`itertools` — Functions creating iterators for efficient looping

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), ...`. The same effect can be achieved in Python by combining `map()` and `count()` to form `map(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(starmap(operator.mul, zip(vec1, vec2, strict=True)))`.

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	[start[, step]]	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:

Iterator	Arguments	Results	Example
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, ...	`accumulate([1,2,3,4,5]) → 1 3 6 10 15`
`batched()`	p, n	(p0, p1, ..., p_n-1), ...	`batched('ABCDEFG', n=3) → ABC DEF G`
`chain()`	p, q, ...	p0, p1, ... plast, q0, q1, ...	`chain('ABC', 'DEF') → A B C D E F`
`chain.from_iterable()`	iterable	p0, p1, ... plast, q0, q1, ...	`chain.from_iterable(['ABC', 'DEF']) → A B C D E F`
`compress()`	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	`compress('ABCDEF', [1,0,1,0,1,1]) → A C E F`

dropwhile()	predicate, seq	seq[n], seq[n+1], starting when predicate fails	dropwhile(lambda x: x<5, [1,4,6,3,8]) → 6 3 8
filterfalse()	predicate, seq	elements of seq where predicate(elem) fails	filterfalse(lambda x: x<5, [1,4,6,3,8]) → 6 8
groupby()	iterable[, key]	sub-iterators grouped by value of key(v)	
islice()	seq, [start,] stop [, step]	elements from seq[start:stop:step]	islice('ABCDEFG', 2, None) → C D E F G
pairwise()	iterable	(p[0], p[1]), (p[1], p[2])	pairwise('ABCDEFG') → AB BC CD DE EF FG
starmap()	func, seq	func(*seq[0]), func(*seq[1]), ...	starmap(pow, [(2,5), (3,2), (10,3)]) → 32 9 1000
takewhile()	predicate, seq	seq[0], seq[1], until predicate fails	takewhile(lambda x: x<5, [1,4,6,3,8]) → 1 4
tee()	it, n	it1, it2, ... itn splits one iterator into n	
zip_longest()	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

Combinatoric iterators:

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
combinations_with_replacement()	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', 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.

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

Make an iterator that returns accumulated sums or accumulated results from other binary functions.

The *function* defaults to addition. The *function* should accept two arguments, an accumulated total and a value from the *iterable*.

If an *initial* value is provided, the accumulation will start with that value and the output will have one more element than the input iterable.

Roughly equivalent to:

```
def accumulate(iterable, function=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

    iterator = iter(iterable)
    total = initial
    if initial is None:
        try:
            total = next(iterator)
        except StopIteration:
            return

    yield total
    for element in iterator:
        total = function(total, element)
        yield total
```

The *function* argument can be set to `min()` for a running minimum, `max()` for a running maximum, or `operator.mul()` for a running product. Amortization tables can be built by accumulating interest and applying payments:

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

# Amortize a 5% loan of 1000 with 10 annual payments of 90
>>> update = lambda balance, payment: round(balance * 1.05) - payment
>>> list(accumulate(repeat(90, 10), update, initial=1_000))
[1000, 960, 918, 874, 828, 779, 728, 674, 618, 559, 497]
```

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

Changed in version 3.3: Added the optional *function* parameter.

Changed in version 3.8: Added the optional *initial* parameter.

itertools.**batched**(*iterable*, *n*)

Batch data from the *iterable* into tuples of length *n*. The last batch may be shorter than *n*.

Loops over the input iterable and accumulates data into tuples up to size *n*. The input is consumed lazily, just enough to fill a batch. The result is yielded as soon as the batch is full or when the input iterable is exhausted:

```
>>> flattened_data = ['roses', 'red', 'violets', 'blue', 'sugar', 'sweet']
>>> unflattened = list(batched(flattened_data, 2))
>>> unflattened
[('roses', 'red'), ('violets', 'blue'), ('sugar', 'sweet')]
```

Roughly equivalent to:

```
def batched(iterable, n):
    # batched('ABCDEFG', 3) → ABC DEF G
    if n < 1:
        raise ValueError('n must be at least one')
    iterator = iter(iterable)
    while batch := tuple(islice(iterator, n)):
        yield batch
```

Added in version 3.12.

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 iterable in iterables:
        yield from iterable
```

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 iterable in iterables:
        yield from iterable
```

itertools.**combinations**(*iterable*, *r*)

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

The output is a subsequence of product() keeping only entries that are subsequences of the *iterable*. The length of the output is given by math.comb() which computes `n! / r! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within 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)
```

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.

