

# Itertools in Python 3, By Example

by David Amos   May 30, 2018   35 Comments

**advanced** **python**

Mark as Completed  

Tweet   Share   Email

## Table of Contents

- [What Is Itertools and Why Should You Use It?](#)
- [The grouper Recipe](#)
- [Et tu, Brute Force?](#)
  - [Section Recap](#)
- [Sequences of Numbers](#)
  - [Evens and Odds](#)
  - [Recurrence Relations](#)
  - [Section Recap](#)
- [Dealing a Deck of Cards](#)
  - [Section Recap](#)
- [Intermission: Flattening A List of Lists](#)
- [Analyzing the S&P500](#)
  - [Maximum Gain and Loss](#)
  - [Longest Growth Streak](#)
  - [Section Recap](#)
- [Building Relay Teams From Swimmer Data](#)
- [Where to Go From Here](#)



**Master Real-World Python Skills  
With a Community of Experts**

**Level Up With Unlimited Access** to Our Vast Library  
of Python Tutorials and Video Lessons

**Watch Now »**

[Remove ads](#)

It has been called a “gem” and “[pretty much the coolest thing ever](#),” and if you have not heard of it, then you are missing out on one of the greatest corners of the Python 3 standard library: `itertools`.

[Help](#)

A handful of excellent resources exist for learning what functions are available in the `itertools` module. The [docs](#) themselves are a great place to start. So is [this post](#).

The thing about `itertools`, though, is that it is not enough to just know the definitions of the functions it contains. The real power lies in composing these functions to create fast, memory-efficient, and good-looking code.

This article takes a different approach. Rather than introducing `itertools` to you one function at a time, you will construct practical examples designed to encourage you to “think iteratively.” In general, the examples will start simple and gradually increase in complexity.

A word of warning: this article is long and intended for the intermediate-to-advanced Python programmer. Before diving in, you should be confident using iterators and generators in Python 3, multiple assignment, and tuple unpacking. If you aren’t, or if you need to brush up on your knowledge, consider checking out the following before reading on:

- [Python Iterators: A Step-By-Step Introduction](#)
- [Introduction to Python Generators](#)
- Chapter 6 of [Python Tricks: The Book](#) by Dan Bader
- [Multiple assignment and tuple unpacking improve Python code readability](#)

**Free Bonus:** [Click here to get our itertools cheat sheet](#) that summarizes the techniques demonstrated in this tutorial.

All set? Let’s start the way any good journey should—with a question.

## What Is `itertools` and Why Should You Use It?

According to the [itertools docs](#), it is a “module [that] implements a number of iterator building blocks inspired by constructs from APL, Haskell, and SML... Together, they form an ‘iterator algebra’ making it possible to construct specialized tools succinctly and efficiently in pure Python.”

Loosely speaking, this means that the functions in `itertools` “operate” on iterators to produce more complex iterators. Consider, for example, the [built-in `zip\(\)` function](#), which takes any number of [iterables](#) as arguments and returns an [iterator](#) over tuples of their corresponding elements:

```
Python >>>
>>> list(zip([1, 2, 3], ['a', 'b', 'c']))
[(1, 'a'), (2, 'b'), (3, 'c')]
```

How, exactly, does `zip()` work?

`[1, 2, 3]` and `['a', 'b', 'c']`, like all lists, are iterable, which means they can return their elements one at a time. Technically, any Python object that implements the `__iter__()` or `__getitem__()` methods is iterable. (See the [Python 3 docs glossary](#) for a more detailed explanation.)

The [iter\(\) built-in function](#), when called on an iterable, returns an [iterator object](#) for that iterable:

```
Python >>>
>>> iter([1, 2, 3, 4])
<list_iterator object at 0x7fa80af0d898>
```

Under the hood, the [zip\(\) function](#) works, in essence, by calling `iter()` on each of its arguments, then advancing each iterator returned by `iter()` with `next()` and aggregating the results into tuples. The iterator returned by [zip\(\)](#) iterates over these tuples.

The [map\(\) built-in function](#) is another “iterator operator” that, in its simplest form, applies a single-parameter function to each element of an iterable one element at a time:

Python

&gt;&gt;&gt;

```
>>> list(map(len, ['abc', 'de', 'fghi']))  
[3, 2, 4]
```

The `map()` function works by calling `iter()` on its second argument, advancing this iterator with `next()` until the iterator is exhausted, and applying the function passed to its first argument to the value returned by `next()` at each step. In the above example, `len()` is called on each element of `['abc', 'de', 'fghi']` to return an iterator over the lengths of each string in the list.

Since [iterators are iterable](#), you can compose `zip()` and `map()` to produce an iterator over combinations of elements in more than one iterable. For example, the following sums corresponding elements of two lists:

Python

&gt;&gt;&gt;

```
>>> list(map(sum, zip([1, 2, 3], [4, 5, 6])))  
[5, 7, 9]
```

This is what is meant by the functions in `itertools` forming an “iterator algebra.” `itertools` is best viewed as a collection of building blocks that can be combined to form specialized “data pipelines” like the one in the example above.

**Historical Note:** In Python 2, the built-in `zip()` and `map()` functions do not return an iterator, but rather a list. To return an iterator, the `izip()` and `imap()` functions of `itertools` must be used. In Python 3, `izip()` and `imap()` have been [removed from itertools](#) and replaced the `zip()` and `map()` built-ins. So, in a way, if you have ever used `zip()` or `map()` in Python 3, you have already been using `itertools`!

There are two main reasons why such an “iterator algebra” is useful: improved memory efficiency (via [lazy evaluation](#)) and faster execution time. To see this, consider the following problem:

Given a list of values `inputs` and a positive integer `n`, write a function that splits `inputs` into groups of length `n`. For simplicity, assume that the length of the input list is divisible by `n`. For example, if `inputs = [1, 2, 3, 4, 5, 6]` and `n = 2`, your function should return `[(1, 2), (3, 4), (5, 6)]`.

Taking a naive approach, you might write something like this:

Python

```
def naive_grouper(inputs, n):  
    num_groups = len(inputs) // n  
    return [tuple(inputs[i*n:(i+1)*n]) for i in range(num_groups)]
```

When you test it, you see that it works as expected:

Python

&gt;&gt;&gt;

```
>>> nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
>>> naive_grouper(nums, 2)  
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

What happens when you try to pass it a list with, say, 100 million elements? You will need a whole lot of available memory! Even if you have enough memory available, your program will hang for a while until the output list is populated.

To see this, store the following in a script called `naive.py`:

Python

```
def naive_grouper(inputs, n):
    num_groups = len(inputs) // n
    return [tuple(inputs[i*n:(i+1)*n]) for i in range(num_groups)]

for _ in naive_grouper(range(100000000), 10):
    pass
```

From the console, you can use the `time` command (on UNIX systems) to measure memory usage and CPU user time.

**Make sure you have at least 5GB of free memory before executing the following:**

Shell

```
$ time -f "Memory used (kB): %M\nUser time (seconds): %U" python3 naive.py
Memory used (kB): 4551872
User time (seconds): 11.04
```

**Note:** On Ubuntu, you may need to run `/usr/bin/time` instead of `time` for the above example to work.

The `list` and `tuple` implementation in `naive_grouper()` requires approximately 4.5GB of memory to process `range(100000000)`. Working with iterators drastically improves this situation. Consider the following:

Python

```
def better_grouper(inputs, n):
    iters = [iter(inputs)] * n
    return zip(*iters)
```

There’s a lot going on in this little function, so let’s break it down with a concrete example. The expression `[iters(inputs)] * n` creates a list of `n` references to the same iterator:

Python

&gt;&gt;&gt;

```
>>> nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> iters = [iter(nums)] * 2
>>> list(id(itr) for itr in iters) # IDs are the same.
[139949748267160, 139949748267160]
```

Next, `zip(*iters)` returns an iterator over pairs of corresponding elements of each iterator in `iters`. When the first element, 1, is taken from the “first” iterator, the “second” iterator now starts at 2 since it is just a reference to the “first” iterator and has therefore been advanced one step. So, the first tuple produced by `zip()` is (1, 2).

At this point, “both” iterators in `iters` start at 3, so when `zip()` pulls 3 from the “first” iterator, it gets 4 from the “second” to produce the tuple (3, 4). This process continues until `zip()` finally produces (9, 10) and “both” iterators in `iters` are exhausted:

Python

&gt;&gt;&gt;

```
>>> nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(better_grouper(nums, 2))
[(1, 2), (3, 4), (5, 6), (7, 8), (9, 10)]
```

The `better_grouper()` function is better for a couple of reasons. First, without the reference to the `len()` built-in, `better_grouper()` can take any iterable as an argument (even infinite iterators). Second, by returning an iterator rather than a list, `better_grouper()` can process enormous iterables without trouble and uses much less memory.

Store the following in a file called `better.py` and run it with `time` from the console again:

Python

