

Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [10. Numeric and Mathematical Modules](#) »

10.7. itertools — Functions creating iterators for efficient looping

New in version 2.3.

This module implements a number of [iterator](#) building blocks inspired by constructs from APL, Haskell, and SML. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence $f(0), f(1), \dots$. The same effect can be achieved in Python by combining [imap\(\)](#) and [count\(\)](#) to form `imap(f, count())`.

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

Infinite Iterators:

Iterator	Arguments	Results	Example
count()	start	start, start+1, start+2, ...	<code>count(10) --> 10 11 12 13 14 ...</code>
cycle()	p	p0, p1, ... plast, p0, p1, ...	<code>cycle('ABCD') --> A B C D A B C D ...</code>
repeat()	elem [,n]	elem, elem, elem, ... endlessly or up to n times	<code>repeat(10, 3) --> 10 10 10</code>

Iterators terminating on the shortest input sequence:

Iterator	Arguments	Results	Example
chain()	p, q, ...	p0, p1, ... plast, q0, q1, ...	<code>chain('ABC', 'DEF') --> A B C D E F</code>
dropwhile()	pred, seq	seq[n], seq[n+1], starting when pred fails	<code>dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1</code>
groupby()	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
ifilter()	pred, seq	elements of seq where pred(elem) is True	<code>ifilter(lambda x: x%2, range(10)) --> 1 3 5 7 9</code>
ifilterfalse()	pred, seq	elements of seq where pred(elem) is False	<code>ifilterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8</code>
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	<code>islice('ABCDEFG', 2, None) --> C D E F G</code>
imap()	func, p, q, ...	func(p0, q0), func(p1, q1), ...	<code>imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000</code>
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	<code>starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000</code>
tee()	it, n	it1, it2, ... itn splits one iterator into n	
takewhile()	pred, seq	seq[0], seq[1], until pred fails	<code>takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4</code>
izip()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>izip('ABCD', 'xy') --> Ax By</code>
izip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>izip_longest('ABCD', 'xy', fillvalue='-') --> Ax By C-D-</code>

Combinatoric generators:

Iterator	Arguments	Results
product()	p, q, ... [repeat=1]	cartesian product, equivalent to a nested for-loop
permutations()	p[, r]	r-length tuples, all possible orderings, no repeated elements
combinations()	p, r	r-length tuples, in sorted order, no repeated elements

<code>product('ABCD', repeat=2)</code>		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>		AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>		AB AC AD BC BD CD

10.7.1. 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.

`itertools.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. Equivalent to:

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

`itertools.chain.from_iterable(iterable)`

Alternate constructor for `chain()`. Gets chained inputs from a single iterable argument that is evaluated lazily. Equivalent to:

```
@classmethod
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
```

New in version 2.6.

`itertools.combinations(iterable, r)`

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

Combinations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the combination tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. So if the input elements are unique, there will be no repeat values in each combination.

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 = 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 `combinations()` can be also expressed as a subsequence of `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 $0 \leq r \leq n$ or zero when $r > n$.

New in version 2.6.

`itertools.count(n)`

Make an iterator that returns consecutive integers starting with *n*. If not specified *n* defaults to zero. Often used as an argument to [imap\(\)](#) to generate consecutive data points. Also, used with [izip\(\)](#) to add sequence numbers. Equivalent to:

```
def count(n=0):
    # count(10) --> 10 11 12 13 14 ...
    while True:
        yield n
        n += 1
```

`itertools.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. 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).

`itertools.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. 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
```

`itertools.groupby(iterable[, key])`

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 [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 [groupby\(\)](#). Because the source is shared, when the [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)
```

[groupby\(\)](#) is equivalent to:

```

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

```

New in version 2.4.

`itertools.ifilter(predicate, iterable)`[¶](#)

Make an iterator that filters elements from *iterable* returning only those for which the predicate is `True`. If *predicate* is `None`, return the items that are true. Equivalent to:

```

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

```

`itertools.ifilterfalse(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. Equivalent to:

```

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

```

`itertools.imap(function, *iterables)`[¶](#)

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

```

def imap(function, *iterables):
    # imap(pow, (2,3,10), (5,2,3)) --> 32 9 1000
    iterables = map(iter, iterables)
    while True:
        args = [next(it) for it in iterables]
        if function is None:
            yield tuple(args)
        else:
            yield function(*args)

```

`itertools.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, `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). Equivalent to:

```
def islice(iterable, *args):
    # islice('ABCDEFGH', 2) --> A B
    # islice('ABCDEFGH', 2, 4) --> C D
    # islice('ABCDEFGH', 2, None) --> C D E F G
    # islice('ABCDEFGH', 0, None, 2) --> A C E G
    s = slice(*args)
    it = iter(xrange(s.start or 0, s.stop or sys.maxint, s.step or 1))
    nexti = next(it)
    for i, element in enumerate(iterable):
        if i == nexti:
            yield element
            nexti = next(it)
```

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

Changed in version 2.5: accept `None` values for default *start* and *step*.