The output is a subsequence of product() that keeps only entries that are subsequences (with possible repeated elements) of the *iterable*. The number of subsequence returned is `(n + r - 1)! / r! / (n - 1)!` when `n > 0`.

The combination tuples are emitted in lexicographic order according to the order of the input *iterable*. if the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. 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)):
```

```
        else:
            return
        indices[i:] = [indices[i] + 1] * (r - i)
        yield tuple(pool[i] for i in indices)
```

Added in version 3.1.

`itertools.`**`compress`**`(`*`data, selectors`*`)`

Make an iterator that returns elements from *data* where the corresponding element in *selectors* is true. Stops when either the *data* or *selectors* iterables have been exhausted. Roughly equivalent to:

```
def compress(data, selectors):
    # compress('ABCDEF', [1,0,1,0,1,1]) → A C E F
    return (datum for datum, selector in zip(data, selectors) if selector)
```

Added in version 3.1.

`itertools.`**`count`**`(`*`start=0, step=1`*`)`

Make an iterator that returns evenly spaced values beginning with *start*. Can be used with `map()` to generate consecutive data points or with `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
```

This itertool 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* while the *predicate* is true and afterwards returns every element. Roughly equivalent to:

```
    iterator = iter(iterable)
    for x in iterator:
        if not predicate(x):
            yield x
            break

    for x in iterator:
        yield x
```

Note this does not produce *any* output until the predicate first becomes false, so this itertool may have a lengthy start-up time.

itertools.**filterfalse**(*predicate*, *iterable*)

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

```
def filterfalse(predicate, iterable):
    # filterfalse(lambda x: x<5, [1,4,6,3,8]) → 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:

```
def groupby(iterable, key=None):
    # [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

    keyfunc = (lambda x: x) if key is None else key
    iterator = iter(iterable)
```

```
    def _grouper(target_key):
        nonlocal curr_value, curr_key, exhausted
        yield curr_value
        for curr_value in iterator:
            curr_key = keyfunc(curr_value)
            if curr_key != target_key:
                return
            yield curr_value
        exhausted = True

    try:
        curr_value = next(iterator)
    except StopIteration:
        return
    curr_key = keyfunc(curr_value)

    while not exhausted:
        target_key = curr_key
        curr_group = _grouper(target_key)
        yield curr_key, curr_group
        if curr_key == target_key:
            for _ in curr_group:
                pass
```

itertools.**islice**(*iterable*, *stop*)
itertools.**islice**(*iterable*, *start*, *stop*[, *step*])

Make an iterator that returns selected elements from the iterable. Works like sequence slicing but does not support negative values for *start*, *stop*, or *step*.

If *start* is zero or `None`, iteration starts at zero. Otherwise, elements from the iterable are skipped until *start* is reached.

If *stop* is `None`, iteration continues until the iterator is exhausted, if at all. Otherwise, it stops at the specified position.

If *step* is `None`, the step defaults to one. Elements are returned consecutively unless *step* is set higher than one which results in items being skipped.

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 = 0 if s.start is None else s.start
    stop = s.stop
    step = 1 if s.step is None else s.step
    if start < 0 or (stop is not None and stop < 0) or step <= 0:
        raise ValueError

    indices = count() if stop is None else range(max(start, stop))
    next_i = start
    for i, element in zip(indices, iterable):
        if i == next_i:
```