```
def better_grouper(inputs, n):
    iters = [iter(inputs)] * n
    return zip(*iters)

for _ in better_grouper(range(100000000), 10):
    pass
```

Shell

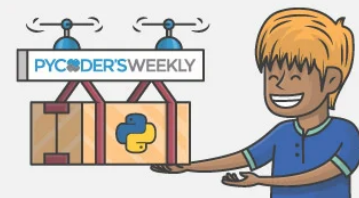
```
$ time -f "Memory used (kB): %M\nUser time (seconds): %U" python3 better.py
Memory used (kB): 7224
User time (seconds): 2.48
```

That’s a whopping 630 times less memory used than `naive.py` in less than a quarter of the time!

Now that you’ve seen what `itertools` is (“iterator algebra”) and why you should use it (improved memory efficiency and faster execution time), let’s take a look at how to take `better_grouper()` to the next level with `itertools`.

## Your Weekly Dose of All Things Python!

pycoders.com



[Remove ads](#)

## The grouper Recipe

The problem with `better_grouper()` is that it doesn’t handle situations where the value passed to the second argument isn’t a factor of the length of the iterable in the first argument:

Python

&gt;&gt;&gt;

```
>>> nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(better_grouper(nums, 4))
[(1, 2, 3, 4), (5, 6, 7, 8)]
```

The elements 9 and 10 are missing from the grouped output. This happens because `zip()` stops aggregating elements once the shortest iterable passed to it is exhausted. It would make more sense to return a third group containing 9 and 10.

To do this, you can use `itertools.zip_longest()`. This function accepts any number of iterables as arguments and a `fillvalue` keyword argument that defaults to `None`. The easiest way to get a sense of the difference between `zip()` and `zip_longest()` is to look at some example output:

Python

&gt;&gt;&gt;

```
>>> import itertools as it
>>> x = [1, 2, 3, 4, 5]
>>> y = ['a', 'b', 'c']
>>> list(zip(x, y))
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> list(it.zip_longest(x, y))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, None), (5, None)]
```

With this in mind, replace `zip()` in `better_grouper()` with `zip_longest()`:

Python

```
import itertools as it

def grouper(inputs, n, fillvalue=None):
    iters = [iter(inputs)] * n
    return it.zip_longest(*iters, fillvalue=fillvalue)
```



Now you get a better result:

Python

&gt;&gt;&gt;

```
>>> nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> print(list(grouper(nums, 4)))
[(1, 2, 3, 4), (5, 6, 7, 8), (9, 10, None, None)]
```

The `grouper()` function can be found in the [Recipes section](#) of the `itertools` docs. The recipes are an excellent source of inspiration for ways to use `itertools` to your advantage.

**Note:** From this point forward, the line `import itertools` as it will not be included at the beginning of examples. All `itertools` methods in code examples are prefaced with `it.` The module import is implied.

If you get a `NameError: name 'itertools' is not defined` or a `NameError: name 'it' is not defined` exception when running one of the examples in this tutorial you'll need to import the `itertools` module first.

## Et tu, Brute Force?

Here's a common interview-style problem:

You have three \$20 dollar bills, five \$10 dollar bills, two \$5 dollar bills, and five \$1 dollar bills. How many ways can you make change for a \$100 dollar bill?

To “brute force” this problem, you just start listing off the ways there are to choose one bill from your wallet, check whether any of these makes change for \$100, then list the ways to pick two bills from your wallet, check again, and so on and so forth.

But you are a programmer, so naturally you want to automate this process.

First, create a list of the bills you have in your wallet:

Python

```
bills = [20, 20, 20, 10, 10, 10, 10, 10, 5, 5, 1, 1, 1, 1, 1]
```

A choice of  $k$  things from a set of  $n$  things is called a [combination](#), and `itertools` has your back here. The `itertools.combinations()` function takes two arguments—an iterable `inputs` and a positive integer `n`—and produces an iterator over tuples of all combinations of `n` elements in `inputs`.

For example, to list the combinations of three bills in your wallet, just do:

Python

&gt;&gt;&gt;

```
>>> list(it.combinations(bills, 3))
[(20, 20, 20), (20, 20, 10), (20, 20, 10), ... ]
```

To solve the problem, you can loop over the positive integers from 1 to `len(bills)`, then check which combinations of each size add up to \$100:

Python

&gt;&gt;&gt;

```
>>> makes_100 = []
>>> for n in range(1, len(bills) + 1):
...     for combination in it.combinations(bills, n):
...         if sum(combination) == 100:
...             makes_100.append(combination)
```

If you print out `makes_100`, you will notice there are a lot of repeated combinations. This makes sense because you can make change for \$100 with three \$20 dollar bills and four \$10 bills, but `combinations()` does this with the first four \$10 dollars bills in your wallet; the first, third, fourth and fifth \$10 dollar bills; the first, second, fourth and fifth \$10 bills; and so on.

To remove duplicates from `makes_100`, you can convert it to a set:

Python

&gt;&gt;&gt;

```
>>> set(makes_100)
{(20, 20, 10, 10, 10, 10, 10, 5, 1, 1, 1, 1, 1),
 (20, 20, 10, 10, 10, 10, 10, 5, 5),
 (20, 20, 20, 10, 10, 10, 5, 1, 1, 1, 1, 1),
 (20, 20, 20, 10, 10, 10, 5, 5),
 (20, 20, 20, 10, 10, 10, 10)}
```

So, there are five ways to make change for a \$100 bill with the bills you have in your wallet.

Here's a variation on the same problem:

How many ways are there to make change for a \$100 bill using any number of \$50, \$20, \$10, \$5, and \$1 dollar bills?

In this case, you don't have a pre-set collection of bills, so you need a way to generate all possible combinations using any number of bills. For this, you'll need the `itertools.combinations_with_replacement()` function.

It works just like `combinations()`, accepting an iterable `inputs` and a positive integer `n`, and returns an iterator over `n`-tuples of elements from `inputs`. The difference is that `combinations_with_replacement()` allows elements to be repeated in the tuples it returns.

For example:

Python

&gt;&gt;&gt;

```
>>> list(it.combinations_with_replacement([1, 2], 2))
[(1, 1), (1, 2), (2, 2)]
```

Compare that to `combinations()`:

Python

&gt;&gt;&gt;

```
>>> list(it.combinations([1, 2], 2))
[(1, 2)]
```

Here's what the solution to the revised problem looks like:

Python

&gt;&gt;&gt;

```
>>> bills = [50, 20, 10, 5, 1]
>>> make_100 = []
>>> for n in range(1, 101):
...     for combination in it.combinations_with_replacement(bills, n):
...         if sum(combination) == 100:
...             makes_100.append(combination)
```

In this case, you do not need to remove any duplicates since `combinations_with_replacement()` won't produce any:

Python

&gt;&gt;&gt;

```
>>> len(makes_100)
343
```

If you run the above solution, you may notice that it takes a while for the output to display. That is because it has to process 96,560,645 combinations!

Another "brute force" `itertools` function is `permutations()`, which accepts a single iterable and produces all possible permutations (rearrangements) of its elements:

Python

&gt;&gt;&gt;

```
>>> list(it.permutations(['a', 'b', 'c']))
[('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'),
 ('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')]
```

Any iterable of three elements will have six permutations, and the number of permutations of longer iterables grows extremely fast. In fact, an iterable of length  $n$  has  $n!$  permutations, where

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$$

To put this in perspective, here’s a table of these [numbers](#) for  $n = 1$  to  $n = 10$ :


$n$	$n!$
2	2
3	6
4	24
5	120
6	720
7	5,040
8	40,320
9	362,880
10	3,628,800


The phenomenon of just a few inputs producing a large number of outcomes is called a [combinatorial explosion](#) and is something to keep in mind when working with `combinations()`, `combinations_with_replacement()`, and `permutations()`.

It is usually best to avoid brute force algorithms, although there are times you may need to use one (for example, if the correctness of the algorithm is critical, or every possible outcome *must* be considered). In that case, `itertools` has you covered.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You

pythonistacafe.com



 [Remove ads](#)

## Section Recap

In this section you met three `itertools` functions: `combinations()`, `combinations_with_replacement()`, and `permutations()`.

Let’s review these functions before moving on:

### `itertools.combinations` Example

`combinations(iterable, n)`

Return successive n-length combinations of elements in the iterable.

Python

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

>>>

### `itertools.combinations_with_replacement` Example

https://realpython.com/python-itertools/

8/35



```
combinations_with_replacement(iterable, n)
```

Return successive n-length combinations of elements in the iterable allowing individual elements to have successive repeats.

Python

&gt;&gt;&gt;

```
>>> combinations_with_replacement([1, 2], 2)
(1, 1), (1, 2), (2, 2)
```

## itertools.permutations Example

```
permutations(iterable, n=None)
```

Return successive n-length permutations of elements in the iterable.

Python

&gt;&gt;&gt;

```
>>> permutations('abc')
('a', 'b', 'c'), ('a', 'c', 'b'), ('b', 'a', 'c'),
('b', 'c', 'a'), ('c', 'a', 'b'), ('c', 'b', 'a')
```

# Sequences of Numbers

With `itertools`, you can easily generate iterators over infinite sequences. In this section, you will explore numeric sequences, but the tools and techniques seen here are by no means limited to numbers.

## Evens and Odds

For the first example, you will create a pair of iterators over even and odd integers *without explicitly doing any arithmetic*. Before diving in, let's look at an arithmetic solution using [generators](#):

Python

&gt;&gt;&gt;

