# more-itertools Documentation

*Release 2.6.0*

Erik Rose

Mar 22, 2017

# Contents

I love itertools; it's one of the most beautiful, composable standard libs. Whenever I have an iteration problem, there's almost always an itertools routine that fits it perfectly. Sometimes, however, neither itertools nor the recipes included in its docs do quite what I need.

Here I've collected several routines I've reached for but not found. Since they are deceptively tricky to get right, I've wrapped them up into a library. We've also included implementations of the recipes from the itertools documentation. Enjoy! Any additions are welcome; just file a pull request.

CHAPTER 1

# Contents

## API Reference

### New Routines

more_itertools.**adjacent**(*predicate*, *iterable*, *distance=1*)

Return an iterable over `(bool, item)` tuples where the `item` is drawn from *iterable* and the `bool` indicates whether that item satisfies the *predicate* or is adjacent to an item that does.

For example, to find whether items are adjacent to a `3`:

```
>>> list(adjacent(lambda x: x == 3, range(6)))
[(False, 0), (False, 1), (True, 2), (True, 3), (True, 4), (False, 5)]
```

Set *distance* to change what counts as adjacent. For example, To find whether items are two places away from a `3`:

```
>>> list(adjacent(lambda x: x == 3, range(6), distance=2))
[(False, 0), (True, 1), (True, 2), (True, 3), (True, 4), (True, 5)]
```

This is useful for contextualizing the results of a search function. For example, a code comparison tool might want to identify lines that have changed, but also surrounding lines to give the viewer of the diff context.

The predicate function will only be called once for each item in the iterable.

See also `groupby_transform()`, which can be used with this function to group ranges of items with the same `bool` value.

more_itertools.**always_iterable**(*obj*)

Given an object, always return an iterable.

If the object is not already iterable, return a tuple containing containing the object:

```
>>> always_iterable(1)
(1,)
```

If the object is `None`, return an empty iterable:

```
>>> always_iterable(None)
()
```

Otherwise, return the object itself:

```
>>> always_iterable([1, 2, 3])
[1, 2, 3]
```

Strings (binary or unicode) are not considered to be iterable:

```
>>> always_iterable('foo')
('foo',)
```

This function is useful in applications where a passed parameter may be either a single item or a collection of items:

```
>>> def item_sum(param):
...     total = 0
...     for item in always_iterable(param):
...         total += item
...
...     return total
>>> item_sum(10)
10
>>> item_sum([10, 20])
30
```

`more_itertools.`**`bucket`**`(`*iterable*, *key*`)`

Wrap an iterable and return an object that buckets the iterable into child iterables based on a `key` function.

```
>>> iterable = ['a1', 'b1', 'c1', 'a2', 'b2', 'c2', 'b3']
>>> s = bucket(iterable, key=lambda s: s[0])
>>> a_iterable = s['a']
>>> next(a_iterable)
'a1'
>>> next(a_iterable)
'a2'
>>> list(s['b'])
['b1', 'b2', 'b3']
```

The original iterable will be advanced and its items will be cached until they are used by the child iterables. This may require significant storage. Be aware that attempting to select a bucket that no items correspond to will exhaust the iterable and cache all values.

`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.**collapse**(*iterable*, *base_type=None*, *levels=None*)

Flatten an iterable containing some iterables (themselves containing some iterables, etc.) into non-iterable types, strings, elements matching `isinstance(element, base_type)`, and elements that are `levels` levels down.

```
>>> list(collapse([[1], 2, [[3], 4], [[[5]]], 'abc']))
[1, 2, 3, 4, 5, 'abc']
>>> list(collapse([[1], 2, [[3], 4], [[[5]]]], levels=2))
[1, 2, 3, 4, [5]]
>>> list(collapse((1, [2], (3, [4, (5,)])), list))
[1, [2], 3, [4, (5,)]]
```

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.

If neither of the keyword arguments are specified, this function delegates to `heapq.merge()`.

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 `next(t)` before `t.send()` could be used.

more_itertools.**context**(*obj*)

Wrap *obj*, an object that supports the context manager protocol, in a `with` statement and then yield the resultant object.

