:mod:`itertools` --- Functions creating iterators for efficient looping

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 4)

Unknown directive type "module".

```
.. module:: itertools
   :synopsis: Functions creating iterators for efficient looping.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 7)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Raymond Hettinger <python@rcn.com>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 8)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Raymond Hettinger <python@rcn.com>
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 10)

Unknown directive type "testsetup".

```
.. testsetup::

   from itertools import *
```

This module implements a number of :term:`iterator` building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 16); *backlink*

Unknown interpreted text role "term".

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.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. The same effect can be achieved in Python by combining :func:`map` and :func:`count` to form `map(f, count())`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 25); *backlink*

Unknown interpreted text role "func".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 25); *backlink*

Unknown interpreted text role "func".

These tools and their built-in counterparts also work well with the high-speed functions in the :mod:`operator` module. For example, the multiplication operator can be mapped across two vectors to form an efficient dot-product: `sum(map(operator.mul, vector1, vector2))`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 29); *backlink*

Unknown interpreted text role "mod".

Infinite iterators:

Iterator	Arguments	Results	Example
:func:`count` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 41); *backlink* Unknown interpreted text role "func".	start, [step]	start, start+step, start+2*step, ...	`count(10) --> 10 11 12 13 14 ...`
:func:`cycle` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 42); *backlink* Unknown interpreted text role "func".	p	p0, p1, ... plast, p0, p1, ...	`cycle('ABCD') --> A B C D A B C D ...`
:func:`repeat` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 43); *backlink* Unknown interpreted text role "func".	elem [,n]	elem, elem, elem, ... endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
:func:`accumulate` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 51); *backlink* Unknown interpreted text role "func".	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) --> 1 3 6 10 15`

Iterator	Arguments	Results	Example
:func:`chain` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 52); *backlink*** Unknown interpreted text role "func".	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') --> A B C D E F`
:func:`chain.from_iterable` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 53); *backlink*** Unknown interpreted text role "func".	iterable	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) --> A B C D E F`
:func:`compress` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 54); *backlink*** Unknown interpreted text role "func".	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F`
:func:`dropwhile` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 55); *backlink*** Unknown interpreted text role "func".	pred, seq	seq[n], seq[n+1], starting when pred fails	`dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1`

Iterator	Arguments	Results	Example
:func:`filterfalse` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 56); *backlink*** Unknown interpreted text role "func".	pred, seq	elements of seq where pred(elem) is false	`filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8`
:func:`groupby` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 57); *backlink*** Unknown interpreted text role "func".	iterable[, key]	sub-iterators grouped by value of key(v)	
:func:`islice` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 58); *backlink*** Unknown interpreted text role "func".	seq, [start,] stop [, step]	elements from seq[start:stop:step]	`islice('ABCDEFG', 2, None) --> C D E F G`
:func:`pairwise` **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 59); *backlink*** Unknown interpreted text role "func".	iterable	(p[0], p[1]), (p[1], p[2])	`pairwise('ABCDEFG') --> AB BC CD DE EF FG`

Iterator	Arguments	Results	Example
:func:`starmap` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 60); *backlink* Unknown interpreted text role "func".	func, seq	func(*seq[0]), func(*seq[1]), ...	`starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000`
:func:`takewhile` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 61); *backlink* Unknown interpreted text role "func".	pred, seq	seq[0], seq[1], until pred fails	`takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4`
:func:`tee` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 62); *backlink* Unknown interpreted text role "func".	it, n	it1, it2, ... itn splits one iterator into n	
:func:`zip_longest` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 63); *backlink* Unknown interpreted text role "func".	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	`zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-`

Combinatoric iterators:

Iterator	Arguments	Results

Iterator	Arguments	Results
:func:`product` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 71); *backlink* Unknown interpreted text role "func".	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
:func:`permutations` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 72); *backlink* Unknown interpreted text role "func".	p[, r]	r-length tuples, all possible orderings, no repeated elements
:func:`combinations` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 73); *backlink* Unknown interpreted text role "func".	p, r	r-length tuples, in sorted order, no repeated elements
:func:`combinations_with_replacement` System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 74); *backlink* Unknown interpreted text role "func".	p, r	r-length tuples, in sorted order, with repeated elements

Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD',Â 2)`	`AA AB AC AD BB BC BD CC CD DD`

Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be

accessed by functions or loops that truncate the stream.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 95)

Unknown directive type "function".

```
.. function:: accumulate(iterable[, func, *, initial=None])

    Make an iterator that returns accumulated sums, or accumulated
    results of other binary functions (specified via the optional
    *func* argument).

    If *func* is supplied, it should be a function
    of two arguments. Elements of the input *iterable* may be any type
    that can be accepted as arguments to *func*. (For example, with
    the default operation of addition, elements may be any addable
    type including :class:`~decimal.Decimal` or
    :class:`~fractions.Fraction`.)

