

How to Use Generators and yield in Python

by [Kyle Stratis](#) · Sep 25, 2019 · 16 Comments · [intermediate](#) [python](#)

[Tweet](#) [Share](#) [Email](#)

Table of Contents

- [Using Generators](#)
 - [Example 1: Reading Large Files](#)
 - [Example 2: Generating an Infinite Sequence](#)
 - [Example 3: Detecting Palindromes](#)
- [Understanding Generators](#)
 - [Building Generators With Generator Expressions](#)
 - [Profiling Generator Performance](#)
- [Understanding the Python Yield Statement](#)
- [Using Advanced Generator Methods](#)
 - [How to Use .send\(\)](#)
 - [How to Use .throw\(\)](#)
 - [How to Use .close\(\)](#)
- [Creating Data Pipelines With Generators](#)
- [Conclusion](#)





Set Up Your **Python** Environment in minutes

Automatically build your Python runtime with the packages you need

BUILD FOR FREE

Linux and Windows

Have you ever had to work with a dataset so large that it overwhelmed your machine’s memory? Or maybe you have a complex function that needs to maintain an internal state every time it’s called, but the function is too small to justify creating its own class. In these cases and more, generators and the Python yield statement are here to help.

By the end of this article, you’ll know:

- What **generators** are and how to use them
- How to create **generator functions and expressions**
- How the **Python yield** statement works
- How to use **multiple** Python yield statements in a generator function

[Help](#)

- How to use **advanced generator methods**
- How to **build data pipelines** with multiple generators

If you're a beginner or intermediate Pythonista and you're interested in learning how to work with large datasets in a more Pythonic fashion, then this is the tutorial for you.

You can get a copy of the dataset used in this tutorial by clicking the link below:

Download Dataset: [Click here to download the dataset you'll use in this tutorial](#) to learn about generators and yield in Python.

Using Generators

Introduced with [PEP 255](#), **generator functions** are a special kind of function that return a [lazy iterator](#). These are objects that you can loop over like a list. However, unlike lists, lazy iterators do not store their contents in memory. For an overview of iterators in Python, take a look at [Python “for” Loops \(Definite Iteration\)](#).

Now that you have a rough idea of what a generator does, you might wonder what they look like in action. Let's take a look at two examples. In the first, you'll see how generators work from a bird's eye view. Then, you'll zoom in and examine each example more thoroughly.

Example 1: Reading Large Files

A common use case of generators is to work with data streams or large files, like [CSV files](#). These text files separate data into columns by using commas. This format is a common way to share data. Now, what if you want to count the number of rows in a CSV file? The code block below shows one way of counting those rows:

Python

```
csv_gen = csv_reader("some_csv.txt")
row_count = 0

for row in csv_gen:
    row_count += 1

print(f"Row count is {row_count}")
```

Looking at this example, you might expect `csv_gen` to be a list. To populate this list, `csv_reader()` opens a file and loads its contents into `csv_gen`. Then, the program iterates over the list and increments `row_count` for each row.

This is a reasonable explanation, but would this design still work if the file is very large? What if the file is larger than the memory you have available? To answer this question, let's assume that `csv_reader()` just opens the file and reads it into an array:

Python

```
def csv_reader(file_name):
    file = open(file_name)
    result = file.read().split("\n")
    return result
```

This function opens a given file and uses `file.read()` along with `.split()` to add each line as a separate element to a list. If you were to use this version of `csv_reader()` in the row counting code block you saw further up, then you'd get the following output:

Python

>>>

```
Traceback (most recent call last):
  File "ex1_naive.py", line 22, in <module>
    . , ,
```

```
main()
File "ex1_naive.py", line 13, in main
    csv_gen = csv_reader("file.txt")
File "ex1_naive.py", line 6, in csv_reader
    result = file.read().split("\n")
MemoryError
```

In this case, `open()` returns a generator object that you can lazily iterate through line by line. However, `file.read().split()` loads everything into memory at once, causing the `MemoryError`.

Before that happens, you'll probably notice your computer slow to a crawl. You might even need to kill the program with a `KeyboardInterrupt`. So, how can you handle these huge data files? Take a look at a new definition of `csv_reader()`:

Python

```
def csv_reader(file_name):
    for row in open(file_name, "r"):
        yield row
```

In this version, you open the file, iterate through it, and yield a row. This code should produce the following output, with no memory errors:

Shell

```
Row count is 64186394
```

What's happening here? Well, you've essentially turned `csv_reader()` into a generator function. This version opens a file, loops through each line, and yields each row, instead of returning it.