The object's `__enter__()` method runs before this function yields, and its `__exit__()` method runs after control returns to this function.

This can be used to operate on objects that can close automatically when using a `with` statement, like IO objects:

```
>>> from io import StringIO
>>> from more_itertools import consume
>>> it = [u'1', u'2', u'3']
>>> file_obj = StringIO()
>>> consume(print(x, file=f) for f in context(file_obj) for x in it)
>>> file_obj.closed
True
```

Be sure to iterate over the returned context manager in the outermost loop of a nested loop structure so it only enters and exits once:

```
>>> # Right
>>> file_obj = StringIO()
>>> consume(print(x, file=f) for f in context(file_obj) for x in it)

>>> # Wrong
>>> file_obj = StringIO()
>>> consume(print(x, file=f) for x in it for f in context(file_obj))
Traceback (most recent call last):
...
ValueError: I/O operation on closed file.
```

`more_itertools.`**`distinct_permutations`**(*iterable*)

Yield successive distinct permutations of the elements in the iterable.

```
>>> sorted(distinct_permutations([1, 0, 1]))
[(0, 1, 1), (1, 0, 1), (1, 1, 0)]
```

Equivalent to `set(permutations(iterable))`, except duplicates are not generated and thrown away. For larger input sequences this is much more efficient.

Duplicate permutations arise when there are duplicated elements in the input iterable. The number of items returned is `n! / (x_1! * x_2! * ... * x_n!)`, where `n` is the total number of items input, and each `x_i` is the count of a distinct item in the input sequence.

`more_itertools.`**`distribute`**(*n*, *iterable*)

Distribute the items from *iterable* among *n* smaller iterables.

```
>>> group_1, group_2 = distribute(2, [1, 2, 3, 4, 5, 6])
>>> list(group_1)
[1, 3, 5]
>>> list(group_2)
[2, 4, 6]
```

If the length of the iterable is not evenly divisible by n, then the length of the smaller iterables will not be identical:

```
>>> children = distribute(3, [1, 2, 3, 4, 5, 6, 7])
>>> [list(c) for c in children]
[[1, 4, 7], [2, 5], [3, 6]]
```

If the length of the iterable is smaller than n, then the last returned iterables will be empty:

```
>>> children = distribute(5, [1, 2, 3])
>>> [list(c) for c in children]
[[1], [2], [3], [], []]
```

This function uses `itertools.tee` and may require significant storage. If you need the order items in the smaller iterables to match the original iterable, see `divide()`.

`more_itertools.`**`divide`**(*n*, *iterable*)

Divide the elements from *iterable* into *n* parts, maintaining order.

```
>>> group_1, group_2 = divide(2, [1, 2, 3, 4, 5, 6])
>>> list(group_1)
[1, 2, 3]
>>> list(group_2)
[4, 5, 6]
```

If the length of the iterable is not evenly divisible by n, then the length of the smaller iterables will not be identical:

```
>>> children = divide(3, [1, 2, 3, 4, 5, 6, 7])
>>> [list(c) for c in children]
[[1, 2, 3], [4, 5], [6, 7]]
```

If the length of the iterable is smaller than n, then the last returned iterables will be empty:

```
>>> children = divide(5, [1, 2, 3])
>>> [list(c) for c in children]
[[1], [2], [3], [], []]
```

This function will exhaust the iterable before returning and may require significant storage. If order is not important, see `distribute()`, which does not first pull the iterable into memory.

