>>> @consumer
... def tally():
...     i = 0
...     while True:
...         print 'Thing number %s is %s.' % (i, (yield))
...         i += 1
...
>>> t = tally()
>>> t.send('red')
Thing number 0 is red.
>>> t.send('fish')
Thing number 1 is fish.
```

Without the decorator, you would have to call `t.next()` before `t.send()` could be used.

more_itertools.**first**(*iterable*[, *default*])

Return the first item of an iterable, `default` if there is none.

```
>>> first(xrange(4))
0
>>> first(xrange(0), 'some default')
'some default'
```

If `default` is not provided and there are no items in the iterable, raise `ValueError`.

`first()` is useful when you have a generator of expensive-to-retrieve values and want any arbitrary one. It is marginally shorter than `next(iter(...))` but saves you an entire `try/except` when you want to provide a fallback value.

more_itertools.**ilen**(*iterable*)

Return the number of items in `iterable`.

```
>>> from itertools import ifilter
>>> ilen(ifilter(lambda x: x % 3 == 0, xrange(1000000)))
333334
```

This does, of course, consume the iterable, so handle it with care.

more_itertools.**iterate**(*func*, *start*)

Return `start`, `func(start)`, `func(func(start))`, ...

```
>>> from itertools import islice
>>> list(islice(iterate(lambda x: 2*x, 1), 10))
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

class more_itertools.**peekable**(*iterable*)

Wrapper for an iterator to allow 1-item lookahead

Call `peek()` on the result to get the value that will next pop out of `next()`, without advancing the iterator:

```
>>> p = peekable(xrange(2))
>>> p.peek()
0
>>> p.next()
0
>>> p.peek()
1
>>> p.next()
1
```

Pass `peek()` a default value, and it will be returned in the case where the iterator is exhausted:

```
>>> p = peekable([])
>>> p.peek('hi')
'hi'
```

If no default is provided, `peek()` raises `StopIteration` when there are no items left.

To test whether there are more items in the iterator, examine the peekable's truth value. If it is truthy, there are more items.

```
>>> assert peekable(xrange(1))
>>> assert not peekable([])
```

more_itertools.**with_iter**(*context_manager*)

Wrap an iterable in a `with` statement, so it closes once exhausted.

For example, this will close the file when the iterator is exhausted:

```
upper_lines = (line.upper() for line in with_iter(open('foo')))
```

Any context manager which returns an iterable is a candidate for `with_iter`.

Itertools Recipes

more_itertools.**take**(*n*, *iterable*)

Return first n items of the iterable as a list

```
>>> take(3, range(10))
[0, 1, 2]
>>> take(5, range(3))
[0, 1, 2]
```

Effectively a short replacement for `next` based iterator consumption when you want more than one item, but less than the whole iterator.

more_itertools.**tabulate**(*function*, *start=0*)

Return an iterator mapping the function over linear input.

The start argument will be increased by 1 each time the iterator is called and fed into the function.

```
>>> t = tabulate(lambda x: x**2, -3)
>>> take(3, t)
[9, 4, 1]
```

more_itertools.**consume**(*iterator*, *n=None*)

Advance the iterator n-steps ahead. If n is none, consume entirely.

Efficiently exhausts an iterator without returning values. Defaults to consuming the whole iterator, but an optional second argument may be provided to limit consumption.

```
>>> i = (x for x in range(10))
>>> next(i)
0
>>> consume(i, 3)
>>> next(i)
4
>>> consume(i)
>>> next(i)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

If the iterator has fewer items remaining than the provided limit, the whole iterator will be consumed.