You can also define a **generator expression** (also called a **generator comprehension**), which has a very similar syntax to [list comprehensions](#). In this way, you can use the generator without calling a function:

Python

```
csv_gen = (row for row in open(file_name))
```

This is a more succinct way to create the list `csv_gen`. You'll learn more about the Python `yield` statement soon. For now, just remember this key difference:

- Using `yield` will result in a generator object.
- Using `return` will result in the first line of the file *only*.

Example 2: Generating an Infinite Sequence

Let's switch gears and look at infinite sequence generation. In Python, to get a finite sequence, you call `range()` and evaluate it in a list context:

Python

>>>

```
>>> a = range(5)
>>> list(a)
[0, 1, 2, 3, 4, 5]
```

Generating an **infinite sequence**, however, will require the use of a generator, since your computer memory is finite:

Python

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

This code block is short and sweet. First, you initialize the variable `num` and start an infinite loop. Then, you immediately `yield num` so that you can capture the initial state. This mimics the action of `range()`.

After `yield`, you increment `num` by 1. If you try this with a `for` loop, then you'll see that it really does seem infinite:

Python

>>>

```
>>> for i in infinite_sequence():
...     print(i, end=" ")
...
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 40 41 42
[...]
```

6157818 6157819 6157820 6157821 6157822 6157823 6157824 6157825 6157826 6157827
6157828 6157829 6157830 6157831 6157832 6157833 6157834 6157835 6157836 6157837
6157838 6157839 6157840 6157841 6157842

KeyboardInterrupt
Traceback (most recent call last):
 File "<stdin>", line 2, in <module>

The program will continue to execute until you stop it manually.

Instead of using a `for` loop, you can also call `next()` on the generator object directly. This is especially useful for testing a generator in the console:

Python

>>>

```
>>> gen = infinite_sequence()
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
```

Here, you have a generator called `gen`, which you manually iterate over by repeatedly calling `next()`. This works as a great sanity check to make sure your generators are producing the output you expect.

Note: When you use `next()`, Python calls `.__next__()` on the function you pass in as a parameter. There are some special effects that this parameterization allows, but it goes beyond the scope of this article. Experiment with changing the parameter you pass to `next()` and see what happens!

Example 3: Detecting Palindromes

You can use infinite sequences in many ways, but one practical use for them is in building palindrome detectors. A **palindrome detector** will locate all sequences of letters or numbers that are palindromes. These are words or numbers that are read the same forward and backward, like 121. First, define your numeric palindrome detector:

Python

```
def is_palindrome(num):
    # Skip single-digit inputs
    if num // 10 == 0:
        return False
    temp = num
    reversed_num = 0

    while temp != 0:
        reversed_num = (reversed_num * 10) + (temp % 10)
```

```
temp = temp // 10

if num == reversed_num:
    return num
else:
    return False
```

Don't worry too much about understanding the underlying math in this code. Just note that the function takes an input number, reverses it, and checks to see if the reversed number is the same as the original. Now you can use your infinite sequence generator to get a running list of all numeric palindromes:

Python

>>>

```
>>> for i in infinite_sequence():
...     pal = is_palindrome(i)
...     if pal:
...         print(pal)
...
11
22
33
[...]
99799
99899
99999
100001
101101
102201
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 5, in is_palindrome
```

In this case, the only numbers that are printed to the console are those that are the same forward or backward.

Note: In practice, you're unlikely to write your own infinite sequence generator. The [itertools](#) module provides a very efficient infinite sequence generator with `itertools.count()`.

Now that you've seen a simple use case for an infinite sequence generator, let's dive deeper into how generators work.

Understanding Generators

So far, you've learned about the two primary ways of creating generators: by using generator functions and generator expressions. You might even have an intuitive understanding of how generators work. Let's take a moment to make that knowledge a little more explicit.

Generator functions look and act just like regular functions, but with one defining characteristic. Generator functions use the Python `yield` keyword instead of `return`. Recall the generator function you wrote earlier:

Python

```
def infinite_sequence():
    num = 0
    while True:
        yield num
        num += 1
```

This looks like a typical function definition, except for the Python `yield` statement and the code that follows it. `yield` indicates where a value is sent back to the caller, but unlike `return`, you don't exit the function afterward.

Instead, the **state** of the function is remembered. That way, when `next()` is called on a generator object (either explicitly or implicitly within a `for` loop), the previously yielded variable `num` is incremented, and then yielded again.

Since generator functions look like other functions and act very similarly to them, you can assume that generator expressions are very similar to other comprehensions available in Python.