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

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

```
>>> first([0, 1, 2, 3])
0
>>> first([], '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(...), default)`.

`more_itertools.`**`groupby_transform`**(*iterable*, *keyfunc=None*, *valuefunc=None*)

Make an iterator that returns consecutive keys and groups from the *iterable*. *keyfunc* is a function used to compute a grouping key for each item. *valuefunc* is a function for transforming the items after grouping.

*keyfunc* and *valuefunc* default to identity functions if they are not specified. When *valuefunc* is not specified, `groupby_transform` is the same as `itertools.groupby()`.

For example, to group a list of numbers by rounding down to the nearest 10, and then transform them into strings:

```
>>> iterable = [0, 1, 12, 13, 23, 24]
>>> keyfunc = lambda x: 10 * (x // 10)
>>> valuefunc = lambda x: str(x)
>>> grouper = groupby_transform(iterable, keyfunc, valuefunc)
>>> [(k, list(g)) for k, g in grouper]
[(0, ['0', '1']), (10, ['12', '13']), (20, ['23', '24'])]
```

`groupby_transform` is useful when grouping elements of an iterable using a separate iterable as the key. To do this, `zip()` the iterables and pass a *keyfunc* that extracts the first element and a *valuefunc* that extracts the second element:

```
>>> from operator import itemgetter
>>> keys = [0, 0, 1, 1, 1, 2, 2, 2, 3]
>>> values = 'abcdefghi'
>>> iterable = zip(keys, values)
>>> grouper = groupby_transform(iterable, itemgetter(0), itemgetter(1))
>>> [(k, ''.join(g)) for k, g in grouper]
[(0, 'ab'), (1, 'cde'), (2, 'fgh'), (3, 'i')]
```

`more_itertools.`**`ilen`**`(`*iterable*`)`
Return the number of items in `iterable`.

```
>>> ilen(x for x in range(1000000) if x % 3 == 0)
333334
```

This consumes the iterable, so handle with care.

`more_itertools.`**`interleave`**`(`**iterables*`)`
Return a new iterable yielding from each iterable in turn, until the shortest is exhausted. Note that this is the same as `chain(*zip(*iterables))`.

```
>>> list(interleave([1, 2, 3], [4, 5], [6, 7, 8]))
[1, 4, 6, 2, 5, 7]
```

`more_itertools.`**`interleave_longest`**`(`**iterables*`)`
Return a new iterable yielding from each iterable in turn, skipping any that are exhausted. Note that this is not the same as `chain(*zip_longest(*iterables))`.

```
>>> list(interleave_longest([1, 2, 3], [4, 5], [6, 7, 8]))
[1, 4, 6, 2, 5, 7, 3, 8]
```

`more_itertools.`**`intersperse`**`(`*e*, *iterable*`)`
Intersperse element `e` between the elements of *iterable*.

```
>>> list(intersperse('x', 'ABCD'))
['A', 'x', 'B', 'x', 'C', 'x', 'D']
>>> list(intersperse(None, [1, 2, 3]))
[1, None, 2, None, 3]
```

`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]
```

`more_itertools.`**`one`**`(`*iterable*`)`
Return the only element from the iterable.

Raise ValueError if the iterable is empty or longer than 1 element. For example, assert that a DB query returns a single, unique result.

```
>>> one(['val'])
'val'
```

```
>>> one(['val', 'other'])
Traceback (most recent call last):
...
ValueError: too many values to unpack (expected 1)
```

```
>>> one([])
Traceback (most recent call last):
...
ValueError: not enough values to unpack (expected 1, got 0)
```

`one()` attempts to advance the iterable twice in order to ensure there aren't further items. Because this discards any second item, `one()` is not suitable in situations where you want to catch its exception and then try an alternative treatment of the iterable. It should be used only when a iterable longer than 1 item is, in fact, an error.

`more_itertools.`**`padded`**`(`*iterable*, *fillvalue=None*, *n=None*, *next_multiple=False*`)`

Yield the elements from *iterable*, followed by *fillvalue*, such that at least *n* items are emitted.

```
>>> list(padded([1, 2, 3], '?', 5))
[1, 2, 3, '?', '?']
```

If *next_multiple* is `True`, *fillvalue* will be emitted until the number of items emitted is a multiple of *n*:

```
>>> list(padded([1, 2, 3, 4], n=3, next_multiple=True))
[1, 2, 3, 4, None, None]
```

If *n* is `None`, *fillvalue* will be emitted indefinitely.

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

Wrap an iterator to allow lookahead and prepending elements.