```
>>> def evens():
...     """Generate even integers, starting with 0."""
...     n = 0
...     while True:
...         yield n
...         n += 2
...
>>> evens = evens()
>>> list(next(evens) for _ in range(5))
[0, 2, 4, 6, 8]

>>> def odds():
...     """Generate odd integers, starting with 1."""
...     n = 1
...     while True:
...         yield n
...         n += 2
...
>>> odds = odds()
>>> list(next(odds) for _ in range(5))
[1, 3, 5, 7, 9]
```

That is pretty straightforward, but with `itertools` you can do this much more compactly. The function you need is `itertools.count()`, which does exactly what it sounds like: it counts, starting by default with the number 0.

Python

&gt;&gt;&gt;

```
>>> counter = it.count()
>>> list(next(counter) for _ in range(5))
[0, 1, 2, 3, 4]
```

You can start counting from any number you like by setting the `start` keyword argument, which defaults to 0. You can even set a `step` keyword argument to determine the interval between numbers returned from `count()`—this defaults to 1.

With `count()`, iterators over even and odd integers become literal one-liners:

Python

&gt;&gt;&gt;

```
>>> evens = it.count(step=2)
>>> list(next(evens) for _ in range(5))
[0, 2, 4, 6, 8]

>>> odds = it.count(start=1, step=2)
>>> list(next(odds) for _ in range(5))
[1, 3, 5, 7, 9]
```

Ever since [Python 3.1](#), the `count()` function also accepts non-integer arguments:

Python

&gt;&gt;&gt;

```
>>> count_with_floats = it.count(start=0.5, step=0.75)
>>> list(next(count_with_floats) for _ in range(5))
[0.5, 1.25, 2.0, 2.75, 3.5]
```

You can even pass it negative numbers:

Python

&gt;&gt;&gt;

```
>>> negative_count = it.count(start=-1, step=-0.5)
>>> list(next(negative_count) for _ in range(5))
[-1, -1.5, -2.0, -2.5, -3.0]
```

In some ways, `count()` is similar to the built-in `range()` function, but `count()` always returns an infinite sequence. You might wonder what good an infinite sequence is since it's impossible to iterate over completely. That is a valid question, and I admit the first time I was introduced to infinite iterators, I too didn't quite see the point.

The example that made me realize the power of the infinite iterator was the following, which emulates the behavior of the [built-in `enumerate\(\)` function](#):

Python

&gt;&gt;&gt;

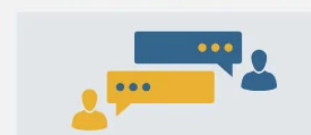
```
>>> list(zip(it.count(), ['a', 'b', 'c']))
[(0, 'a'), (1, 'b'), (2, 'c')]
```


It is a simple example, but think about it: you just enumerated a list without a `for` loop and without knowing the length of the list ahead of time.

**A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You**

pythonistacafe.com

 PYTHONISTACAFE



 [Remove ads](#)

## Recurrence Relations

A [recurrence relation](#) is a way of describing a sequence of numbers with a recursive formula. One of the best-known recurrence relations is the one that describes the [Fibonacci sequence](#).

The Fibonacci sequence is the sequence 0, 1, 1, 2, 3, 5, 8, 13, ... It starts with 0 and 1, and each subsequent number in the sequence is the sum of the previous two. The numbers in this sequence are called the Fibonacci numbers. In mathematical notation, the recurrence relation describing the  $n$ -th Fibonacci number looks like this:

$$F_n = F_{n-1} + F_{n-2}; \quad F_0 = 0, F_1 = 1$$

**Note:** If you search Google, you will find a host of implementations of these numbers in Python. You can find a recursive function that produces them in the [Thinking Recursively in Python](#) article here on Real Python.

It is common to see the Fibonacci sequence produced with a generator:

Python

```
def fibs():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
```

The recurrence relation describing the Fibonacci numbers is called a *second order recurrence relation* because, to calculate the next number in the sequence, you need to look back two numbers behind it.

In general, second order recurrence relations have the form:

$$s_n = Ps_{n-1} + Qs_{n-2} + R$$

Here,  $P$ ,  $Q$ , and  $R$  are constants. To generate the sequence, you need two initial values. For the Fibonacci numbers,  $P = Q = 1$ ,  $R = 0$ , and the initial values are 0 and 1.

As you might guess, a *first order recurrence relation* has the following form:

$$s_n = Ps_{n-1} + Q$$

There are countless sequences of numbers that can be described by first and second order recurrence relations. For example, the positive integers can be described as a first order recurrence relation with  $P = Q = 1$  and initial value 1. For the even integers, take  $P = 1$  and  $Q = 2$  with initial value 0.

In this section, you will construct functions for producing *any* sequence whose values can be described with a first or second order recurrence relation.

## First Order Recurrence Relations

You've already seen how `count()` can generate the sequence of non-negative integers, the even integers, and the odd integers. You can also use it to generate the sequence  $3n = 0, 3, 6, 9, 12, \dots$  and  $4n = 0, 4, 8, 12, 16, \dots$

Python

```
count_by_three = it.count(step=3)
count_by_four = it.count(step=4)
```

In fact, `count()` can produce sequences of multiples of any number you wish. These sequences can be described with first-order recurrence relations. For example, to generate the sequence of multiples of some number  $n$ , just take  $P = 1$ ,  $Q = n$ , and initial value 0.

Another easy example of a first-order recurrence relation is the constant sequence  $n, n, n, n, n, \dots$ , where  $n$  is any value you'd like. For this sequence, set  $P = 1$  and  $Q = 0$  with initial value  $n$ . `itertools` provides an easy way to implement this sequence as well, with the `repeat()` function:

Python

```
all_ones = it.repeat(1) # 1, 1, 1, 1, ...
all_twos = it.repeat(2) # 2, 2, 2, 2, ...
```

If you need a finite sequence of repeated values, you can set a stopping point by passing a positive integer as a second argument:

Python

```
five_ones = it.repeat(1, 5) # 1, 1, 1, 1, 1
three_fours = it.repeat(4, 3) # 4, 4, 4
```

What may not be quite as obvious is that the sequence 1, -1, 1, -1, 1, -1, ... of alternating 1s and -1s can also be described by a first order recurrence relation. Just take  $P = -1$ ,  $Q = 0$ , and initial value 1.

There's an easy way to generate this sequence with the `itertools.cycle()` function. This function takes an iterable `inputs` as an argument and returns an infinite iterator over the values in `inputs` that returns to the beginning once the end of `inputs` is reached. So, to produce the alternating sequence of 1s and -1s, you could do this:

Python

```
alternating_ones = it.cycle([1, -1]) # 1, -1, 1, -1, 1, -1, ...
```

The goal of this section, though, is to produce a single function that can generate *any* first order recurrence relation—just pass it  $P$ ,  $Q$ , and an initial value. One way to do this is with `itertools.accumulate()`.

The `accumulate()` function takes two arguments—an iterable `inputs` and a **binary function** `func` (that is, a function with exactly two inputs)—and returns an iterator over accumulated results of applying `func` to elements of `inputs`. It is roughly equivalent to the following generator:

Python

```
def accumulate(inputs, func):
    itr = iter(inputs)
    prev = next(itr)
    for cur in itr:
        yield prev
        prev = func(prev, cur)
```

For example:

Python

&gt;&gt;&gt;

```
>>> import operator
>>> list(it.accumulate([1, 2, 3, 4, 5], operator.add))
[1, 3, 6, 10, 15]
```

The first value in the iterator returned by `accumulate()` is always the first value in the input sequence. In the above example, this is 1—the first value in `[1, 2, 3, 4, 5]`.

The next value in the output iterator is the sum of the first two elements of the input sequence: `add(1, 2) = 3`. To produce the next value, `accumulate()` takes the result of `add(1, 2)` and adds this to the third value in the input sequence:

```
add(3, 3) = add(add(1, 2), 3) = 6
```

The fourth value produced by `accumulate()` is `add(add(add(1, 2), 3), 4) = 10`, and so on.

The second argument of `accumulate()` defaults to `operator.add()`, so the previous example can be simplified to:

Python

&gt;&gt;&gt;

```
>>> list(it.accumulate([1, 2, 3, 4, 5]))
[1, 3, 6, 10, 15]
```

Passing the built-in `min()` to `accumulate()` will keep track of a running minimum:

Python

&gt;&gt;&gt;

```
>>> list(it.accumulate([9, 21, 17, 5, 11, 12, 2, 6], min))
[9, 9, 9, 5, 5, 5, 2, 2]
```

More complex functions can be passed to `accumulate()` with lambda expressions:

Python

&gt;&gt;&gt;

```
>>> list(it.accumulate([1, 2, 3, 4, 5], lambda x, y: (x + y) / 2))
[1, 1.5, 2.25, 3.125, 4.0625]
```

The order of the arguments in the binary function passed to `accumulate()` is important. The first argument is always the previously accumulated result and the second argument is always the next element of the input iterable. For example, consider the difference in output of the following expressions:

Python

&gt;&gt;&gt;