    Usually, the number of elements output matches the input iterable.
    However, if the keyword argument *initial* is provided, the
    accumulation leads off with the *initial* value so that the output
    has one more element than the input iterable.

    Roughly equivalent to::

        def accumulate(iterable, func=operator.add, *, initial=None):
            'Return running totals'
            # accumulate([1,2,3,4,5]) --> 1 3 6 10 15
            # accumulate([1,2,3,4,5], initial=100) --> 100 101 103 106 110 115
            # accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120
            it = iter(iterable)
            total = initial
            if initial is None:
                try:
                    total = next(it)
                except StopIteration:
                    return
            yield total
            for element in it:
                total = func(total, element)
                yield total

    There are a number of uses for the *func* argument.  It can be set to
    :func:`min` for a running minimum, :func:`max` for a running maximum, or
    :func:`operator.mul` for a running product.  Amortization tables can be
    built by accumulating interest and applying payments.  First-order
    `recurrence relations <https://en.wikipedia.org/wiki/Recurrence_relation>`_
    can be modeled by supplying the initial value in the iterable and using only
    the accumulated total in *func* argument::

      >>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
      >>> list(accumulate(data, operator.mul))     # running product
      [3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]
      >>> list(accumulate(data, max))              # running maximum
      [3, 4, 6, 6, 6, 9, 9, 9, 9, 9]

      # Amortize a 5% loan of 1000 with 4 annual payments of 90
      >>> cashflows = [1000, -90, -90, -90, -90]
      >>> list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))
      [1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]

      # Chaotic recurrence relation https://en.wikipedia.org/wiki/Logistic_map
      >>> logistic_map = lambda x, _:  r * x * (1 - x)
      >>> r = 3.8
      >>> x0 = 0.4
      >>> inputs = repeat(x0, 36)     # only the initial value is used
      >>> [format(x, '.2f') for x in accumulate(inputs, logistic_map)]
      ['0.40', '0.91', '0.30', '0.81', '0.60', '0.92', '0.29', '0.79', '0.63',
       '0.88', '0.39', '0.90', '0.33', '0.84', '0.52', '0.95', '0.18', '0.57',
       '0.93', '0.25', '0.71', '0.79', '0.63', '0.88', '0.39', '0.91', '0.32',
       '0.83', '0.54', '0.95', '0.20', '0.60', '0.91', '0.30', '0.80', '0.60']

    See :func:`functools.reduce` for a similar function that returns only the
    final accumulated value.

    .. versionadded:: 3.2

    .. versionchanged:: 3.3
       Added the optional *func* parameter.

    .. versionchanged:: 3.8
       Added the optional *initial* parameter.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 173)

Unknown directive type "function".

```
.. function:: chain(*iterables)
```

```
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.
Roughly equivalent to::

   def chain(*iterables):
       # chain('ABC', 'DEF') --> A B C D E F
       for it in iterables:
           for element in it:
               yield element
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 187)

Unknown directive type "classmethod".

```
.. classmethod:: chain.from_iterable(iterable)

   Alternate constructor for :func:`chain`.  Gets chained inputs from a
   single iterable argument that is evaluated lazily.  Roughly equivalent to::

      def from_iterable(iterables):
          # chain.from_iterable(['ABC', 'DEF']) --> A B C D E F
          for it in iterables:
              for element in it:
                  yield element
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 199)

Unknown directive type "function".

```
.. function:: combinations(iterable, r)

   Return *r* length subsequences of elements from the input *iterable*.

   The combination tuples are emitted in lexicographic ordering according to
   the order of the input *iterable*. 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.

   Roughly equivalent to::

        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 = list(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)

   The code for :func:`combinations` can be also expressed as a subsequence
   of :func:`permutations` after filtering entries where the elements are not
   in sorted order (according to their position in the input pool)::

        def combinations(iterable, r):
            pool = tuple(iterable)
            n = len(pool)
            for indices in permutations(range(n), r):
                if sorted(indices) == list(indices):
                    yield tuple(pool[i] for i in indices)

   The number of items returned is ``n! / r! / (n-r)!`` when ``0 <= r <= n``
   or zero when ``r > n``.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 247)

Unknown directive type "function".

```
.. function:: combinations_with_replacement(iterable, r)

   Return *r* length subsequences of elements from the input *iterable*
   allowing individual elements to be repeated more than once.

   The combination tuples are emitted in lexicographic ordering according to
   the order of the input *iterable*. 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, the generated combinations
   will also be unique.