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

```
>>> p = peekable(['a', 'b'])
>>> p.peek()
'a'
>>> next(p)
'a'
```

Pass `peek()` a default value to return that instead of raising `StopIteration` when the iterator is exhausted.

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

peekables also offer a `prepend()` method which will insert items before the remaining part of the underlying source iterator.

```
>>> p = peekable([1, 2, 3])
>>> p.prepend(10, 11, 12)
>>> next(p)
10
>>> p.peek()
11
>>> list(p)
[11, 12, 1, 2, 3]
```

Prepended items are treated by other peekable methods exactly as if they had come from the source iterator.

You may index the peekable to look ahead by more than one item. The values up to the index you specified will be cached. Index 0 is the item that will be returned by `next()`, index 1 is the item after that, and so on:

```
>>> p = peekable(['a', 'b', 'c', 'd'])
>>> p[0]
'a'
>>> p[1]
'b'
>>> next(p)
'a'
>>> p.prepend('x')
>>> p[1]
'b'
>>> next(p)
'x'
>>> next(p)
'b'
```

Negative indexes are supported, but be aware that they will cache the remaining items in the source iterator, which may require significant storage.

To test whether there are more items in the iterator, examine the peekable's truth value. If it is truthy, there are more items (which may have been prepended or obtained from the source iterator).

```
>>> assert peekable([1])
>>> p = peekable([])
>>> assert not p
>>> p.prepend(1)
>>> assert p
```

`more_itertools.`**`side_effect`**(*func*, *iterable*, *chunk_size=None*)

Invoke *func* on each item in *iterable* (or on each *chunk_size* group of items) before yielding the item.

*func* must be a function that takes a single argument. Its return value will be discarded.

*side_effect* can be used for logging, updating progress bars, or anything that is not functionally "pure."

Emitting a status message:

```
>>> from more_itertools import consume
>>> func = lambda item: print('Received {}'.format(item))
>>> consume(side_effect(func, range(2)))
Received 0
Received 1
```

Operating on chunks of items:

```
>>> pair_sums = []
>>> func = lambda chunk: pair_sums.append(sum(chunk))
>>> list(side_effect(func, [0, 1, 2, 3, 4, 5], 2))
[0, 1, 2, 3, 4, 5]
>>> list(pair_sums)
[1, 5, 9]
```

Writing to a file-like object:

```
>>> from io import StringIO
>>> from more_itertools import consume
>>> with StringIO() as f:
...     func = lambda x: print(x, end=u',', file=f)
...     it = [u'a', u'b', u'c']
...     consume(side_effect(func, it))
...     print(f.getvalue())
a,b,c,
```

`more_itertools.sliced`(*seq*, *n*)

Yield slices of length *n* from the sequence *seq*.

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

If the length of the sequence is not divisible by the requested slice length, the last slice will be shorter.

```
>>> list(sliced((1, 2, 3, 4, 5, 6, 7, 8), 3))
[(1, 2, 3), (4, 5, 6), (7, 8)]
```

This function will only work for sliceable objects. For non-sliceable iterable, see `chunked()`.

`more_itertools.sort_together`(*iterables*, *key_list=(0, )*, *reverse=False*)

Return the input iterables sorted together, with *key_list* as the priority for sorting. All iterables are trimmed to the length of the shortest one.

This can be used like the sorting function in a spreadsheet. If each iterable represents a column of data, the key list determines which columns are used for sorting.

By default, all iterables are sorted using the `0`-th iterable:

```
>>> iterables = [(4, 3, 2, 1), ('a', 'b', 'c', 'd')]
>>> sort_together(iterables)
[(1, 2, 3, 4), ('d', 'c', 'b', 'a')]
```

Set a different key list to sort according to another iterable. Specifying mutliple keys dictates how ties are broken:

```
>>> iterables = [(3, 1, 2), (0, 1, 0), ('c', 'b', 'a')]
>>> sort_together(iterables, key_list=(1, 2))
[(2, 3, 1), (0, 0, 1), ('a', 'c', 'b')]
```