```
>>> list(it.accumulate([1, 2, 3, 4, 5], lambda x, y: x - y))
[1, -1, -4, -8, -13]

>>> list(it.accumulate([1, 2, 3, 4, 5], lambda x, y: y - x))
[1, 1, 2, 2, 3]
```

To model a recurrence relation, you can just ignore the second argument of the binary function passed to `accumulate()`. That is, given values  $p$ ,  $q$ , and  $s$ , `lambda x, _: p*s + q` will return the value following  $x$  in the recurrence relation defined by  $s_i = Ps_{i-1} + Q$ .

In order for `accumulate()` to iterate over the resulting recurrence relation, you need to pass to it an infinite sequence with the right initial value. It doesn't matter what the rest of the values in the sequence are, as long as the initial value is the initial value of the recurrence relation. You can do this with `repeat()`:

Python

```
def first_order(p, q, initial_val):
    """Return sequence defined by  $s(n) = p * s(n-1) + q$ ."""
    return it.accumulate(it.repeat(initial_val), lambda s, _: p*s + q)
```

Using `first_order()`, you can build the sequences from above as follows:

Python

```
>>> evens = first_order(p=1, q=2, initial_val=0)
>>> list(next(evens) for _ in range(5))
[0, 2, 4, 6, 8]

>>> odds = first_order(p=1, q=2, initial_val=1)
>>> list(next(odds) for _ in range(5))
[1, 3, 5, 7, 9]

>>> count_by_threes = first_order(p=1, q=3, initial_val=0)
>>> list(next(count_by_threes) for _ in range(5))
[0, 3, 6, 9, 12]

>>> count_by_fours = first_order(p=1, q=4, initial_val=0)
>>> list(next(count_by_fours) for _ in range(5))
[0, 4, 8, 12, 16]

>>> all_ones = first_order(p=1, q=0, initial_val=1)
>>> list(next(all_ones) for _ in range(5))
[1, 1, 1, 1, 1]

>>> all_twos = first_order(p=1, q=0, initial_val=2)
>>> list(next(all_twos) for _ in range(5))
[2, 2, 2, 2, 2]

>>> alternating_ones = first_order(p=-1, q=0, initial_val=1)
>>> list(next(alternating_ones) for _ in range(5))
[1, -1, 1, -1, 1]
```

## Second Order Recurrence Relations

Generating sequences described by second order recurrence relations, like the Fibonacci sequence, can be accomplished using a similar technique as the one used for first order recurrence relations.

The difference here is that you need to create an intermediate sequence of tuples that keep track of the previous two elements of the sequence, and then `map()` each of these tuples to their first component to get the final sequence.

Here's what it looks like:

Python

```
def second_order(p, q, r, initial_values):
    """Return sequence defined by  $s(n) = p * s(n-1) + q * s(n-2) + r$ ."""
    intermediate = it.accumulate(
        it.repeat(initial_values),
        lambda s, _: (s[1], p*s[1] + q*s[0] + r)
    )
    return map(lambda x: x[0], intermediate)
```

Using `second_order()`, you can generate the Fibonacci sequence like this:

Python

```
>>> fibs = second_order(p=1, q=1, r=0, initial_values=(0, 1))
>>> list(next(fibs) for _ in range(8))
[0, 1, 1, 2, 3, 5, 8, 13]
```

Other sequences can be easily generated by changing the values of `p`, `q`, and `r`. For example, the [Pell numbers](#) and the [Lucas numbers](#) can be generated as follows:

Python

```
pell = second_order(p=2, q=1, r=0, initial_values=(0, 1))
>>> list(next(pell) for _ in range(6))
[0, 1, 2, 5, 12, 29]

>>> lucas = second_order(p=1, q=1, r=0, initial_values=(2, 1))
>>> list(next(lucas) for _ in range(6))
[2, 1, 3, 4, 7, 11]
```

You can even generate the alternating Fibonacci numbers:

Python

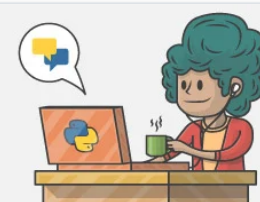
```
>>> alt_fibs = second_order(p=-1, q=1, r=0, initial_values=(-1, 1))
>>> list(next(alt_fibs) for _ in range(6))
[-1, 1, -2, 3, -5, 8]
```


This is all really cool if you are a giant math nerd like I am, but step back for a second and compare `second_order()` to the `fibs()` generator from the beginning of this section. Which one is easier to understand?

This is a valuable lesson. The `accumulate()` function is a powerful tool to have in your toolkit, but there are times when using it could mean sacrificing clarity and readability.

**A Peer-to-Peer Learning Community for  
Python Enthusiasts...Just Like You**

pythonistacafe.com



 [Remove ads](#)

## Section Recap

You saw several `itertools` function in this section. Let's review those now.

### `itertools.count` Example

```
count(start=0, step=1)
```

Return a count object whose `__next__()` method returns consecutive values.



Python

&gt;&gt;&gt;

```
>>> count()
0, 1, 2, 3, 4, ...

>>> count(start=1, step=2)
1, 3, 5, 7, 9, ...
```

## itertools.repeat Example

```
repeat(object, times=1)
```

Create an iterator which returns the object for the specified number of times. If not specified, returns the object endlessly.

Python

&gt;&gt;&gt;

```
>>> repeat(2)
2, 2, 2, 2, 2 ...

>>> repeat(2, 5) # Stops after 5 repetitions.
2, 2, 2, 2, 2
```

## itertools.cycle Example

```
cycle(iterable)
```

Return elements from the iterable until it is exhausted. Then repeat the sequence indefinitely.

Python

&gt;&gt;&gt;

```
>>> cycle(['a', 'b', 'c'])
a, b, c, a, b, c, a, ...
```

## itertools.accumulate Example

```
accumulate(iterable, func=operator.add)
```

Return series of accumulated sums (or other binary function results).

Python

&gt;&gt;&gt;

```
>>> accumulate([1, 2, 3])
1, 3, 6
```

Alright, let's take a break from the math and have some fun with cards.

# Dealing a Deck of Cards

Suppose you are building a Poker app. You'll need a deck of cards. You might start by defining a list of ranks (ace, king, queen, jack, 10, 9, and so on) and a list of suits (hearts, diamonds, clubs, and spades):

Python

```
ranks = ['A', 'K', 'Q', 'J', '10', '9', '8', '7', '6', '5', '4', '3', '2']
suits = ['H', 'D', 'C', 'S']
```

You could represent a card as a tuple whose first element is a rank and second element is a suit. A deck of cards would be a collection of such tuples. The deck should act like the real thing, so it makes sense to define a generator that yields cards one at a time and becomes exhausted once all the cards are dealt.

One way to achieve this is to write a generator with a nested for loop over ranks and suits:

Python

```
def cards():  
    """Return a generator that yields playing cards."""  
    for rank in ranks:  
        for suit in suits:  
            yield rank, suit
```

You could write this more compactly with a generator expression:

Python

```
cards = ((rank, suit) for rank in ranks for suit in suits)
```

However, some might argue that this is actually more difficult to understand than the more explicit nested for loop.

It helps to view nested for loops from a mathematical standpoint—that is, as a [Cartesian product](#) of two or more iterables. In mathematics, the Cartesian product of two sets  $A$  and  $B$  is the set of all tuples of the form  $(a, b)$  where  $a$  is an element of  $A$  and  $b$  is an element of  $B$ .

Here’s an example with Python iterables: the Cartesian product of  $A = [1, 2]$  and  $B = ['a', 'b']$  is  $[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]$ .

The `itertools.product()` function is for exactly this situation. It takes any number of iterables as arguments and returns an iterator over tuples in the Cartesian product:

Python

```
it.product([1, 2], ['a', 'b']) # (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')
```

The `product()` function is by no means limited to two iterables. You can pass it as many as you like—they don’t even have to all be of the same size! See if you can predict what `product([1, 2, 3], ['a', 'b'], ['c'])` is, then check your work by running it in the interpreter.

**Warning:** The `product()` function is another “brute force” function and can lead to a combinatorial explosion if you aren’t careful.

Using `product()`, you can re-write the `cards` in a single line:

Python

```
cards = it.product(ranks, suits)
```

This is all fine and dandy, but any Poker app worth its salt better start with a shuffled deck:

Python

```
import random  
  
def shuffle(deck):  
    """Return iterator over shuffled deck."""  
    deck = list(deck)  
    random.shuffle(deck)  
    return iter(tuple(deck))  
  
cards = shuffle(cards)
```

**Note:** The `random.shuffle()` function uses the [Fisher-Yates shuffle](#) to shuffle a list (or any mutable sequence) in place in  $O(n)$  time. This algorithm is well-suited for shuffling cards because it produces an unbiased permutation—that is, all permutations of the iterable are equally likely to be returned by `random.shuffle()`.

That said, you probably noticed that `shuffle()` creates a copy of its input deck in memory by calling `list(deck)`. While this seemingly goes against the spirit of this article, this author is unaware of a good way to shuffle an iterator without making a copy.

As a courtesy to your users, you would like to give them the opportunity to cut the deck. If you imagine the cards being stacked neatly on a table, you have the user pick a number  $n$  and then remove the first  $n$  cards from the top of the stack and move them to the bottom.

If you know a thing or two about [slicing](#), you might accomplish this like so:

Python