itertools.**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
    iterator = iter(iterable)
    a = next(iterator, None)
    for b in iterator:
        yield a, b
        a = b
```

> *Added in version 3.10.*

itertools.**permutations**(*iterable*, *r=None*)

Return successive *r* length permutations of elements from 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 output is a subsequence of `product()` where entries with repeated elements have been filtered out. The length of the output is given by `math.perm()` which computes `n! / (n - r)!` when `0 ≤ r ≤ n` or zero when `r > n`.

The permutation tuples are emitted in lexicographic order according to the order of the input *iterable*. If the input *iterable* is sorted, the output tuples will be produced in sorted order.

Elements are treated as unique based on their position, not on their value. If the input elements are unique, there will be no repeated values within a 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:
```

```
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

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(*iterables, 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 iterables] * 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 `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.

itertools.**repeat**(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

Roughly equivalent to:

```
def repeat(object, times=None):
    # repeat(10, 3) → 10 10 10
    if times is None:
        while True:
            yield object
    else:
```

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()` when argument parameters have already been "pre-zipped" into tuples.

The difference between `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, [1,4,6,3,8]) → 1 4
    for x in iterable:
        if not predicate(x):
            break
        yield x
```

Note, the element that first fails the predicate condition is consumed from the input iterator and there is no way to access it. This could be an issue if an application wants to further consume the input iterator after *takewhile* has been run to exhaustion. To work around this problem, consider using more-iterools before_and_after() instead.

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

Return *n* independent iterators from a single iterable.

Roughly equivalent to:

```
def tee(iterable, n=2):
    iterator = iter(iterable)
    shared_link = [None, None]
    return tuple(_tee(iterator, shared_link) for _ in range(n))

def _tee(iterator, link):
    try:
        while True:
            if link[1] is None:
                link[0] = next(iterator)
                link[1] = [None, None]
            value, link = link
            yield value
    except StopIteration:
        return
```

tee iterators are not threadsafe. A RuntimeError may be raised when simultaneously using iterators returned by the same 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 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*. If not specified, *fillvalue* defaults to None.

Iteration continues until the longest iterable is exhausted.

Roughly equivalent to:

```
def zip_longest(*iterables, fillvalue=None):
    # zip_longest('ABCD', 'xy', fillvalue='-') → Ax By C- D-

    iterators = list(map(iter, iterables))
    num_active = len(iterators)
    if not num_active:
        return

    while True:
        values = []
        for i, iterator in enumerate(iterators):
            try:
                value = next(iterator)
            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 zip_longest() function should be wrapped with something that limits the number of calls (for example islice() or takewhile()).

Itertools Recipes

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

The primary purpose of the itertools recipes is educational. The recipes show various ways of thinking about individual tools — for example, that chain.from_iterable is related to the concept of flattening. The recipes also give ideas about ways that the tools can be combined — for example, how starmap() and repeat() can work together. The recipes also show patterns for using itertools with the operator and collections modules as well as with the built-in itertools such as map(), filter(), reversed(), and enumerate().

recipes are being tested to see whether they prove their worth.

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

```
python -m pip install more-itertools
```

Many of the recipes offer the same high performance as the underlying toolset. 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. High speed is retained by preferring "vectorized" building blocks over the use of for-loops and generators which incur interpreter overhead.

```
import collections
import contextlib
import functools
import math
import operator
import random

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

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

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

def repeatfunc(func, times=None, *args):
    "Repeat calls to func with specified arguments."
    if times is None:
        return starmap(func, repeat(args))
    return starmap(func, repeat(args, times))

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

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

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:
        collections.deque(iterator, maxlen=0)
    else:
        next(islice(iterator, n, n), None)
```