Set *reverse* to `True` to sort in descending order.

```
>>> sort_together([(1, 2, 3), ('c', 'b', 'a')], reverse=True)
[(3, 2, 1), ('a', 'b', 'c')]
```

`more_itertools.split_after`(*iterable*, *pred*)

Yield lists of items from *iterable*, where each list ends with an item where callable *pred* returns `True`:

```
>>> list(split_after('one1two2', lambda s: s.isdigit()))
[['o', 'n', 'e', '1'], ['t', 'w', 'o', '2']]
```

```
>>> list(split_after(range(10), lambda n: n % 3 == 0))
[[0], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`more_itertools.split_before`(*iterable*, *pred*)

Yield lists of items from *iterable*, where each list starts with an item where callable *pred* returns `True`:

```
>>> list(split_before('OneTwo', lambda s: s.isupper()))
[['O', 'n', 'e'], ['T', 'w', 'o']]
```

```
>>> list(split_before(range(10), lambda n: n % 3 == 0))
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9]]
```

`more_itertools.`**`spy`**(*iterable*, *n=1*)

Return a 2-tuple with a list containing the first *n* elements of *iterable*, and an iterator with the same items as *iterable*. This allows you to "look ahead" at the items in the iterable without advancing it.

There is one item in the list by default:

```
>>> iterable = 'abcdefg'
>>> head, iterable = spy(iterable)
>>> head
['a']
>>> list(iterable)
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

You may use unpacking to retrieve items instead of lists:

```
>>> (head,), iterable = spy('abcdefg')
>>> head
'a'
>>> (first, second), iterable = spy('abcdefg', 2)
>>> first
'a'
>>> second
'b'
```

The number of items requested can be larger than the number of items in the iterable:

```
>>> iterable = [1, 2, 3, 4, 5]
>>> head, iterable = spy(iterable, 10)
>>> head
[1, 2, 3, 4, 5]
>>> list(iterable)
[1, 2, 3, 4, 5]
```

`more_itertools.`**`stagger`**(*iterable*, *offsets=(-1, 0, 1)*, *longest=False*, *fillvalue=None*)

Yield tuples whose elements are offset from *iterable*. The amount by which the `i`-th item in each tuple is offset is given by the `i`-th item in *offsets*.

```
>>> list(stagger([0, 1, 2, 3]))
[(None, 0, 1), (0, 1, 2), (1, 2, 3)]
>>> list(stagger(range(8), offsets=(0, 2, 4)))
[(0, 2, 4), (1, 3, 5), (2, 4, 6), (3, 5, 7)]
```

By default, the sequence will end when the final element of a tuple is the last item in the iterable. To continue until the first element of a tuple is the last item in the iterable, set *longest* to `True`:

```
>>> list(stagger([0, 1, 2, 3], longest=True))
[(None, 0, 1), (0, 1, 2), (1, 2, 3), (2, 3, None), (3, None, None)]
```

By default, `None` will be used to replace offsets beyond the end of the sequence. Specify *fillvalue* to use some other value.

`more_itertools.`**`unique_to_each`**(**iterables*)

Return the elements from each of the input iterables that aren't in the other input iterables.

For example, suppose you have a set of packages, each with a set of dependencies:

```
{'pkg_1': {'A', 'B'}, 'pkg_2': {'B', 'C'}, 'pkg_3': {'B', 'D'}}
```

If you remove one package, which dependencies can also be removed?

If `pkg_1` is removed, then `A` is no longer necessary - it is not associated with `pkg_2` or `pkg_3`. Similarly, `C` is only needed for `pkg_2`, and `D` is only needed for `pkg_3`:

```
>>> unique_to_each({'A', 'B'}, {'B', 'C'}, {'B', 'D'})
[['A'], ['C'], ['D']]
```

If there are duplicates in one input iterable that aren't in the others they will be duplicated in the output. Input order is preserved:

```
>>> unique_to_each("mississippi", "missouri")
[['p', 'p'], ['o', 'u', 'r']]
```

It is assumed that the elements of each iterable are hashable.