`itertools.izip(*iterables)`¶

Make an iterator that aggregates elements from each of the iterables. Like `zip()` except that it returns an iterator instead of a list. Used for lock-step iteration over several iterables at a time. Equivalent to:

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

Changed in version 2.4: When no iterables are specified, returns a zero length iterator instead of raising a `TypeError` exception.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using `izip(*[iter(s)]*n)`.

`izip()` should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables. If those values are important, use `izip_longest()` instead.

`itertools.izip_longest(*iterables[, fillvalue])`¶

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. Equivalent to:

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

If one of the iterables is potentially infinite, then the `izip_longest()` function should be wrapped with something that limits the number of calls (for example `islice()` or `takewhile()`). If not specified, *fillvalue* defaults to `None`.

New in version 2.6.

`itertools.permutations(iterable[, r])`¶

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.

Permutations are emitted in lexicographic sort order. So, if the input *iterable* is sorted, the permutation 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.

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 = range(n)
    cycles = 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 [permutations\(\)](#) can be also expressed as a subsequence of [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 $0 \leq r \leq n$ or zero when $r > n$.

New in version 2.6.

`itertools.product(*iterables, repeat=1)`

Cartesian product of input iterables.

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

```
def product(*args, **kwargs):
    # 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 = map(tuple, args) * kwargs.get('repeat', 1)
    result = [[]]
    for pool in pools:
        result = [x+[y] for x in result for y in pool]
    for prod in result:
        yield tuple(prod)
```

New in version 2.6.

`itertools.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 [imap\(\)](#) for invariant function parameters. Also used with [izip\(\)](#) to create constant fields in a tuple record. 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 xrange(times):
            yield object
```

`itertools.starmap(function, iterable)`

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

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

Changed in version 2.6: Previously, [starmap\(\)](#) required the function arguments to be tuples. Now, any iterable is allowed.

`itertools.takewhile(predicate, iterable)`

Make an iterator that returns elements from the iterable as long as the predicate is true. 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
```

`itertools.tee(iterable, n=2)`

Return *n* independent iterators from a single iterable. Equivalent to:

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

Once [tee\(\)](#) has made a split, the original *iterable* should not be used anywhere else; otherwise, the *iterable* could get advanced without the tee objects being informed.

This itertools may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use [list\(\)](#) instead of [tee\(\)](#).

New in version 2.4.

10.7.2. Examples

The following examples show common uses for each tool and demonstrate ways they can be combined.

```
>>> # Show a dictionary sorted and grouped by value
>>> from operator import itemgetter
>>> d = dict(a=1, b=2, c=1, d=2, e=1, f=2, g=3)
>>> di = sorted(d.iteritems(), key=itemgetter(1))
>>> for k, g in groupby(di, key=itemgetter(1)):
...     print k, map(itemgetter(0), g)
```

```

...
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 (i,x):i-x):
...     print map(itemgetter(1), g)
...
[1]
[4, 5, 6]
[10]
[15, 16, 17, 18]
[22]
[25, 26, 27, 28]

```

10.7.3. Recipes

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

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 [generators](#) which incur interpreter overhead.

```

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

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

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

def consume(iterator, n):
    "Advance the iterator n-steps ahead. If n is none, consume entirely."
    # The technique uses objects 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 quantify(iterable, pred=bool):
    "Count how many times the predicate is true"
    return sum(imap(pred, iterable))

def padnone(iterable):
    """Returns the sequence elements and then returns None indefinitely.

    Useful for emulating the behavior of the built-in map() function.
    """
    return chain(iterable, repeat(None))

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

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

```



```

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

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 pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)

def grouper(n, iterable, fillvalue=None):
    "grouper(3, 'ABCDEFG', 'x') --> ABC DEF Gxx"
    args = [iter(iterable)] * n
    return izip_longest(fillvalue=fillvalue, *args)

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

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

def combinations_with_replacement(iterable, r):
    "combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC"
    # number items returned: (n+r-1)! / r! / (n-1)!
    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)

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('ABCCAD', str.lower) --> A B C D
    seen = set()

```

```

seen_add = seen.add
if key is None:
    for element in iterable:
        if element not in seen:
            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 imap(next, imap(itemgetter(1), groupby(iterable, key)))

```

Note, many of the above recipes can be optimized by replacing global lookups with local variables defined as default values. For example, the *dotproduct* recipe can be written as:

```

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

```

[Table Of Contents](#)

[10.7. itertools — Functions creating iterators for efficient looping](#)

- [10.7.1. Itertool functions](#)
- [10.7.2. Examples](#)
- [10.7.3. Recipes](#)

Previous topic

[10.6. random — Generate pseudo-random numbers](#)

Next topic

[10.8. functools — Higher order functions and operations on callable objects](#)

This Page

- [Show Source](#)

Navigation

- [index](#)
- [modules](#) |
- [next](#) |
- [previous](#) |
- [Python v2.6.4 documentation](#) »
- [The Python Standard Library](#) »
- [10. Numeric and Mathematical Modules](#) »

© [Copyright](#) 1990-2010, Python Software Foundation.

The Python Software Foundation is a non-profit corporation. [Please donate.](#)

Last updated on Feb 26, 2010. Created using [Sphinx](#) 0.6.3.