   Roughly equivalent to::

        def combinations_with_replacement(iterable, r):
            # combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC
            pool = tuple(iterable)
            n = len(pool)
            if not n and r:
                return
            indices = [0] * r
            yield tuple(pool[i] for i in indices)
            while True:
                for i in reversed(range(r)):
                    if indices[i] != n - 1:
                        break
                else:
                    return
                indices[i:] = [indices[i] + 1] * (r - i)
                yield tuple(pool[i] for i in indices)

   The code for :func:`combinations_with_replacement` can be also expressed as
   a subsequence of :func:`product` after filtering entries where the elements
   are not in sorted order (according to their position in the input pool)::

        def combinations_with_replacement(iterable, r):
            pool = tuple(iterable)
            n = len(pool)
            for indices in product(range(n), repeat=r):
                if sorted(indices) == list(indices):
                    yield tuple(pool[i] for i in indices)

   The number of items returned is ``(n+r-1)! / r! / (n-1)!`` when ``n > 0``.

   .. versionadded:: 3.1
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 295)

Unknown directive type "function".

```
.. function:: compress(data, selectors)

   Make an iterator that filters elements from *data* returning only those that
   have a corresponding element in *selectors* that evaluates to ``True``.
   Stops when either the *data* or *selectors* iterables has been exhausted.
   Roughly equivalent to::

       def compress(data, selectors):
           # compress('ABCDEF', [1,0,1,0,1,1]) --> A C E F
           return (d for d, s in zip(data, selectors) if s)

   .. versionadded:: 3.1
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 309)

Unknown directive type "function".

```
.. function:: count(start=0, step=1)

   Make an iterator that returns evenly spaced values starting with number *start*. Often
   used as an argument to :func:`map` to generate consecutive data points.
   Also, used with :func:`zip` to add sequence numbers.  Roughly equivalent to::

      def count(start=0, step=1):
          # count(10) --> 10 11 12 13 14 ...
          # count(2.5, 0.5) -> 2.5 3.0 3.5 ...
          n = start
          while True:
              yield n
              n += step

   When counting with floating point numbers, better accuracy can sometimes be
   achieved by substituting multiplicative code such as: ``(start + step * i
   for i in count())``.
```

```
   .. versionchanged:: 3.1
      Added *step* argument and allowed non-integer arguments.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 330)

Unknown directive type "function".

```
.. function:: cycle(iterable)

   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.  Repeats
   indefinitely.  Roughly equivalent to::

      def cycle(iterable):
          # cycle('ABCD') --> A B C D A B C D A B C D ...
          saved = []
          for element in iterable:
              yield element
              saved.append(element)
          while saved:
              for element in saved:
                    yield element

   Note, this member of the toolkit may require significant auxiliary storage
   (depending on the length of the iterable).
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 350)

Unknown directive type "function".

```
.. function:: dropwhile(predicate, iterable)

   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.  Roughly equivalent to::

      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
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 367)

Unknown directive type "function".

```
.. function:: filterfalse(predicate, iterable)

   Make an iterator that filters elements from iterable returning only those for
   which the predicate is ``False``. If *predicate* is ``None``, return the items
   that are false. Roughly equivalent to::

      def filterfalse(predicate, iterable):
          # filterfalse(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
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 382)

Unknown directive type "function".

```
.. function:: groupby(iterable, key=None)

   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.

   The operation of :func:`groupby` is similar to the ``uniq`` filter in Unix.  It
   generates a break or new group every time the value of the key function changes
   (which is why it is usually necessary to have sorted the data using the same key
```

```
function).  That behavior differs from SQL's GROUP BY which aggregates common
elements regardless of their input order.

The returned group is itself an iterator that shares the underlying iterable
with :func:`groupby`.  Because the source is shared, when the :func:`groupby`
object is advanced, the previous group is no longer visible.  So, if that data
is needed later, it should be stored as a list::

   groups = []
   uniquekeys = []
   data = sorted(data, key=keyfunc)
   for k, g in groupby(data, keyfunc):
       groups.append(list(g))      # Store group iterator as a list
       uniquekeys.append(k)

:func:`groupby` is roughly equivalent to::

   class groupby:
       # [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):
           self.id = object()
           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, self.id))
       def _grouper(self, tgtkey, id):
           while self.id is id and self.currkey == tgtkey:
               yield self.currvalue
               try:
                   self.currvalue = next(self.it)
               except StopIteration:
                   return
               self.currkey = self.keyfunc(self.currvalue)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 438)

Unknown directive type "function".

```
.. function:: islice(iterable, stop)
              islice(iterable, start, stop[, step])

   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, :func:`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).  Roughly equivalent to::

      def islice(iterable, *args):
          # islice('ABCDEFG', 2) --> A B
          # islice('ABCDEFG', 2, 4) --> C D
          # islice('ABCDEFG', 2, None) --> C D E F G
          # islice('ABCDEFG', 0, None, 2) --> A C E G
          s = slice(*args)
          start, stop, step = s.start or 0, s.stop or sys.maxsize, s.step or 1
          it = iter(range(start, stop, step))
          try:
              nexti = next(it)
          except StopIteration:
              # Consume *iterable* up to the *start* position.
              for i, element in zip(range(start), iterable):
                  pass
              return
          try:
              for i, element in enumerate(iterable):
                  if i == nexti:
                      yield element
                      nexti = next(it)
          except StopIteration:
              # Consume to *stop*.
              for i, element in zip(range(i + 1, stop), iterable):
                  pass

   If *start* is ``None``, then iteration starts at zero. If *step* is ``None``,
   then the step defaults to one.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 479)

Unknown directive type "function".