```
    "Returns the nth item or a default value."
    return next(islice(iterable, n, None), default)

def quantify(iterable, predicate=bool):
    "Given a predicate that returns True or False, count the True results."
    return sum(map(predicate, iterable))

def first_true(iterable, default=False, predicate=None):
    "Returns the first true value or the *default* if there is no true value."
    # 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(predicate, iterable), default)

def all_equal(iterable, key=None):
    "Returns True if all the elements are equal to each other."
    # all_equal('4٤௪౪໔', key=int) → True
    return len(take(2, groupby(iterable, key))) <= 1

def unique_justseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember only the element just seen."
    # unique_justseen('AAAABBBCCDAABBB') → A B C D A B
    # unique_justseen('ABBcCAD', str.casefold) → A B c A D
    if key is None:
        return map(operator.itemgetter(0), groupby(iterable))
    return map(next, map(operator.itemgetter(1), groupby(iterable, key)))

def unique_everseen(iterable, key=None):
    "Yield unique elements, preserving order. Remember all elements ever seen."
    # unique_everseen('AAAABBBCCDAABBB') → A B C D
    # unique_everseen('ABBcCAD', str.casefold) → A B c D
    seen = set()
    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(iterable, key=None, reverse=False):
   "Yield unique elements in sorted order. Supports unhashable inputs."
   # unique([[1, 2], [3, 4], [1, 2]]) → [1, 2] [3, 4]
   return unique_justseen(sorted(iterable, key=key, reverse=reverse), key=key)

def sliding_window(iterable, n):
    "Collect data into overlapping fixed-length chunks or blocks."
    # sliding_window('ABCDEFG', 4) → ABCD BCDE CDEF DEFG
    iterator = iter(iterable)
    window = collections.deque(islice(iterator, n - 1), maxlen=n)
    for x in iterator:
        window.append(x)
        yield tuple(window)

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
    iterators = [iter(iterable)] * n
```

```
        return zip_longest(*iterators, fillvalue=fillvalue)
    case 'strict':
        return zip(*iterators, strict=True)
    case 'ignore':
        return zip(*iterators)
    case _:
        raise ValueError('Expected fill, strict, or ignore')

def roundrobin(*iterables):
    "Visit input iterables in a cycle until each is exhausted."
    # roundrobin('ABC', 'D', 'EF') → A D E B F C
    # Algorithm credited to George Sakkis
    iterators = map(iter, iterables)
    for num_active in range(len(iterables), 0, -1):
        iterators = cycle(islice(iterators, num_active))
        yield from map(next, iterators)

def partition(predicate, iterable):
    """Partition entries into false entries and true entries.

    If *predicate* is slow, consider wrapping it with functools.lru_cache().
    """
    # partition(is_odd, range(10)) → 0 2 4 6 8   and  1 3 5 7 9
    t1, t2 = tee(iterable)
    return filterfalse(predicate, t1), filter(predicate, t2)

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 iter_index(iterable, value, start=0, stop=None):
    "Return indices where a value occurs in a sequence or iterable."
    # iter_index('AABCADEAF', 'A') → 0 1 4 7
    seq_index = getattr(iterable, 'index', None)
    if seq_index is None:
        iterator = islice(iterable, start, stop)
        for i, element in enumerate(iterator, start):
            if element is value or element == value:
                yield i
    else:
        stop = len(iterable) if stop is None else stop
        i = start
        with contextlib.suppress(ValueError):
            while True:
                yield (i := seq_index(value, i, stop))
                i += 1

def iter_except(func, exception, first=None):
    "Convert a call-until-exception interface to an iterator interface."
    # iter_except(d.popitem, KeyError) → non-blocking dictionary iterator
    with contextlib.suppress(exception):
        if first is not None:
            yield first()
        while True:
            yield func()
```

The following recipes have a more mathematical flavor:

```
    s = list(iterable)
    return chain.from_iterable(combinations(s, r) for r in range(len(s)+1))

def sum_of_squares(iterable):
    "Add up the squares of the input values."
    # sum_of_squares([10, 20, 30]) → 1400
    return math.sumprod(*tee(iterable))

def reshape(matrix, cols):
    "Reshape a 2-D matrix to have a given number of columns."
    # reshape([(0, 1), (2, 3), (4, 5)], 3) →  (0, 1, 2), (3, 4, 5)
    return batched(chain.from_iterable(matrix), cols)

def transpose(matrix):
    "Swap the rows and columns of a 2-D matrix."
    # transpose([(1, 2, 3), (11, 22, 33)]) → (1, 11) (2, 22) (3, 33)
    return zip(*matrix, strict=True)

def matmul(m1, m2):
    "Multiply two matrices."
    # matmul([(7, 5), (3, 5)], [(2, 5), (7, 9)]) → (49, 80), (41, 60)
    n = len(m2[0])
    return batched(starmap(math.sumprod, product(m1, transpose(m2))), n)

def convolve(signal, kernel):
    """Discrete linear convolution of two iterables.
    Equivalent to polynomial multiplication.

    Convolutions are mathematically commutative; however, the inputs are
    evaluated differently.  The signal is consumed lazily and can be
    infinite. The kernel is fully consumed before the calculations begin.

    Article:  https://betterexplained.com/articles/intuitive-convolution/
    Video:    https://www.youtube.com/watch?v=KuXjwB4LzSA
    """
    # convolve([1, -1, -20], [1, -3]) → 1 -4 -17 60
    # convolve(data, [0.25, 0.25, 0.25, 0.25]) → Moving average (blur)
    # convolve(data, [1/2, 0, -1/2]) → 1st derivative estimate
    # convolve(data, [1, -2, 1]) → 2nd derivative estimate
    kernel = tuple(kernel)[::-1]
    n = len(kernel)
    padded_signal = chain(repeat(0, n-1), signal, repeat(0, n-1))
    windowed_signal = sliding_window(padded_signal, n)
    return map(math.sumprod, repeat(kernel), windowed_signal)

def polynomial_from_roots(roots):
    """Compute a polynomial's coefficients from its roots.

       (x - 5) (x + 4) (x - 3)  expands to:   x³ -4x² -17x + 60
    """
    # polynomial_from_roots([5, -4, 3]) → [1, -4, -17, 60]
    factors = zip(repeat(1), map(operator.neg, roots))
    return list(functools.reduce(convolve, factors, [1]))

def polynomial_eval(coefficients, x):
    """Evaluate a polynomial at a specific value.

    Computes with better numeric stability than Horner's method.
    """
    # Evaluate x³ -4x² -17x + 60 at x = 5
    # polynomial_eval([1, -4, -17, 60], x=5) → 0
    n = len(coefficients)
```

```
    powers = map(pow, repeat(x), reversed(range(n)))
    return math.sumprod(coefficients, powers)

def polynomial_derivative(coefficients):
    """Compute the first derivative of a polynomial.

       f(x)  =  x³ -4x² -17x + 60
       f'(x) = 3x² -8x  -17
    """
    # polynomial_derivative([1, -4, -17, 60]) → [3, -8, -17]
    n = len(coefficients)
    powers = reversed(range(1, n))
    return list(map(operator.mul, coefficients, powers))

def sieve(n):
    "Primes less than n."
    # sieve(30) → 2 3 5 7 11 13 17 19 23 29
    if n > 2:
        yield 2
    data = bytearray((0, 1)) * (n // 2)
    for p in iter_index(data, 1, start=3, stop=math.isqrt(n) + 1):
        data[p*p : n : p+p] = bytes(len(range(p*p, n, p+p)))
    yield from iter_index(data, 1, start=3)

def factor(n):
    "Prime factors of n."
    # factor(99) → 3 3 11
    # factor(1_000_000_000_000_007) → 47 59 360620266859
    # factor(1_000_000_000_000_403) → 1000000000000403
    for prime in sieve(math.isqrt(n) + 1):
        while not n % prime:
            yield prime
            n //= prime
            if n == 1:
                return
    if n > 1:
        yield n

def totient(n):
    "Count of natural numbers up to n that are coprime to n."
    # https://mathworld.wolfram.com/TotientFunction.html
    # totient(12) → 4 because len([1, 5, 7, 11]) == 4
    for prime in set(factor(n)):
        n -= n // prime
    return n
```