```
def cut(deck, n):
    """Return an iterator over a deck of cards cut at index `n`."""
    if n < 0:
        raise ValueError("`n` must be a non-negative integer')

    deck = list(deck)
    return iter(deck[n:] + deck[:n])

cards = cut(cards, 26) # Cut the deck in half.
```

The `cut()` function first converts `deck` to a list so that you can slice it to make the cut. To guarantee your slices behave as expected, you’ve got to check that  $n$  is non-negative. If it isn’t, you better throw an exception so that nothing crazy happens.

Cutting the deck is pretty straightforward: the top of the cut deck is just `deck[:n]`, and the bottom is the remaining cards, or `deck[n:]`. To construct the new deck with the top “half” moved to the bottom, you just append it to the bottom: `deck[n:] + deck[:n]`.

The `cut()` function is pretty simple, but it suffers from a couple of problems. When you slice a list, you make a copy of the original list and return a new list with the selected elements. With a deck of only 52 cards, this increase in space complexity is trivial, but you could reduce the memory overhead using `itertools`. To do this, you’ll need three functions: `itertools.tee()`, `itertools.islice()`, and `itertools.chain()`.

Let’s take a look at how those functions work.

The `tee()` function can be used to create any number of independent iterators from a single iterable. It takes two arguments: the first is an iterable `inputs`, and the second is the number  $n$  of independent iterators over `inputs` to return (by default,  $n$  is set to 2). The iterators are returned in a tuple of length  $n$ .

Python

>>>

```
>>> iterator1, iterator2 = it.tee([1, 2, 3, 4, 5], 2)
>>> list(iterator1)
[1, 2, 3, 4, 5]
>>> list(iterator1) # iterator1 is now exhausted.
[]
>>> list(iterator2) # iterator2 works independently of iterator1
[1, 2, 3, 4, 5].
```

While `tee()` is useful for creating independent iterators, it is important to understand a little bit about how it works under the hood. When you call `tee()` to create  $n$  independent iterators, each iterator is essentially working with its own FIFO queue.

When a value is extracted from one iterator, that value is appended to the queues for the other iterators. Thus, if one iterator is exhausted before the others, each remaining iterator will hold a copy of the entire iterable in memory. (You can find a Python function that emulates `tee()` in the `itertools` [docs](#).)

For this reason, `tee()` should be used with care. If you are exhausting large portions of an iterator before working with the other returned by `tee()`, you may be better off casting the input iterator to a list or tuple.

The `islice()` function works much the same way as slicing a list or tuple. You pass it an iterable, a starting, and stopping point, and, just like slicing a list, the slice returned stops at the index just before the stopping point. You can optionally include a step value, as well. The biggest difference here is, of course, that `islice()` returns an iterator.

Python

&gt;&gt;&gt;

```
>>> # Slice from index 2 to 4
>>> list(it.islice('ABCDEFG', 2, 5))
['C' 'D' 'E']

>>> # Slice from beginning to index 4, in steps of 2
>>> list(it.islice([1, 2, 3, 4, 5], 0, 5, 2))
[1, 3, 5]

>>> # Slice from index 3 to the end
>>> list(it.islice(range(10), 3, None))
[3, 4, 5, 6, 7, 8, 9]

>>> # Slice from beginning to index 3
>>> list(it.islice('ABCDE', 4))
['A', 'B', 'C', 'D']
```

The last two examples above are useful for truncating iterables. You can use this to replace the list slicing used in `cut()` to select the “top” and “bottom” of the deck. As an added bonus, `islice()` won’t accept negative indices for the start/stop positions and the step value, so you won’t need to raise an exception if `n` is negative.

The last function you need is `chain()`. This function takes any number of iterables as arguments and “chains” them together. For example:

Python

&gt;&gt;&gt;

```
>>> list(it.chain('ABC', 'DEF'))
['A' 'B' 'C' 'D' 'E' 'F']

>>> list(it.chain([1, 2], [3, 4, 5, 6], [7, 8, 9]))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Now that you’ve got some additional firepower in your arsenal, you can re-write the `cut()` function to cut the deck of cards without making a full copy cards in memory:

Python

```
def cut(deck, n):
    """Return an iterator over a deck of cards cut at index `n`."""
    deck1, deck2 = it.tee(deck, 2)
    top = it.islice(deck1, n)
    bottom = it.islice(deck2, n, None)
    return it.chain(bottom, top)

cards = cut(cards, 26)
```

Now that you have shuffled and cut the cards, it is time to deal some hands. You could write a function `deal()` that takes a deck, the number of hands, and the hand size as arguments and returns a tuple containing the specified number of hands.

You do not need any new `itertools` functions to write this function. See what you can come up with on your own before reading ahead.

Here’s one solution:

Python

```
def deal(deck, num_hands=1, hand_size=5):
    iters = [iter(deck)] * hand_size
    return tuple(zip(*(tuple(it.islice(itr, num_hands)) for itr in iters)))
```

You start by creating a list of `hand_size` references to an iterator over `deck`. You then iterate over this list, removing `num_hands` cards at each step and storing them in tuples.

Next, you `zip()` these tuples up to emulate dealing one card at a time to each player. This produces `num_hands` tuples, each containing `hand_size` cards. Finally, you package the hands up into a tuple to return them all at once.

This implementation sets the default values for `num_hands` to 1 and `hand_size` to 5—maybe you are making a “Five Card Draw” app. Here’s how you would use this function, with some sample output:

Python

&gt;&gt;&gt;

```
>>> p1_hand, p2_hand, p3_hand = deal(cards, num_hands=3)
>>> p1_hand
(('A', 'S'), ('5', 'S'), ('7', 'H'), ('9', 'H'), ('5', 'H'))
>>> p2_hand
(('10', 'H'), ('2', 'D'), ('2', 'S'), ('J', 'C'), ('9', 'C'))
>>> p3_hand
(('2', 'C'), ('Q', 'S'), ('6', 'C'), ('Q', 'H'), ('A', 'C'))
```

What do you think the state of `cards` is now that you have dealt three hands of five cards?

Python


&gt;&gt;&gt;

```
>>> len(tuple(cards))
37
```

The fifteen cards dealt are consumed from the `cards` iterator, which is exactly what you want. That way, as the game continues, the state of the `cards` iterator reflects the state of the deck in play.

**A Peer-to-Peer Learning Community for  
Python Enthusiasts...Just Like You**  
pythonistacafe.com



 [Remove ads](#)

## Section Recap

Let’s review the `itertools` functions you saw in this section.

### `itertools.product` Example

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

Cartesian product of input iterables. Equivalent to nested for loops.

Python

&gt;&gt;&gt;

```
>>> product([1, 2], ['a', 'b'])
(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')
```

### `itertools.tee` Example

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

Create any number of independent iterators from a single input iterable.

Python

&gt;&gt;&gt;

```
>>> iter1, iter2 = it.tee(['a', 'b', 'c'], 2)
>>> list(iter1)
['a', 'b', 'c']
>>> list(iter2)
['a', 'b', 'c']
```

### `itertools.islice` Example

```
islice(iterable, stop) islice(iterable, start, stop, step=1)
```

Return an iterator whose `__next__()` method returns selected values from an iterable. Works like a `slice()` on a list but returns an iterator.

Python

&gt;&gt;&gt;

```
>>> islice([1, 2, 3, 4], 3)
1, 2, 3

>>> islice([1, 2, 3, 4], 1, 2)
2, 3
```

## itertools.chain Example

```
chain(*iterables)
```

Return a chain object whose `__next__()` method returns elements from the first iterable until it is exhausted, then elements from the next iterable, until all of the iterables are exhausted.

Python

&gt;&gt;&gt;

```
>>> chain('abc', [1, 2, 3])
'a', 'b', 'c', 1, 2, 3
```

## Intermission: Flattening A List of Lists

In the previous example, you used `chain()` to tack one iterator onto the end of another. The `chain()` function has a class method `.from_iterable()` that takes a single iterable as an argument. The elements of the iterable must themselves be iterable, so the net effect is that `chain.from_iterable()` flattens its argument:

Python

&gt;&gt;&gt;

```
>>> list(it.chain.from_iterable([[1, 2, 3], [4, 5, 6]]))
[1, 2, 3, 4, 5, 6]
```

There’s no reason the argument of `chain.from_iterable()` needs to be finite. You could emulate the behavior of `cycle()`, for example:

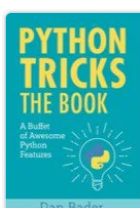
Python

```
>>> cycle = it.chain.from_iterable(it.repeat('abc'))
>>> list(it.islice(cycle, 8))
['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b']
```

The `chain.from_iterable()` function is useful when you need to build an iterator over data that has been “chunked.”


In the next section, you will see how to use `itertools` to do some data analysis on a large dataset. But you deserve a break for having stuck with it this far. Why not hydrate yourself and relax a bit? Maybe even play a little [Star Trek: The Nth Iteration](#).

Back? Great! Let’s do some data analysis.



