itertools - Functional tools for creating and using iterators.

**count(start=0, step=1)** --> start, start+step, start+2\*step, ...

#count(10) --> 10 11 12 13 14 ...

**cycle(p)** --> p0, p1, ... plast, p0, p1, ...

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

**repeat(elem [,n])** --> elem, elem, elem, ... endlessly or up to n times

#repeat(10, 3) --> 10 10 10

**Iterators terminating on the shortest input sequence:**

accumulate(p[, func]) --> p0, p0+p1, p0+p1+p2

#accumulate([1,2,3,4,5]) --> 1 3 6 10 15

**chain**(p, q, ...) --> p0, p1, ... plast, q0, q1, ...

#chain('ABC', 'DEF') --> A B C D E F

#chain.from\_iterable(['ABC', 'DEF'])

**compress**(iterable, Boolean)

#输出iterable对象中对应选择器为True的元素

addresses **=** [

'5412 N CLARK',

'5148 N CLARK',

'5800 E 58TH',

'2122 N CLARK',

'5645 N RAVENSWOOD',

'1060 W ADDISON',

'4801 N BROADWAY',

'1039 W GRANVILLE',

]

counts **=** [ 0, 3, 10, 4, 1, 7, 6, 1]

>>> more5 **=** [n **>** 5 **for** n **in** counts]

>>> more5

[False, False, True, False, False, True, True, False]

>>> list(compress(addresses, more5))

['5800 E 58TH', '1060 W ADDISON', '4801 N BROADWAY']

**dropwhile(pred, seq)** --> seq[n], seq[n+1], starting **when pred fails**

#dropwhile(lambda x: x<5, [1,4,6,4,1]) --> 6 4 1

**takewhile(pred, seq)** --> seq[0], seq[1], **until pred fails**

#takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4

**groupby**(iterable[, keyfunc]) --> 通过某个字段将记录分组

#groupby函数扫描整个序列并且查找**连续相同值，因此要先排序**

rows **=** [

{'address': '5412 N CLARK', 'date': '07/01/2012'},

{'address': '5148 N CLARK', 'date': '07/04/2012'},

{'address': '5800 E 58TH', 'date': '07/02/2012'},

{'address': '2122 N CLARK', 'date': '07/03/2012'},

{'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},

{'address': '1060 W ADDISON', 'date': '07/02/2012'},

{'address': '4801 N BROADWAY', 'date': '07/01/2012'},

{'address': '1039 W GRANVILLE', 'date': '07/04/2012'},

]

**from** operator **import** itemgetter

**from** itertools **import** groupby

rows**.**sort(key**=**itemgetter('date'))

**for** date, items **in** groupby(rows, key**=**itemgetter('date')):

**print**(date)

**for** i **in** items:

**print**(' ', i)

07**/**01**/**2012

{'date': '07/01/2012', 'address': '5412 N CLARK'}

{'date': '07/01/2012', 'address': '4801 N BROADWAY'}

07**/**02**/**2012

{'date': '07/02/2012', 'address': '5800 E 58TH'}

{'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}

{'date': '07/02/2012', 'address': '1060 W ADDISON'}

07**/**03**/**2012

{'date': '07/03/2012', 'address': '2122 N CLARK'}

07**/**04**/**2012

{'date': '07/04/2012', 'address': '5148 N CLARK'}

{'date': '07/04/2012', 'address': '1039 W GRANVILLE'}

**filterfalse**(pred, seq) --> elements of seq where pred(elem) is False

#filterfalse(lambda x: x%2, range(10)) --> 0 2 4 6 8

**islice**(seq, [start,] stop [, step]) --> elements from seq[start:stop:step]

#islice('ABCDEFG', 2, None) --> C D E F G

starmap(fun, seq) --> fun(\*seq[0]), fun(\*seq[1]), ...

#starmap(pow, [(2,5), (3,2), (10,3)]) --> 32 9 1000

tee(it, n=2) --> (it1, it2 , ... itn) 克隆

#从iterable创建n个独立的迭代器，创建的迭代器以n元组的形式返回

**zip\_longest**(p, q, ...) --> (p[0], q[0]), (p[1], q[1]), ...

#zip\_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-

**Combinatoric generators:**

**product**(p, q, ... [repeat=1]) --> **cartesian product**

#product('ABCD', repeat=2)

结果：AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD

**permutations**(p[, r]): r-length tuples, all possible orderings, no repeated elements

#permutations('ABCD', 2) 排列

AB AC AD BA BC BD CA CB CD DA DB DC

**combinations**(p, r): r-length tuples, in sorted order, no repeated elements

#combinations('ABCD', 2) 组合

AB AC AD BC BD CD

**combinations\_with\_replacement**(p, r) : r-length tuples, in sorted order, with repeated elements

#combinations\_with\_replacement('ABCD', 2)

AA AB AC AD BB BC BD CC CD DD

itertools.**accumulate**(iterable[, func])

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 [Decimal](https://docs.python.org/3/library/decimal.html#decimal.Decimal) or [Fraction](https://docs.python.org/3/library/fractions.html#fractions.Fraction).) If the input iterable is empty, the output iterable will also be empty.