Note: Are you rusty on Python’s list, set, and dictionary comprehensions? You can check out [Using List Comprehensions Effectively](#).

Building Generators With Generator Expressions

Like list comprehensions, generator expressions allow you to quickly create a generator object in just a few lines of code. They’re also useful in the same cases where list comprehensions are used, with an added benefit: you can create them without building and holding the entire object in memory before iteration. In other words, you’ll have no memory penalty when you use generator expressions. Take this example of squaring some numbers:

Python

>>>

```
>>> nums_squared_lc = [num**2 for num in range(5)]
>>> nums_squared_gc = (num**2 for num in range(5))
```

Both `nums_squared_lc` and `nums_squared_gc` look basically the same, but there’s one key difference. Can you spot it? Take a look at what happens when you inspect each of these objects:

Python

>>>

```
>>> nums_squared_lc
[0, 1, 4, 9, 16]
>>> nums_squared_gc
<generator object <genexpr> at 0x107fbbc78>
```

The first object used brackets to build a list, while the second created a generator expression by using parentheses. The output confirms that you’ve created a generator object and that it is distinct from a list.

Profiling Generator Performance

You learned earlier that generators are a great way to optimize memory. While an infinite sequence generator is an extreme example of this optimization, let’s amp up the number squaring examples you just saw and inspect the size of the resulting objects. You can do this with a call to `sys.getsizeof()`:

Python

>>>

```
>>> import sys
>>> nums_squared_lc = [i * 2 for i in range(10000)]
>>> sys.getsizeof(nums_squared_lc)
87624
>>> nums_squared_gc = (i ** 2 for i in range(10000))
>>> print(sys.getsizeof(nums_squared_gc))
120
```

In this case, the list you get from the list comprehension is 87,624 bytes, while the generator object is only 120. This means that the list is over 700 times larger than the generator object!

There is one thing to keep in mind, though. If the list is smaller than the running machine’s available memory, then list comprehensions can be [faster to evaluate](#) than the equivalent generator expression. To explore this, let’s sum across the results from the two comprehensions above. You can generate a readout with `cProfile.run()`:

Python

>>>

```
>>> import cProfile
>>> cProfile.run('sum([i * 2 for i in range(10000)])')
5 function calls in 0.001 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1   0.001    0.001    0.001    0.001 <string>:1(<listcomp>)
      1   0.000    0.000    0.001    0.001 <string>:1(<module>)
      1   0.000    0.000    0.001    0.001 {built-in method builtins.exec}
```



```

1    0.000    0.000    0.000    0.000 {built-in method builtins.sum}
1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

>>> cProfile.run('sum((i * 2 for i in range(10000)))')
10005 function calls in 0.003 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
10001    0.002    0.000    0.002    0.000 <string>:1(<genexpr>)
1      0.000    0.000    0.003    0.003 <string>:1(<module>)
1      0.000    0.000    0.003    0.003 {built-in method builtins.exec}
1      0.001    0.001    0.003    0.003 {built-in method builtins.sum}
1      0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}

```

Here, you can see that summing across all values in the list comprehension took about a third of the time as summing across the generator. If speed is an issue and memory isn't, then a list comprehension is likely a better tool for the job.

Note: These measurements aren't only valid for objects made with generator expressions. They're also the same for objects made from the analogous generator function since the resulting generators are equivalent.

Remember, list comprehensions return full lists, while generator expressions return generators. Generators work the same whether they're built from a function or an expression. Using an expression just allows you to define simple generators in a single line, with an assumed `yield` at the end of each inner iteration.

The Python `yield` statement is certainly the linchpin on which all of the functionality of generators rests, so let's dive into how `yield` works in Python.

Understanding the Python Yield Statement

On the whole, `yield` is a fairly simple statement. Its primary job is to control the flow of a generator function in a way that's similar to `return` statements. As briefly mentioned above, though, the Python `yield` statement has a few tricks up its sleeve.

When you call a generator function or use a generator expression, you return a special iterator called a generator. You can assign this generator to a variable in order to use it. When you call special methods on the generator, such as `next()`, the code within the function is executed up to `yield`.

When the Python `yield` statement is hit, the program suspends function execution and returns the yielded value to the caller. (In contrast, `return` stops function execution completely.) When a function is suspended, the state of that function is saved. This includes any variable bindings local to the generator, the instruction pointer, the internal stack, and any exception handling.