**“I wished I had access to a book like this when I started learning Python many years ago”**  
— Mariatta Wijaya, CPython Core Developer

[Learn More »](#)

 [Remove ads](#)



# Analyzing the S&P500

In this example, you will get your first taste of using `itertools` to manipulate a large dataset—in particular, the historical daily price data of the S&P500 index. A CSV file `SP500.csv` with this data can be found [here](#) (source: [Yahoo Finance](#)). The problem you’ll tackle is this:

Determine the maximum daily gain, daily loss (in percent change), and the longest growth streak in the history of the S&P500.

To get a feel for what you’re dealing with, here are the first ten rows of `SP500.csv`:

## Shell

```
$ head -n 10 SP500.csv
Date,Open,High,Low,Close,Adj Close,Volume
1950-01-03,16.660000,16.660000,16.660000,16.660000,16.660000,1260000
1950-01-04,16.850000,16.850000,16.850000,16.850000,16.850000,1890000
1950-01-05,16.930000,16.930000,16.930000,16.930000,16.930000,2550000
1950-01-06,16.980000,16.980000,16.980000,16.980000,16.980000,2010000
1950-01-09,17.080000,17.080000,17.080000,17.080000,17.080000,2520000
1950-01-10,17.030001,17.030001,17.030001,17.030001,17.030001,2160000
1950-01-11,17.090000,17.090000,17.090000,17.090000,17.090000,2630000
1950-01-12,16.760000,16.760000,16.760000,16.760000,16.760000,2970000
1950-01-13,16.670000,16.670000,16.670000,16.670000,16.670000,3330000
```

As you can see, the early data is limited. The data improves for later dates, and, as a whole, is sufficient for this example.

The strategy for solving this problem is as follows:

- Read data from the CSV file and transform it into a sequence gains of daily percent changes using the “Adj Close” column.
- Find the maximum and minimum values of the gains sequence, and the date on which they occur. (Note that it is possible that these values are attained on more than one date; in that case, the most recent date will suffice.)
- Transform gains into a sequence `growth_streaks` of tuples of consecutive positive values in gains. Then determine the length of the longest tuple in `growth_streaks` and the beginning and ending dates of the streak. (It is possible that the maximum length is attained by more than one tuple in `growth_streaks`; in that case, the tuple with the most recent beginning and ending dates will suffice.)

The *percent change* between two values  $x$  and  $y$  is given by the following formula:

$$\frac{x - y}{y} \times 100 = \left( \frac{x}{y} - 1 \right) \times 100$$

For each step in the analysis, it is necessary to compare values associated with dates. To facilitate these comparisons, you can subclass the [namedtuple object](#) from the [collections module](#):

## Python

```
from collections import namedtuple

class DataPoint(namedtuple('DataPoint', ['date', 'value'])):
    __slots__ = ()

    def __le__(self, other):
        return self.value <= other.value

    def __lt__(self, other):
        return self.value < other.value

    def __gt__(self, other):
        return self.value > other.value
```

The `DataPoint` class has two attributes: `date` (a `datetime.datetime` instance) and `value`. The `__le__()`, `__lt__()`, and `__gt__()` [dunder methods](#) are implemented so that the `<=`, `<`, and `>` boolean comparators can be used to compare the values of two `DataPoint` objects. This also allows the `max()` and `min()` built-in functions to be called with `DataPoint` arguments.

**Note:** If you are not familiar with `namedtuple`, check out [this excellent resource](#). The `namedtuple` implementation for `DataPoint` is just one of many ways to build this data structure. For example, in Python 3.7 you could implement `DataPoint` as a data class. Check out our [Ultimate Guide to Data Classes](#) for more information.

The following reads the data from `SP500.csv` to a tuple of `DataPoint` objects:

Python

```
import csv
from datetime import datetime

def read_prices(csvfile, _strptime=datetime.strptime):
    with open(csvfile) as infile:
        reader = csv.DictReader(infile)
        for row in reader:
            yield DataPoint(date=_strptime(row['Date'], '%Y-%m-%d').date(),
                           value=float(row['Adj Close']))

prices = tuple(read_prices('SP500.csv'))
```

The `read_prices()` generator opens `SP500.csv` and reads each row with a `csv.DictReader()` object. `DictReader()` returns each row as an `OrderedDict` whose keys are the column names from the header row of the CSV file.

For each row, `read_prices()` yields a `DataPoint` object containing the values in the “Date” and “Adj Close” columns. Finally, the full sequence of data points is committed to memory as a tuple and stored in the `prices` [variable](#).

Next, `prices` needs to be transformed to a sequence of daily percent changes:

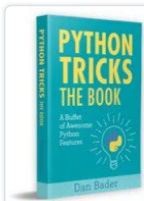
Python

```
gains = tuple(DataPoint(day.date, 100*(day.value/prev_day.value - 1.))
               for day, prev_day in zip(prices[1:], prices))
```

The choice of storing the data in a tuple is intentional. Although you could point `gains` to an iterator, you will need to iterate over the data twice to find the minimum and maximum values.

If you use `tee()` to create two independent iterators, exhausting one iterator to find the maximum will create a copy of all of the data in memory for the second iterator. By creating a tuple up front, you do not lose anything in terms of space complexity compared to `tee()`, and you may even gain a little speed.

**Note:** This example focuses on leveraging `itertools` for analyzing the S&P500 data. Those intent on working with a lot of time series financial data might also want to check out the [Pandas](#) library, which is well suited for such tasks.



“I don’t even feel like I’ve scratched the surface of what I can do with Python”

Write More Pythonic Code »

[Remove ads](#)

## Maximum Gain and Loss

To determine the maximum gain on any single day, you might do something like this:

Python

```
max_gain = DataPoint(None, 0)
for data_point in gains:
    max_gain = max(data_point, max_gain)

print(max_gain)    # DataPoint(date='2008-10-28', value=11.58)
```

You can simplify the for loop using the [functools.reduce\(\)](#) function. This function accepts a binary function func and an iterable inputs as arguments, and “reduces” inputs to a single value by applying func cumulatively to pairs of objects in the iterable.

For example, `functools.reduce(operator.add, [1, 2, 3, 4, 5])` will return the sum  $1 + 2 + 3 + 4 + 5 = 15$ . You can think of [reduce\(\)](#) as working in much the same way as `accumulate()`, except that it returns only the final value in the new sequence.

Using `reduce()`, you can get rid of the for loop altogether in the above example:

Python

```
import functools as ft

max_gain = ft.reduce(max, gains)

print(max_gain)    # DataPoint(date='2008-10-28', value=11.58)
```

The above solution works, but it isn’t equivalent to the for loop you had before. Do you see why? Suppose the data in your CSV file recorded a loss every single day. What would the value of `max_gain` be?

In the for loop, you first set `max_gain = DataPoint(None, 0)`, so if there are no gains, the final `max_gain` value will be this empty `DataPoint` object. However, the `reduce()` solution returns the smallest loss. That is not what you want and could introduce a difficult to find bug.

This is where `itertools` can help you out. The `itertools.filterfalse()` function takes two arguments: a function that returns `True` or `False` (called a **predicate**), and an iterable inputs. It returns an iterator over the elements in inputs for which the predicate returns `False`.

Here’s a simple example:

Python

```
>>> only_positives = it.filterfalse(lambda x: x <= 0, [0, 1, -1, 2, -2])
>>> list(only_positives)
[1, 2]
```

You can use `filterfalse()` to filter out the values in `gains` that are negative or zero so that `reduce()` only works on positive values:

Python

```
max_gain = ft.reduce(max, it.filterfalse(lambda p: p <= 0, gains))
```

What happens if there are never any gains? Consider the following:

Python Traceback

```
>>> ft.reduce(max, it.filterfalse(lambda x: x <= 0, [-1, -2, -3]))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
```

Well, that’s not what you want! But, it makes sense because the iterator returned by `filterfalse()` is empty. You could handle the `TypeError` by wrapping the call to `reduce()` with `try...except`, but there’s a better way.

The `reduce()` function accepts an optional third argument for an initial value. Passing `0` to this third argument gets you the expected behavior:

Python

&gt;&gt;&gt;

```
>>> ft.reduce(max, it.filterfalse(lambda x: x <= 0, [-1, -2, -3]), 0)
0
```

Applying this to the S&P500 example:

Python

```
zdp = DataPoint(None, 0) # zero DataPoint
max_gain = ft.reduce(max, it.filterfalse(lambda p: p <= 0, diffs), zdp)
```

Great! You've got it working just the way it should! Now, finding the maximum loss is easy:

Python

```
max_loss = ft.reduce(min, it.filterfalse(lambda p: p > 0, gains), zdp)

print(max_loss) # DataPoint(date='2018-02-08', value=-20.47)
```

## Python Tricks The Book

A Buffet of Awesome Python Features

Get Your Free Sample Chapter



 [Remove ads](#)

## Longest Growth Streak

Finding the longest growth streak in the history of the S&P500 is equivalent to finding the largest number of consecutive positive data points in the gains sequence. The `itertools.takewhile()` and `itertools.dropwhile()` functions are perfect for this situation.