Roughly equivalent to:

**def** accumulate(iterable, func=operator.add):

'Return running totals'

*# accumulate([1,2,3,4,5]) --> 1 3 6 10 15*

*# accumulate([1,2,3,4,5], operator.mul) --> 1 2 6 24 120*

it = iter(iterable)

**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 [min()](https://docs.python.org/3/library/functions.html#min) for a running minimum, [max()](https://docs.python.org/3/library/functions.html#max) for a running maximum, or [operator.mul()](https://docs.python.org/3/library/operator.html#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 [functools.reduce()](https://docs.python.org/3/library/functools.html#functools.reduce) for a similar function that returns only the final accumulated value.

New in version 3.2.

Changed in version 3.3: Added the optional func parameter.

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. 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

classmethod chain.**from\_iterable**(iterable)¶

Alternate constructor for 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

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.

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 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 <= r <= n or zero when r > n.

itertools.**combinations\_with\_replacement**(iterable, r)¶

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

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, 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 combinations\_with\_replacement() can be also expressed as a subsequence of 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.

New in version 3.1.

itertools.**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)

New in version 3.1.

itertools.**count**(start=0, step=1)¶

Make an iterator that returns evenly spaced values starting with number start. Often used as an argument to [map()](https://docs.python.org/3/library/functions.html#map) to generate consecutive data points. Also, used with [zip()](https://docs.python.org/3/library/functions.html#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()).

Changed in version 3.1: Added step argument and allowed non-integer arguments.

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. 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).

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. 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

itertools.**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

itertools.**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 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 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):

**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

**try**:

self.currvalue = next(self.it)

**except** StopIteration:

**return**

self.currkey = self.keyfunc(self.currvalue)

itertools.**islice**(iterable, stop)¶

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). 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)

it = iter(range(s.start **or** 0, s.stop **or** sys.maxsize, s.step **or** 1))

**try**:

nexti = next(it)

**except** StopIteration:

**return**

**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.

itertools.**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.

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.

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 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 <= r <= n or zero when r > n.

itertools.**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)

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 [map()](https://docs.python.org/3/library/functions.html#map) for invariant parameters to the called function. Also used with [zip()](https://docs.python.org/3/library/functions.html#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]

itertools.**starmap**(function, iterable)¶

Make an iterator that computes the function using arguments obtained from the iterable. Used instead of [map()](https://docs.python.org/3/library/functions.html#map) when argument parameters are already grouped in tuples from a single iterable (the data has been “pre-zipped”). The difference between [map()](https://docs.python.org/3/library/functions.html#map) and 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)

itertools.**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**

itertools.**tee**(iterable, n=2)¶

Return n independent iterators from a single iterable. 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 tee() has made a split, the original iterable should not be used anywhere else; otherwise, the iterable could get advanced without the tee objects being informed.

This itertool may require significant auxiliary storage (depending on how much temporary data needs to be stored). In general, if one iterator uses most or all of the data before another iterator starts, it is faster to use [list()](https://docs.python.org/3/library/stdtypes.html#list) instead of tee().

itertools.**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:

**class** **ZipExhausted**(Exception):

**pass**

**def** zip\_longest(\*args, \*\*kwds):

*# zip\_longest('ABCD', 'xy', fillvalue='-') --> Ax By C- D-*

fillvalue = kwds.get('fillvalue')

counter = len(args) - 1

**def** sentinel():

**nonlocal** counter

**if** **not** counter:

**raise** ZipExhausted

counter -= 1

**yield** fillvalue

fillers = repeat(fillvalue)

iterators = [chain(it, sentinel(), fillers) **for** it **in** args]

**try**:

**while** iterators:

**yield** tuple(map(next, iterators))

**except** ZipExhausted:

**pass**

If one of the iterables is potentially infinite, then the zip\_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.

10.1.2. Itertools 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 [generator](https://docs.python.org/3/glossary.html#term-generator)s which incur interpreter overhead.

**def** take(n, iterable):

"Return first n items of the iterable as a list"

**return** list(islice(iterable, n))

**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):

"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** 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(tuple(iterable), n))

**def** dotproduct(vec1, vec2):

**return** sum(map(operator.mul, vec1, vec2))

**def** flatten(listOfLists):

"Flatten one level of nesting"

**return** 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** zip(a, b)

**def** grouper(iterable, n, fillvalue=**None**):

"Collect data into fixed-length chunks or blocks"

*# grouper('ABCDEFG', 3, 'x') --> ABC DEF Gxx"*

args = [iter(iterable)] \* n

**return** zip\_longest(\*args, fillvalue=fillvalue)

**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** 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** 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(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(random.choice(pool) **for** pool **in** 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.randrange(n) **for** i **in** range(r))

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

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, map=map, mul=operator.mul):

**return** sum(map(mul, vec1, vec2))