```
.. function:: pairwise(iterable)

   Return successive overlapping pairs taken from the input *iterable*.

   The number of 2-tuples in the output iterator will be one fewer than the
   number of inputs.  It will be empty if the input iterable has fewer than
   two values.

   Roughly equivalent to::

        def pairwise(iterable):
            # pairwise('ABCDEFG') --> AB BC CD DE EF FG
            a, b = tee(iterable)
            next(b, None)
            return zip(a, b)

   .. versionadded:: 3.10
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 498)

Unknown directive type "function".

```
.. function:: permutations(iterable, r=None)

   Return successive *r* length permutations of elements in the *iterable*.

   If *r* is not specified or is ``None``, then *r* defaults to the length
   of the *iterable* and all possible full-length permutations
   are generated.

   The permutation tuples are emitted in lexicographic ordering according to
   the order of the input *iterable*. 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 permutation.

   Roughly equivalent to::

        def permutations(iterable, r=None):
            # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
            # permutations(range(3)) --> 012 021 102 120 201 210
            pool = tuple(iterable)
            n = len(pool)
            r = n if r is None else r
            if r > n:
                return
            indices = list(range(n))
            cycles = list(range(n, n-r, -1))
            yield tuple(pool[i] for i in indices[:r])
            while n:
                for i in reversed(range(r)):
                    cycles[i] -= 1
                    if cycles[i] == 0:
                        indices[i:] = indices[i+1:] + indices[i:i+1]
                        cycles[i] = n - i
                    else:
                        j = cycles[i]
                        indices[i], indices[-j] = indices[-j], indices[i]
                        yield tuple(pool[i] for i in indices[:r])
                        break
                else:
                    return

   The code for :func:`permutations` can be also expressed as a subsequence of
   :func:`product`, filtered to exclude entries with repeated elements (those
   from the same position in the input pool)::

        def permutations(iterable, r=None):
            pool = tuple(iterable)
            n = len(pool)
            r = n if r is None else r
            for indices in product(range(n), repeat=r):
                if len(set(indices)) == r:
                    yield tuple(pool[i] for i in indices)

   The number of items returned is ``n! / (n-r)!`` when ``0 <= r <= n``
   or zero when ``r > n``.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 556)

Unknown directive type "function".

```
.. function:: product(*iterables, repeat=1)

   Cartesian product of input iterables.

   Roughly equivalent to nested for-loops in a generator expression. For example,
   ``product(A, B)`` returns the same as ``((x,y) for x in A for y in B)``.

   The nested loops cycle like an odometer with the rightmost element advancing
   on every iteration.  This pattern creates a lexicographic ordering so that if
   the input's iterables are sorted, the product tuples are emitted in sorted
   order.

   To compute the product of an iterable with itself, specify the number of
   repetitions with the optional *repeat* keyword argument.  For example,
   ``product(A, repeat=4)`` means the same as ``product(A, A, A, A)``.

   This function is roughly equivalent to the following code, except that the
   actual implementation does not build up intermediate results in memory::

       def product(*args, repeat=1):
           # product('ABCD', 'xy') --> Ax Ay Bx By Cx Cy Dx Dy
           # product(range(2), repeat=3) --> 000 001 010 011 100 101 110 111
           pools = [tuple(pool) for pool in args] * repeat
           result = [[]]
           for pool in pools:
               result = [x+[y] for x in result for y in pool]
           for prod in result:
               yield tuple(prod)

   Before :func:`product` runs, it completely consumes the input iterables,
   keeping pools of values in memory to generate the products.  Accordingly,
   it is only useful with finite inputs.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 589)

Unknown directive type "function".

```
.. function:: repeat(object[, times])

   Make an iterator that returns *object* over and over again. Runs indefinitely
   unless the *times* argument is specified. Used as argument to :func:`map` for
   invariant parameters to the called function.  Also used with :func:`zip` to
   create an invariant part of a tuple record.

   Roughly equivalent to::

      def repeat(object, times=None):
          # repeat(10, 3) --> 10 10 10
          if times is None:
              while True:
                  yield object
          else:
              for i in range(times):
                  yield object

   A common use for *repeat* is to supply a stream of constant values to *map*
   or *zip*::

      >>> list(map(pow, range(10), repeat(2)))
      [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 613)

Unknown directive type "function".