`more_itertools.`**`windowed`**(*seq*, *n*, *fillvalue=None*, *step=1*)
Return a sliding window of width *n* over the given iterable.

```
>>> all_windows = windowed([1, 2, 3, 4, 5], 3)
>>> list(all_windows)
[(1, 2, 3), (2, 3, 4), (3, 4, 5)]
```

When the window is larger than the iterable, `fillvalue` is used in place of missing values:

```
>>> list(windowed([1, 2, 3], 4))
[(1, 2, 3, None)]
```

Each window will advance in increments of *step*:

```
>>> list(windowed([1, 2, 3, 4, 5, 6], 3, fillvalue='!', step=2))
[(1, 2, 3), (3, 4, 5), (5, 6, '!')]
```

`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`.

`more_itertools.`**`zip_offset`**(**iterables*, *offsets*, *longest=False*, *fillvalue=None*)
`zip` the input *iterables* together, but offset the `i`-th iterable by the `i`-th item in *offsets*.

```
>>> list(zip_offset('0123', 'abcdef', offsets=(0, 1)))
[('0', 'b'), ('1', 'c'), ('2', 'd'), ('3', 'e')]
```

This can be used as a lightweight alternative to SciPy or pandas to analyze data sets in which somes series have a lead or lag relationship.

By default, the sequence will end when the shortest iterable is exhausted. To continue until the longest iterable is exhausted, set *longest* to `True`.

```
>>> list(zip_offset('0123', 'abcdef', offsets=(0, 1), longest=True))
[('0', 'b'), ('1', 'c'), ('2', 'd'), ('3', 'e'), (None, 'f')]
```

By default, `None` will be used to replace offsets beyond the end of the sequence. Specify *fillvalue* to use some other value.

## Itertools Recipes

`more_itertools.`**`accumulate`**(*iterable*, *func=<built-in function add>*)

Return an iterator whose items are the accumulated results of a function (specified by the optional *func* argument) that takes two arguments. By default, returns accumulated sums with `operator.add()`.

```
>>> list(accumulate([1, 2, 3, 4, 5]))  # Running sum
[1, 3, 6, 10, 15]
>>> list(accumulate([1, 2, 3], func=operator.mul))  # Running product
[1, 2, 6]
>>> list(accumulate([0, 1, -1, 2, 3, 2], func=max))  # Running maximum
[0, 1, 1, 2, 3, 3]
```

This function is available in the `itertools` module for Python 3.2 and greater.

`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.`**`tail`**(*n*, *iterable*)

Return an iterator over the last n items”

```
>>> t = tail(3, 'ABCDEFG')
>>> list(t)
['E', 'F', 'G']
```

`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.`**`all_equal`**`(`*iterable*`)`
Returns True if all the elements are equal to each other.

```
>>> all_equal('aaaa')
True
>>> all_equal('aaab')
False
```

`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.`**`partition`**(*pred*, *iterable*)

Returns a 2-tuple of iterables derived from the input iterable. The first yields the items that have `pred(item) == False`. The first yields the items that have `pred(item) == False`.

```
>>> is_odd = lambda x: x % 2 != 0
>>> iterable = range(10)
>>> even_items, odd_items = partition(is_odd, iterable)
>>> list(even_items), list(odd_items)
([0, 2, 4, 6, 8], [1, 3, 5, 7, 9])
```

`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']
```

Sequences with a mix of hashable and unhashable items can be used. The function will be slower (i.e., O(N^2)) for unhashable items.

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

`more_itertools.`**`first_true`**(*iterable*, *default=False*, *pred=None*)

Returns the first true value in the iterable.

If no true value is found, returns *default*

If *pred* is not None, returns the first item for which `pred(item) == True`.

```
>>> first_true(range(10))
1
>>> first_true(range(10), pred=lambda x: x > 5)
6
>>> first_true(range(10), default='missing', pred=lambda x: x > 9)
'missing'
```

`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)
```

## License

more-itertools is under the MIT License. See the LICENSE file.