The `takewhile()` function takes a predicate and an iterable inputs as arguments and returns an iterator over inputs that stops at the first instance of an element for which the predicate returns `False`:

Python

```
it.takewhile(lambda x: x < 3, [0, 1, 2, 3, 4]) # 0, 1, 2
```

The `dropwhile()` function does exactly the opposite. It returns an iterator beginning at the first element for which the predicate returns `False`:

Python

```
it.dropwhile(lambda x: x < 3, [0, 1, 2, 3, 4]) # 3, 4
```

In the following generator function, `takewhile()` and `dropwhile()` are composed to yield tuples of consecutive positive elements of a sequence:

Python

```
def consecutive_positives(sequence, zero=0):
    def _consecutives():
        for itr in it.repeat(iter(sequence)):
            yield tuple(it.takewhile(lambda p: p > zero,
                                   it.dropwhile(lambda p: p <= zero, itr)))
    return it.takewhile(lambda t: len(t), _consecutives())
```

The `consecutive_positives()` function works because `repeat()` keeps returning a [pointer](#) to an iterator over the sequence argument, which is being partially consumed at each iteration by the call to `tuple()` in the `yield` statement.

You can use `consecutive_positives()` to get a generator that produces tuples of consecutive positive data points in gains:

Python

```
growth_streaks = consecutive_positives(gains, zero=DataPoint(None, 0))
```

Now you can use `reduce()` to extract the longest growth streak:

Python

```
longest_streak = ft.reduce(lambda x, y: x if len(x) > len(y) else y,  
                           growth_streaks)
```

Putting the whole thing together, here's a full script that will read data from the `SP500.csv` file and print out the max gain/loss and longest growth streak:

Python

```

from collections import namedtuple
import csv
from datetime import datetime
import itertools as it
import functools as ft

class DataPoint(namedtuple('DataPoint', ['date', 'value'])):
    __slots__ = ()

    def __le__(self, other):
        return self.value <= other.value

    def __lt__(self, other):
        return self.value < other.value

    def __gt__(self, other):
        return self.value > other.value

def consecutive_positives(sequence, zero=0):
    def _consecutives():
        for itr in it.repeat(iter(sequence)):
            yield tuple(it.takewhile(lambda p: p > zero,
                                   it.dropwhile(lambda p: p <= zero, itr)))
    return it.takewhile(lambda t: len(t), _consecutives())

def read_prices(csvfile, _strptime=datetime.strptime):
    with open(csvfile) as infile:
        reader = csv.DictReader(infile)
        for row in reader:
            yield DataPoint(date=_strptime(row['Date'], '%Y-%m-%d').date(),
                            value=float(row['Adj Close']))

# Read prices and calculate daily percent change.
prices = tuple(read_prices('SP500.csv'))
gains = tuple(DataPoint(day.date, 100*(day.value/prev_day.value - 1.))
               for day, prev_day in zip(prices[1:], prices))

# Find maximum daily gain/loss.
zdp = DataPoint(None, 0) # zero DataPoint
max_gain = ft.reduce(max, it.filterfalse(lambda p: p <= zdp, gains))
max_loss = ft.reduce(min, it.filterfalse(lambda p: p > zdp, gains), zdp)

# Find longest growth streak.
growth_streaks = consecutive_positives(gains, zero=DataPoint(None, 0))
longest_streak = ft.reduce(lambda x, y: x if len(x) > len(y) else y,
                           growth_streaks)

# Display results.
print('Max gain: {1:.2f}% on {0}'.format(*max_gain))
print('Max loss: {1:.2f}% on {0}'.format(*max_loss))

print('Longest growth streak: {num_days} days ({first} to {last})'.format(
    num_days=len(longest_streak),
    first=longest_streak[0].date,
    last=longest_streak[-1].date
))

```

Running the above script produces the following output:

Shell

```

Max gain: 11.58% on 2008-10-13
Max loss: -20.47% on 1987-10-19
Longest growth streak: 14 days (1971-03-26 to 1971-04-15)

```



## Section Recap

In this section, you covered a lot of ground, but you only saw a few functions from `itertools`. Let's review those now.

### `itertools.filterfalse` Example

```
filterfalse(pred, iterable)
```

Return those items of sequence for which `pred(item)` is false. If `pred` is `None`, return the items that are false.

Python

>>>

```
>>> filterfalse(bool, [1, 0, 1, 0, 0])
0, 0, 0
```

### `itertools.takewhile` Example

```
takewhile(pred, iterable)
```

Return successive entries from an iterable as long as `pred` evaluates to true for each entry.

Python

>>>

```
>>> takewhile(bool, [1, 1, 1, 0, 0])
1, 1, 1
```

### `itertools.dropwhile` Example

```
dropwhile(pred, iterable)
```

Drop items from the iterable while `pred(item)` is true. Afterwards, return every element until the iterable is exhausted.

Python

>>>

```
>>> dropwhile(bool, [1, 1, 1, 0, 0, 1, 1, 0])
0, 0, 1, 1, 0
```

You are really starting to master this whole `itertools` thing! The community swim team would like to commission you for a small project.

**Learn Python Programming, By Example**

realpython.com



 [Remove ads](#)

## Building Relay Teams From Swimmer Data

In this example, you will read data from a CSV file containing swimming event times for a community swim team from all of the swim meets over the course of a season. The goal is to determine which swimmers should be in the relay teams for each stroke next season.

Each stroke should have an “A” and a “B” relay team with four swimmers each. The “A” team should contain the four swimmers with the best times for the stroke and the “B” team the swimmers with the next four best times.

The data for this example can be found [here](#). If you want to follow along, download it to your current working directory and save it as `swimmers.csv`.

Here are the first 10 rows of `swimmers.csv`:

#### Shell

```
$ head -n 10 swimmers.csv
Event,Name,Stroke,Time1,Time2,Time3
0,Emma,freestyle,00:50:313667,00:50:875398,00:50:646837
0,Emma,backstroke,00:56:720191,00:56:431243,00:56:941068
0,Emma,butterfly,00:41:927947,00:42:062812,00:42:007531
0,Emma,breaststroke,00:59:825463,00:59:397469,00:59:385919
0,Olivia,freestyle,00:45:566228,00:46:066985,00:46:044389
0,Olivia,backstroke,00:53:984872,00:54:575110,00:54:932723
0,Olivia,butterfly,01:12:548582,01:12:722369,01:13:105429
0,Olivia,breaststroke,00:49:230921,00:49:604561,00:49:120964
0,Sophia,freestyle,00:55:209625,00:54:790225,00:55:351528
```

The three times in each row represent the times recorded by three different stopwatches, and are given in MM:SS:mmmmmm format (minutes, seconds, microseconds). The accepted time for an event is the *median* of these three times, *not* the average.

Let's start by creating a subclass `Event` of the `namedtuple` object, just like we did in the [SP500 example](#):

#### Python

```
from collections import namedtuple

class Event(namedtuple('Event', ['stroke', 'name', 'time'])):
    __slots__ = ()

    def __lt__(self, other):
        return self.time < other.time
```

The `.stroke` property stores the name of the stroke in the event, `.name` stores the swimmer name, and `.time` records the accepted time for the event. The `__lt__()` dunder method will allow `min()` to be called on a sequence of `Event` objects.

To read the data from the CSV into a tuple of `Event` objects, you can use the `csv.DictReader` object:

#### Python

```
import csv
import datetime
import statistics

def read_events(csvfile, _strptime=datetime.datetime.strptime):
    def _median(times):
        return statistics.median((_strptime(time, '%M:%S:%F').time()
                                   for time in row['Times']))

    fieldnames = ['Event', 'Name', 'Stroke']
    with open(csvfile) as infile:
        reader = csv.DictReader(infile, fieldnames=fieldnames, restkey='Times')
        next(reader) # skip header
        for row in reader:
            yield Event(row['Stroke'], row['Name'], _median(row['Times']))

events = tuple(read_events('swimmers.csv'))
```

The `read_events()` generator reads each row in the `swimmers.csv` file into an `OrderedDict` object in the following line:

#### Python

```
reader = csv.DictReader(infile, fieldnames=fieldnames, restkey='Times')
```

By assigning the `'Times'` field to `restkey`, the “Time1”, “Time2”, and “Time3” columns of each row in the CSV file will be stored in a list on the `'Times'` key of the `OrderedDict` returned by `csv.DictReader`.

For example, the first row of the file (excluding the header row) is read into the following object:

Python

```
OrderedDict([('Event', '0'),
            ('Name', 'Emma'),
            ('Stroke', 'freestyle'),
            ('Times', ['00:50:313667', '00:50:875398', '00:50:646837'])])
```

Next, `read_events()` yields an `Event` object with the stroke, swimmer name, and median time (as a [`datetime.time` object](#)) returned by the `_median()` function, which calls [`statistics.median\(\)`](#) on the list of times in the row.

Since each item in the list of times is read as a string by `csv.DictReader()`, `_median()` uses the [`datetime.datetime.strptime\(\)` classmethod](#) to instantiate a time object from each string.