```
.. function:: starmap(function, iterable)

   Make an iterator that computes the function using arguments obtained from
   the iterable.  Used instead of :func:`map` when argument parameters are already
   grouped in tuples from a single iterable (the data has been "pre-zipped").  The
   difference between :func:`map` and :func:`starmap` parallels the distinction
   between ``function(a,b)`` and ``function(*c)``. Roughly equivalent to::

      def starmap(function, iterable):
          # starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000
          for args in iterable:
              yield function(*args)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 627)

Unknown directive type "function".

```
.. function:: takewhile(predicate, iterable)

   Make an iterator that returns elements from the iterable as long as the
   predicate is true.  Roughly equivalent to::

      def takewhile(predicate, iterable):
          # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
          for x in iterable:
              if predicate(x):
                  yield x
              else:
                  break
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 641)

Unknown directive type "function".

```
.. function:: tee(iterable, n=2)

   Return *n* independent iterators from a single iterable.

   The following Python code helps explain what *tee* does (although the actual
   implementation is more complex and uses only a single underlying
   :abbr:`FIFO (first-in, first-out)` queue).

   Roughly equivalent to::

        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
                        try:
                            newval = next(it)   # fetch a new value and
                        except StopIteration:
                            return
                        for d in deques:        # load it to all the deques
                            d.append(newval)
                    yield mydeque.popleft()
            return tuple(gen(d) for d in deques)

   Once :func:`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.

   ``tee`` iterators are not threadsafe. A :exc:`RuntimeError` may be
   raised when using simultaneously iterators returned by the same :func:`tee`
   call, even if the original *iterable* is threadsafe.

   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
   :func:`list` instead of :func:`tee`.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 680)

Unknown directive type "function".

```
.. function:: zip_longest(*iterables, fillvalue=None)

   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.  Roughly equivalent to::

      def zip_longest(*args, fillvalue=None):
          # zip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-
          iterators = [iter(it) for it in args]
          num_active = len(iterators)
          if not num_active:
              return
          while True:
              values = []
              for i, it in enumerate(iterators):
                  try:
                      value = next(it)
                  except StopIteration:
                      num_active -= 1
                      if not num_active:
                          return
                      iterators[i] = repeat(fillvalue)
                      value = fillvalue
                  values.append(value)
              yield tuple(values)

   If one of the iterables is potentially infinite, then the :func:`zip_longest`
```

```
function should be wrapped with something that limits the number of calls
(for example :func:`islice` or :func:`takewhile`).  If not specified,
*fillvalue* defaults to ``None``.
```

Itertools Recipes

This section shows recipes for creating an extended toolset using the existing itertools as building blocks.

Substantially all of these recipes and many, many others can be installed from the more-itertools project found on the Python Package Index:

```
pip install more-itertools
```

The extended tools offer the same high performance as the underlying toolset. The superior memory performance is kept by processing elements one at a time rather than bringing the whole iterable into memory all at once. Code volume is kept small by linking the tools together in a functional style which helps eliminate temporary variables. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and :term:`generator`s which incur interpreter overhead.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 726); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 734)