```
>>> i = (x for x in range(3))
>>> consume(i, 5)
>>> next(i)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

more_itertools.**nth**(*iterable*, *n*, *default=None*)

Returns the nth item or a default value

```
>>> l = range(10)
>>> nth(l, 3)
3
>>> nth(l, 20, "zebra")
'zebra'
```

more_itertools.**quantify**(*iterable*, *pred=<type 'bool'>*)

Return the how many times the predicate is true

```
>>> quantify([True, False, True])
2
```

more_itertools.**padnone**(*iterable*)

Returns the sequence of elements and then returns None indefinitely.

```
>>> take(5, padnone(range(3)))
[0, 1, 2, None, None]
```

Useful for emulating the behavior of the built-in map() function.

more_itertools.**ncycles**(*iterable*, *n*)

Returns the sequence elements n times

```
>>> list(ncycles(["a", "b"], 3))
['a', 'b', 'a', 'b', 'a', 'b']
```

more_itertools.**dotproduct**(*vec1*, *vec2*)

Returns the dot product of the two iterables

```
>>> dotproduct([10, 10], [20, 20])
400
```

more_itertools.**flatten**(*listOfLists*)

Return an iterator flattening one level of nesting in a list of lists

```
>>> list(flatten([[0, 1], [2, 3]]))
[0, 1, 2, 3]
```

more_itertools.**repeatfunc**(*func*, *times=None*, **args*)

Repeat calls to func with specified arguments.

```
>>> list(repeatfunc(lambda: 5, 3))
[5, 5, 5]
>>> list(repeatfunc(lambda x: x ** 2, 3, 3))
[9, 9, 9]
```

more_itertools.**pairwise**(*iterable*)

Returns an iterator of paired items, overlapping, from the original

```
>>> take(4, pairwise(count()))
[(0, 1), (1, 2), (2, 3), (3, 4)]
```

more_itertools.**grouper**(*n*, *iterable*, *fillvalue=None*)

Collect data into fixed-length chunks or blocks

```
>>> list(grouper(3, 'ABCDEFG', 'x'))
[('A', 'B', 'C'), ('D', 'E', 'F'), ('G', 'x', 'x')]
```

more_itertools.**roundrobin**(**iterables*)

Yields an item from each iterable, alternating between them

```
>>> list(roundrobin('ABC', 'D', 'EF'))
['A', 'D', 'E', 'B', 'F', 'C']
```

more_itertools.**powerset**(*iterable*)

Yields all possible subsets of the iterable

```
>>> list(powerset([1,2,3]))
[(), (1,), (2,), (3,), (1, 2), (1, 3), (2, 3), (1, 2, 3)]
```

more_itertools.**unique_everseen**(*iterable*, *key=None*)

Yield unique elements, preserving order.

```
>>> list(unique_everseen('AAAABBBCCDAABBB'))
['A', 'B', 'C', 'D']
>>> list(unique_everseen('ABBCcAD', str.lower))
['A', 'B', 'C', 'D']
```

more_itertools.**unique_justseen**(*iterable*, *key=None*)

Yields elements in order, ignoring serial duplicates

```
>>> list(unique_justseen('AAAABBBCCDAABBB'))
['A', 'B', 'C', 'D', 'A', 'B']
>>> list(unique_justseen('ABBCcAD', str.lower))
['A', 'B', 'C', 'A', 'D']
```

more_itertools.**iter_except**(*func*, *exception*, *first=None*)

Yields results from a function repeatedly until an exception is raised.

Converts a call-until-exception interface to an iterator interface. Like __builtin__.iter(func, sentinel) but uses an exception instead of a sentinel to end the loop.

```
>>> l = range(3)
>>> list(iter_except(l.pop, IndexError))
[2, 1, 0]
```

more_itertools.**random_product**(**args*, ***kwds*)

Returns a random pairing of items from each iterable argument

If *repeat* is provided as a kwarg, it's value will be used to indicate how many pairings should be chosen.

```
>>> random_product(['a', 'b', 'c'], [1, 2], repeat=2)
('b', '2', 'c', '2')
```

more_itertools.**random_permutation**(*iterable*, *r=None*)

Returns a random permutation.

If r is provided, the permutation is truncated to length r.

```
>>> random_permutation(range(5))
(3, 4, 0, 1, 2)
```

more_itertools.**random_combination**(*iterable*, *r*)

Returns a random combination of length r, chosen without replacement.

```
>>> random_combination(range(5), 3)
(2, 3, 4)
```

more_itertools.**random_combination_with_replacement**(*iterable*, *r*)

Returns a random combination of length r, chosen with replacement.

```
>>> random_combination_with_replacement(range(3), 5) #
(0, 0, 1, 2, 2)
```