Finally, a tuple of `Event` objects is created:

Python

```
events = tuple(read_events('swimmers.csv'))
```

The first five elements of `events` look like this:

Python

&gt;&gt;&gt;

```
>>> events[:5]
(Event(stroke='freestyle', name='Emma', time=datetime.time(0, 0, 50, 646837)),
 Event(stroke='backstroke', name='Emma', time=datetime.time(0, 0, 56, 720191)),
 Event(stroke='butterfly', name='Emma', time=datetime.time(0, 0, 42, 7531)),
 Event(stroke='breaststroke', name='Emma', time=datetime.time(0, 0, 59, 397469)),
 Event(stroke='freestyle', name='Olivia', time=datetime.time(0, 0, 46, 44389)))
```

Now that you’ve got the data into memory, what do you do with it? Here’s the plan of attack:

- Group the events by stroke.
- For each stroke:
  - Group its events by swimmer name and determine the best time for each swimmer.
  - Order the swimmers by best time.
  - The first four swimmers make the “A” team for the stroke, and the next four swimmers make the “B” team.

The `itertools.groupby()` function makes grouping objects in an iterable a snap. It takes an iterable `inputs` and a key to group by, and returns an object containing iterators over the elements of `inputs` grouped by the key.

Here’s a simple `groupby()` example:

Python

&gt;&gt;&gt;

```
>>> data = [{'name': 'Alan', 'age': 34},
...         {'name': 'Catherine', 'age': 34},
...         {'name': 'Betsy', 'age': 29},
...         {'name': 'David', 'age': 33}]
...
>>> grouped_data = it.groupby(data, key=lambda x: x['age'])
>>> for key, grp in grouped_data:
...     print('{}: {}'.format(key, list(grp)))
...
34: [{'name': 'Alan', 'age': 34}, {'name': 'Betsy', 'age': 34}]
29: [{'name': 'Catherine', 'age': 29}]
33: [{'name': 'David', 'age': 33}]
```

If no key is specified, `groupby()` defaults to grouping by “identity”—that is, aggregating identical elements in the iterable:

Python

&gt;&gt;&gt;

```
>>> for key, grp in it.groupby([1, 1, 2, 2, 2, 3]):
...     print('{}: {}'.format(key, list(grp)))
...
1: [1, 1]
2: [2, 2, 2]
3: [3]
```

The object returned by `groupby()` is sort of like a dictionary in the sense that the iterators returned are associated with a key. However, unlike a dictionary, it won't allow you to access its values by key name:

Python Traceback

```
>>> grouped_data[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'itertools.groupby' object is not subscriptable
```

In fact, `groupby()` returns an iterator over tuples whose first components are keys and second components are iterators over the grouped data:

Python

&gt;&gt;&gt;

```
>>> grouped_data = it.groupby([1, 1, 2, 2, 2, 3])
>>> list(grouped_data)
[(1, <itertools._grouper object at 0x7ff3056130b8>),
 (2, <itertools._grouper object at 0x7ff3056130f0>),
 (3, <itertools._grouper object at 0x7ff305613128>)]
```

One thing to keep in mind with `groupby()` is that it isn't as smart as you might like. As `groupby()` traverses the data, it aggregates elements until an element with a different key is encountered, at which point it starts a new group:

Python

&gt;&gt;&gt;

```
>>> grouped_data = it.groupby([1, 2, 1, 2, 3, 2])
>>> for key, grp in grouped_data:
...     print('{}: {}'.format(key, list(grp)))
...
1: [1]
2: [2]
1: [1]
2: [2]
3: [3]
2: [2]
```

Compare this to, say, the SQL `GROUP BY` command, which groups elements regardless of their order of appearance.

When working with `groupby()`, you need to sort your data on the same key that you would like to group by. Otherwise, you may get unexpected results. This is so common that it helps to write a utility function to take care of this for you:

Python

```
def sort_and_group(iterable, key=None):
    """Group sorted `iterable` on `key`."""
    return it.groupby(sorted(iterable, key=key), key=key)
```

Returning to the swimmers example, the first thing you need to do is create a for loop that iterates over the data in the `events` tuple grouped by stroke:

Python

```
for stroke, evts in sort_and_group(events, key=lambda evt: evt.stroke):
```

Next, you need to group the `evts` iterator by swimmer name inside of the above for loop:

Python

```
events_by_name = sort_and_group(evts, key=lambda evt: evt.name)
```

To calculate the best time for each swimmer in `events_by_name`, you can call `min()` on the events in that swimmers group. (This works because you implemented the `__lt__()` dunder method in the `Events` class.)

Python

```
best_times = (min(evt) for _, evt in events_by_name)
```

Note that the `best_times` generator yields `Event` objects containing the best stroke time for each swimmer. To build the relay teams, you'll need to sort `best_times` by time and aggregate the result into groups of four. To aggregate the results, you can use the `grouper()` function from [The grouper\(\) recipe](#) section and use `islice()` to grab the first two groups.

Python

```
sorted_by_time = sorted(best_times, key=lambda evt: evt.time)
teams = zip(('A', 'B'), it.islice(grouper(sorted_by_time, 4), 2))
```

Now `teams` is an iterator over exactly two tuples representing the “A” and the “B” team for the stroke. The first component of each tuple is the letter “A” or “B”, and the second component is an iterator over `Event` objects containing the swimmers in the team. You can now print the results:

Python

```
for team, swimmers in teams:
    print('{stroke} {team}: {names}'.format(
        stroke=stroke.capitalize(),
        team=team,
        names=', '.join(swimmer.name for swimmer in swimmers)
    ))
```

Here's the full script:

Python

```

from collections import namedtuple
import csv
import datetime
import itertools as it
import statistics

class Event(namedtuple('Event', ['stroke', 'name', 'time'])):
    __slots__ = ()

    def __lt__(self, other):
        return self.time < other.time

def sort_and_group(iterable, key=None):
    return it.groupby(sorted(iterable, key=key), key=key)

def grouper(iterable, n, fillvalue=None):
    iters = [iter(iterable)] * n
    return it.zip_longest(*iters, fillvalue=fillvalue)

def read_events(csvfile, _strptime=datetime.datetime.strptime):
    def _median(times):
        return statistics.median((_strptime(time, '%M:%S:%f').time()
                                     for time in row['Times']))

    fieldnames = ['Event', 'Name', 'Stroke']
    with open(csvfile) as infile:
        reader = csv.DictReader(infile, fieldnames=fieldnames, restkey='Times')
        next(reader) # Skip header.
        for row in reader:
            yield Event(row['Stroke'], row['Name'], _median(row['Times']))

events = tuple(read_events('swimmers.csv'))

for stroke, evts in sort_and_group(events, key=lambda evt: evt.stroke):
    events_by_name = sort_and_group(evts, key=lambda evt: evt.name)
    best_times = (min(evt) for _, evt in events_by_name)
    sorted_by_time = sorted(best_times, key=lambda evt: evt.time)
    teams = zip(('A', 'B'), it.islice(grouper(sorted_by_time, 4), 2))
    for team, swimmers in teams:
        print('{stroke} {team}: {names}'.format(
            stroke=stroke.capitalize(),
            team=team,
            names=', '.join(swimmer.name for swimmer in swimmers)
        ))

```

If you run the above code, you'll get the following output:

Shell

```

Backstroke A: Sophia, Grace, Penelope, Addison
Backstroke B: Elizabeth, Audrey, Emily, Aria
Breaststroke A: Samantha, Avery, Layla, Zoe
Breaststroke B: Lillian, Aria, Ava, Alexa
Butterfly A: Audrey, Leah, Layla, Samantha
Butterfly B: Alexa, Zoey, Emma, Madison
Freestyle A: Aubrey, Emma, Olivia, Evelyn
Freestyle B: Elizabeth, Zoe, Addison, Madison

```

## Write Cleaner & More Pythonic Code

realpython.com



[Remove ads](#)



# Where to Go From Here

If you have made it this far, congratulations! I hope you have enjoyed the journey.

`itertools` is a powerful module in the Python standard library, and an essential tool to have in your toolkit. With it, you can write faster and more memory efficient code that is often simpler and easier to read (although that is not always the case, as you saw in the section on [second order recurrence relations](#)).

If anything, though, `itertools` is a testament to the power of iterators and [lazy evaluation](#). Even though you have seen many techniques, this article only scratches the surface.

So I guess this means your journey is only just beginning.

**Free Bonus:** [Click here to get our itertools cheat sheet](#) that summarizes the techniques demonstrated in this tutorial.

In fact, this article skipped two `itertools` functions: [starmap\(\)](#) and [compress\(\)](#). In my experience, these are two of the lesser used `itertools` functions, but I urge you to read their docs an experiment with your own use cases!

Here are a few places where you can find more examples of `itertools` in action (thanks to Brad Solomon for these fine suggestions):

- [What is the Purpose of `itertools.repeat\(\)`?](#)
- [Fastest Way to Generate a Random-like Unique String With Random Length in Python 3](#)
- [Write a Pandas DataFrame to a String Buffer with Chunking](#)

Finally, for even more tools for constructing iterators, take a look at [more-itertools](#).

Do you have any favorite `itertools` recipes/use-cases? We would love to hear about them in the comments!

*We would like to thank our readers Putter and Samir Aghayev for pointing out a couple of errors in the original version of this article.*

Mark as Completed

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

Email Address

Send Me Python Tricks »

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

About **David Amos**