Unknown directive type "testcode".

```
.. testcode::

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

   def prepend(value, iterator):
       "Prepend a single value in front of an iterator"
       # prepend(1, [2, 3, 4]) -> 1 2 3 4
       return chain([value], iterator)

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

   def tail(n, iterable):
       "Return an iterator over the last n items"
       # tail(3, 'ABCDEFG') --> E F G
       return iter(collections.deque(iterable, maxlen=n))

   def consume(iterator, n=None):
       "Advance the iterator n-steps ahead. If n is None, consume entirely."
       # Use functions that consume iterators at C speed.
       if n is None:
           # feed the entire iterator into a zero-length deque
           collections.deque(iterator, maxlen=0)
       else:
           # advance to the empty slice starting at position n
           next(islice(iterator, n, n), None)

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

   def all_equal(iterable):
       "Returns True if all the elements are equal to each other"
       g = groupby(iterable)
       return next(g, True) and not next(g, False)

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

   def pad_none(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(tuple(iterable), n))

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

   def convolve(signal, kernel):
       # See:  https://betterexplained.com/articles/intuitive-convolution/
       # convolve(data, [0.25, 0.25, 0.25, 0.25]) --> Moving average (blur)
```

```
        # convolve(data, [1, -1]) --> 1st finite difference (1st derivative)
        # convolve(data, [1, -2, 1]) --> 2nd finite difference (2nd derivative)
        kernel = tuple(kernel)[::-1]
        n = len(kernel)
        window = collections.deque([0], maxlen=n) * n
        for x in chain(signal, repeat(0, n-1)):
            window.append(x)
            yield sum(map(operator.mul, kernel, window))

    def flatten(list_of_lists):
        "Flatten one level of nesting"
        return chain.from_iterable(list_of_lists)

    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 grouper(iterable, n, *, incomplete='fill', fillvalue=None):
        "Collect data into non-overlapping fixed-length chunks or blocks"
        # grouper('ABCDEFG', 3, fillvalue='x') --> ABC DEF Gxx
        # grouper('ABCDEFG', 3, incomplete='strict') --> ABC DEF ValueError
        # grouper('ABCDEFG', 3, incomplete='ignore') --> ABC DEF
        args = [iter(iterable)] * n
        if incomplete == 'fill':
            return zip_longest(*args, fillvalue=fillvalue)
        if incomplete == 'strict':
            return zip(*args, strict=True)
        if incomplete == 'ignore':
            return zip(*args)
        else:
            raise ValueError('Expected fill, strict, or ignore')

    def triplewise(iterable):
        "Return overlapping triplets from an iterable"
        # triplewise('ABCDEFG') -> ABC BCD CDE DEF EFG
        for (a, _), (b, c) in pairwise(pairwise(iterable)):
            yield a, b, c

    def sliding_window(iterable, n):
        # sliding_window('ABCDEFG', 4) -> ABCD BCDE CDEF DEFG
        it = iter(iterable)
        window = collections.deque(islice(it, n), maxlen=n)
        if len(window) == n:
            yield tuple(window)
        for x in it:
            window.append(x)
            yield tuple(window)

    def roundrobin(*iterables):
        "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
        # Recipe credited to George Sakkis
        num_active = len(iterables)
        nexts = cycle(iter(it).__next__ for it in iterables)
        while num_active:
            try:
                for next in nexts:
                    yield next()
            except StopIteration:
                # Remove the iterator we just exhausted from the cycle.
                num_active -= 1
                nexts = cycle(islice(nexts, num_active))

    def partition(pred, iterable):
        "Use a predicate to partition entries into false entries and true entries"
        # partition(is_odd, range(10)) --> 0 2 4 6 8   and  1 3 5 7 9
        t1, t2 = tee(iterable)
        return filterfalse(pred, t1), filter(pred, t2)

    def before_and_after(predicate, it):
        """ Variant of takewhile() that allows complete
            access to the remainder of the iterator.

            >>> it = iter('ABCdEfGhI')
            >>> all_upper, remainder = before_and_after(str.isupper, it)
            >>> ''.join(all_upper)
            'ABC'
            >>> ''.join(remainder)     # takewhile() would lose the 'd'
            'dEfGhI'

            Note that the first iterator must be fully
            consumed before the second iterator can
            generate valid results.
        """
        it = iter(it)
        transition = []
        def true_iterator():
            for elem in it:
                if predicate(elem):
                    yield elem
```

```
                else:
                    transition.append(elem)
                    return
        def remainder_iterator():
            yield from transition
            yield from it
        return true_iterator(), remainder_iterator()

    def subslices(seq):
        "Return all contiguous non-empty subslices of a sequence"
        # subslices('ABCD') --> A AB ABC ABCD B BC BCD C CD D
        slices = starmap(slice, combinations(range(len(seq) + 1), 2))
        return map(operator.getitem, repeat(seq), slices)

    def powerset(iterable):
        "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
        s = list(iterable)
        return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

    def unique_everseen(iterable, key=None):
        "List unique elements, preserving order. Remember all elements ever seen."
        # unique_everseen('AAAABBBCCDAABBB') --> A B C D
        # unique_everseen('ABBCcAD', str.lower) --> A B C D
        seen = set()
        seen_add = seen.add
        if key is None:
            for element in filterfalse(seen.__contains__, iterable):
                seen_add(element)
                yield element
        else:
            for element in iterable:
                k = key(element)
                if k not in seen:
                    seen_add(k)
                    yield element

    def unique_justseen(iterable, key=None):
        "List unique elements, preserving order. Remember only the element just seen."
        # unique_justseen('AAAABBBCCDAABBB') --> A B C D A B
        # unique_justseen('ABBCcAD', str.lower) --> A B C A D
        return map(next, map(operator.itemgetter(1), groupby(iterable, key)))

    def iter_except(func, exception, first=None):
        """ Call a function repeatedly until an exception is raised.

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

        Examples:
            iter_except(functools.partial(heappop, h), IndexError)   # priority queue iterator
            iter_except(d.popitem, KeyError)                         # non-blocking dict iterator
            iter_except(d.popleft, IndexError)                       # non-blocking deque iterator
            iter_except(q.get_nowait, Queue.Empty)                   # loop over a producer Queue
            iter_except(s.pop, KeyError)                             # non-blocking set iterator

        """
        try:
            if first is not None:
                yield first()            # For database APIs needing an initial cast to db.first()
            while True:
                yield func()
        except exception:
            pass

    def 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) is true.

        """
        # first_true([a,b,c], x) --> a or b or c or x
        # first_true([a,b], x, f) --> a if f(a) else b if f(b) else x
        return next(filter(pred, iterable), default)

    def random_product(*args, repeat=1):
        "Random selection from itertools.product(*args, **kwds)"
        pools = [tuple(pool) for pool in args] * repeat
        return tuple(map(random.choice, pools))

    def random_permutation(iterable, r=None):
        "Random selection from itertools.permutations(iterable, r)"
        pool = tuple(iterable)
        r = len(pool) if r is None else r
        return tuple(random.sample(pool, r))

    def random_combination(iterable, r):
        "Random selection from itertools.combinations(iterable, r)"
        pool = tuple(iterable)
        n = len(pool)
        indices = sorted(random.sample(range(n), r))
```

```
        return tuple(pool[i] for i in indices)

    def random_combination_with_replacement(iterable, r):
        "Random selection from itertools.combinations_with_replacement(iterable, r)"
        pool = tuple(iterable)
        n = len(pool)
        indices = sorted(random.choices(range(n), k=r))
        return tuple(pool[i] for i in indices)

    def nth_combination(iterable, r, index):
        "Equivalent to list(combinations(iterable, r))[index]"
        pool = tuple(iterable)
        n = len(pool)
        if r < 0 or r > n:
            raise ValueError
        c = 1
        k = min(r, n-r)
        for i in range(1, k+1):
            c = c * (n - k + i) // i
        if index < 0:
            index += c
        if index < 0 or index >= c:
            raise IndexError
        result = []
        while r:
            c, n, r = c*r//n, n-1, r-1
            while index >= c:
                index -= c
                c, n = c*(n-r)//n, n-1
            result.append(pool[-1-n])
        return tuple(result)
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]itertools.rst, line 1014)

Unknown directive type "doctest".