### Conditions for Contributors

By contributing to this software project, you are agreeing to the following terms and conditions for your contributions: First, you agree your contributions are submitted under the MIT license. Second, you represent you are authorized to make the contributions and grant the license. If your employer has rights to intellectual property that includes your contributions, you represent that you have received permission to make contributions and grant the required license on behalf of that employer.

## Testing

more-itertools uses nose for its tests. First, install nose:

```
pip install nose
```

Then, run the tests like this:

```
nosetests --with-doctest
```

### Multiple Python Versions

To run the tests on all the versions of Python more-itertools supports, install tox:

```
pip install tox
```

Then, run the tests:

```
tox
```

## Version History

**2.6.0**

- **New itertools:**
  - `adjacent` and `groupby_transform` (Thanks to diazona)
  - `always_iterable` (Thanks to jaraco)
  - `context` (Thanks to yardsale8)
  - `divide` (Thanks to mozbhearsum)

- **Improvements to existing itertools:**
  - `ilen` is now slightly faster. (Thanks to wbolster)
  - `peekable` can now prepend items to an iterable. (Thanks to diazona)

**2.5.0**

- **New itertools:**
  - `distribute` (Thanks to mozbhearsum and coady)
  - `sort_together` (Thanks to clintval)
  - `stagger` and `zip_offset` (Thanks to joshbode)
  - `padded`
- **Improvements to existing itertools:**
  - `peekable` now handles negative indexes and slices with negative components properly.
  - `intersperse` is now slightly faster. (Thanks to pylang)
  - `windowed` now accepts a `step` keyword argument. (Thanks to pylang)
- Python 3.6 is now supported.

**2.4.1**

- Move docs 100% to readthedocs.io.

**2.4**

- **New itertools:**
  - `accumulate`, `all_equal`, `first_true`, `partition`, and `tail` from the itertools documentation.
  - `bucket` (Thanks to Rosuav and cvrebert)
  - `collapse` (Thanks to abarnet)
  - `interleave` and `interleave_longest` (Thanks to abarnet)
  - `side_effect` (Thanks to nvie)
  - `sliced` (Thanks to j4mie and coady)
  - `split_before` and `split_after` (Thanks to astronouth7303)
  - `spy` (Thanks to themiurgo and mathieulongtin)
- **Improvements to existing itertools:**
  - `chunked` is now simpler and more friendly to garbage collection. (Contributed by coady, with thanks to piskvorky)
  - `collate` now delegates to `heapq.merge` when possible. (Thanks to kmike and julianpistorius)
  - `peekable`-wrapped iterables are now indexable and sliceable. Iterating through `peekable`-wrapped iterables is also faster.
  - `one` and `unique_to_each` have been simplified. (Thanks to coady)

**2.3**

- Added `one` from `jaraco.util.itertools`. (Thanks, jaraco!)

- Added `distinct_permutations` and `unique_to_each`. (Contributed by bbayles)
- Added `windowed`. (Contributed by bbayles, with thanks to buchanae, jaraco, and abarnert)
- Simplified the implementation of `chunked`. (Thanks, nvie!)
- Python 3.5 is now supported. Python 2.6 is no longer supported.
- Python 3 is now supported directly; there is no 2to3 step.

**2.2**

- Added `iterate` and `with_iter`. (Thanks, abarnert!)

**2.1**

- Added (tested!) implementations of the recipes from the itertools documentation. (Thanks, Chris Lonnen!)
- Added `ilen`. (Thanks for the inspiration, Matt Basta!)

**2.0**

- `chunked` now returns lists rather than tuples. After all, they're homogeneous. This slightly backward-incompatible change is the reason for the major version bump.
- Added `@consumer`.
- Improved test machinery.

**1.1**

- Added `first` function.
- Added Python 3 support.
- Added a default arg to `peekable.peek()`.
- Noted how to easily test whether a peekable iterator is exhausted.
- Rewrote documentation.

**1.0**

- Initial release, with `collate`, `peekable`, and `chunked`. Could really use better docs.

# Python Module Index

# Index