```
.. doctest::
    :hide:

    These examples no longer appear in the docs but are guaranteed
    to keep working.

    >>> amounts = [120.15, 764.05, 823.14]
    >>> for checknum, amount in zip(count(1200), amounts):
    ...     print('Check %d is for $%.2f' % (checknum, amount))
    ...
    Check 1200 is for $120.15
    Check 1201 is for $764.05
    Check 1202 is for $823.14

    >>> import operator
    >>> for cube in map(operator.pow, range(1,4), repeat(3)):
    ...    print(cube)
    ...
    1
    8
    27

    >>> reportlines = ['EuroPython', 'Roster', '', 'alex', '', 'laura', '', 'martin', '', 'walter', '', 's
    >>> for name in islice(reportlines, 3, None, 2):
    ...    print(name.title())
    ...
    Alex
    Laura
    Martin
    Walter
    Samuele

    >>> from operator import itemgetter
    >>> d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
    >>> di = sorted(sorted(d.items()), key=itemgetter(1))
    >>> for k, g in groupby(di, itemgetter(1)):
    ...     print(k, list(map(itemgetter(0), g)))
    ...
    1 ['a', 'c', 'e']
    2 ['b', 'd', 'f']
    3 ['g']

    # Find runs of consecutive numbers using groupby.  The key to the solution
    # is differencing with a range so that consecutive numbers all appear in
    # same group.
    >>> data = [ 1,  4,5,6, 10, 15,16,17,18, 22, 25,26,27,28]
    >>> for k, g in groupby(enumerate(data), lambda t:t[0]-t[1]):
    ...     print(list(map(operator.itemgetter(1), g)))
    ...
    [1]
    [4, 5, 6]
    [10]
    [15, 16, 17, 18]
    [22]
    [25, 26, 27, 28]
```

```
Now, we test all of the itertool recipes

>>> import operator
>>> import collections

>>> take(10, count())
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(prepend(1, [2, 3, 4]))
[1, 2, 3, 4]

>>> list(enumerate('abc'))
[(0, 'a'), (1, 'b'), (2, 'c')]

>>> list(islice(tabulate(lambda x: 2*x), 4))
[0, 2, 4, 6]

>>> list(tail(3, 'ABCDEFG'))
['E', 'F', 'G']

>>> it = iter(range(10))
>>> consume(it, 3)
>>> next(it)
3
>>> consume(it)
>>> next(it, 'Done')
'Done'

>>> nth('abcde', 3)
'd'

>>> nth('abcde', 9) is None
True

>>> [all_equal(s) for s in ('', 'A', 'AAAA', 'AAAB', 'AAABA')]
[True, True, True, False, False]

>>> quantify(range(99), lambda x: x%2==0)
50

>>> quantify([True, False, False, True, True])
3

>>> quantify(range(12), pred=lambda x: x%2==1)
6

>>> a = [[1, 2, 3], [4, 5, 6]]
>>> list(flatten(a))
[1, 2, 3, 4, 5, 6]

>>> list(repeatfunc(pow, 5, 2, 3))
[8, 8, 8, 8, 8]

>>> import random
>>> take(5, map(int, repeatfunc(random.random)))
[0, 0, 0, 0, 0]

>>> list(islice(pad_none('abc'), 0, 6))
['a', 'b', 'c', None, None, None]

>>> list(ncycles('abc', 3))
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']

>>> dotproduct([1,2,3], [4,5,6])
32

>>> data = [20, 40, 24, 32, 20, 28, 16]
>>> list(convolve(data, [0.25, 0.25, 0.25, 0.25]))
[5.0, 15.0, 21.0, 29.0, 29.0, 26.0, 24.0, 16.0, 11.0, 4.0]
>>> list(convolve(data, [1, -1]))
[20, 20, -16, 8, -12, 8, -12, -16]
>>> list(convolve(data, [1, -2, 1]))
[20, 0, -36, 24, -20, 20, -20, -4, 16]

>>> list(flatten([('a', 'b'), (), ('c', 'd', 'e'), ('f',), ('g', 'h', 'i')]))
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

>>> import random
>>> random.seed(85753098575309)
>>> list(repeatfunc(random.random, 3))
[0.16370491282496968, 0.45889608687313455, 0.3747076837820118]
>>> list(repeatfunc(chr, 3, 65))
['A', 'A', 'A']
>>> list(repeatfunc(pow, 3, 2, 5))
[32, 32, 32]

>>> list(grouper('abcdefg', 3, fillvalue='x'))
[('a', 'b', 'c'), ('d', 'e', 'f'), ('g', 'x', 'x')]

>>> it = grouper('abcdefg', 3, incomplete='strict')
>>> next(it)
('a', 'b', 'c')
>>> next(it)
```

```
('d', 'e', 'f')
>>> next(it)
Traceback (most recent call last):
  ...
ValueError: zip() argument 2 is shorter than argument 1

>>> list(grouper('abcdefg', n=3, incomplete='ignore'))
[('a', 'b', 'c'), ('d', 'e', 'f')]

>>> list(triplewise('ABCDEFG'))
[('A', 'B', 'C'), ('B', 'C', 'D'), ('C', 'D', 'E'), ('D', 'E', 'F'), ('E', 'F', 'G')]

>>> list(sliding_window('ABCDEFG', 4))
[('A', 'B', 'C', 'D'), ('B', 'C', 'D', 'E'), ('C', 'D', 'E', 'F'), ('D', 'E', 'F', 'G')]

>>> list(roundrobin('abc', 'd', 'ef'))
['a', 'd', 'e', 'b', 'f', 'c']

>>> def is_odd(x):
...     return x % 2 == 1

>>> evens, odds = partition(is_odd, range(10))
>>> list(evens)
[0, 2, 4, 6, 8]
>>> list(odds)
[1, 3, 5, 7, 9]

>>> it = iter('ABCdEfGhI')
>>> all_upper, remainder = before_and_after(str.isupper, it)
>>> ''.join(all_upper)
'ABC'
>>> ''.join(remainder)
'dEfGhI'

>>> list(subslices('ABCD'))
['A', 'AB', 'ABC', 'ABCD', 'B', 'BC', 'BCD', 'C', 'CD', 'D']

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

>>> all(len(list(powerset(range(n)))) == 2**n for n in range(18))
True

>>> list(powerset('abcde')) == sorted(sorted(set(powerset('abcde'))), key=len)
True

>>> list(unique_everseen('AAAABBBCCDAABBB'))
['A', 'B', 'C', 'D']

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

>>> list(unique_justseen('AAAABBBCCDAABBB'))
['A', 'B', 'C', 'D', 'A', 'B']

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

>>> d = dict(a=1, b=2, c=3)
>>> it = iter_except(d.popitem, KeyError)
>>> d['d'] = 4
>>> next(it)
('d', 4)
>>> next(it)
('c', 3)
>>> next(it)
('b', 2)
>>> d['e'] = 5
>>> next(it)
('e', 5)
>>> next(it)
('a', 1)
>>> next(it, 'empty')
'empty'

>>> first_true('ABC0DEF1', '9', str.isdigit)
'0'

>>> population = 'ABCDEFGH'
>>> for r in range(len(population) + 1):
...     seq = list(combinations(population, r))
...     for i in range(len(seq)):
...         assert nth_combination(population, r, i) == seq[i]
...     for i in range(-len(seq), 0):
...         assert nth_combination(population, r, i) == seq[i]

>>> iterable = 'abcde'
>>> r = 3
>>> combos = list(combinations(iterable, r))
>>> all(nth_combination(iterable, r, i) == comb for i, comb in enumerate(combos))
True
```