This allows you to resume function execution whenever you call one of the generator's methods. In this way, all function evaluation picks back up right after `yield`. You can see this in action by using multiple Python `yield` statements:

Python

>>>

```

>>> def multi_yield():
...     yield_str = "This will print the first string"
...     yield yield_str
...     yield_str = "This will print the second string"
...     yield yield_str
...
>>> multi_obj = multi_yield()
>>> print(next(multi_obj))
This will print the first string
>>> print(next(multi_obj))
This will print the second string
>>> print(next(multi_obj))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

```

StopIteration

Take a closer look at that last calls to `next()`. You can see that execution has blown up with a [traceback](#). This is because generators, like all iterators, can be exhausted. Unless your generator is infinite, you can iterate through it one time only. Once all values have been evaluated, iteration will stop and the `for` loop will exit. If you used `next()`, then instead you'll get an explicit `StopIteration` exception.

Note: `StopIteration` is a natural exception that's raised to signal the end of an iterator. `for` loops, for example, are built around `StopIteration`. You can even implement your own `for` loop by using a `while` loop:

Python

>>>

```
>>> letters = ["a", "b", "c", "y"]
>>> it = iter(letters)
>>> while True:
...     try:
...         letter = next(it)
...     except StopIteration:
...         break
...     print(letter)
...
a
b
c
y
```

You can read more about `StopIteration` in the Python documentation on [exceptions](#). For more on iteration in general, check out [Python “for” Loops \(Definite Iteration\)](#) and [Python “while” Loops \(Indefinite Iteration\)](#).

`yield` can be used in many ways to control your generator's execution flow. The use of multiple Python `yield` statements can be leveraged as far as your creativity allows.

Using Advanced Generator Methods

You've seen the most common uses and constructions of generators, but there are a few more tricks to cover. In addition to `yield`, generator objects can make use of the following methods:

- `.send()`
- `.throw()`
- `.close()`

How to Use `.send()`

For this next section, you're going to build a program that makes use of all three methods. This program will print numeric palindromes like before, but with a few tweaks. Upon encountering a palindrome, your new program will add a digit and start a search for the next one from there. You'll also handle exceptions with `.throw()` and stop the generator after a given amount of digits with `.close()`. First, let's recall the code for your palindrome detector:

Python

```
def is_palindrome(num):
    # Skip single-digit inputs
    if num // 10 == 0:
        return False
    temp = num
    reversed_num = 0

    while temp != 0:
        reversed_num = (reversed_num * 10) + (temp % 10)
        temp = temp // 10

    if num == reversed_num:
        return True
    else:
        return False
```


This is the same code you saw earlier, except that now the program returns strictly `True` or `False`. You'll also need to modify your original infinite sequence generator, like so:

Python

```
1 def infinite_palindromes():
2     num = 0
3     while True:
4         if is_palindrome(num):
5             i = (yield num)
6             if i is not None:
7                 num = i
8         num += 1
```

There are a lot of changes here! The first one you'll see is in line 5, where `i = (yield num)`. Though you learned earlier that `yield` is a statement, that isn't quite the whole story.

As of Python 2.5 (the same release that introduced the methods you are learning about now), `yield` is an **expression**, rather than a statement. Of course, you can still use it as a statement. But now, you can also use it as you see in the code block above, where `i` takes the value that is yielded. This allows you to manipulate the yielded value. More importantly, it allows you to `.send()` a value back to the generator. When execution picks up after `yield`, `i` will take the value that is sent.

You'll also check `if i is not None`, which could happen if `next()` is called on the generator object. (This can also happen when you iterate with a `for` loop.) If `i` has a value, then you update `num` with the new value. But regardless of whether or not `i` holds a value, you'll then increment `num` and start the loop again.

Now, take a look at the main function code, which sends the lowest number with another digit back to the generator. For example, if the palindrome is 121, then it will `.send()` 1000:

Python

```
pal_gen = infinite_palindromes()
for i in pal_gen:
    digits = len(str(i))
    pal_gen.send(10 ** (digits))
```

With this code, you create the generator object and iterate through it. The program only yields a value once a palindrome is found. It uses `len()` to determine the number of digits in that palindrome. Then, it sends `10 ** digits` to the generator. This brings execution back into the generator logic and assigns `10 ** digits` to `i`. Since `i` now has a value, the program updates `num`, increments, and checks for palindromes again.

Once your code finds and yields another palindrome, you'll iterate via the `for` loop. This is the same as iterating with `next()`. The generator also picks up at line 5 with `i = (yield num)`. However, now `i` is `None`, because you didn't explicitly send a value.

What you've created here is a **coroutine**, or a generator function into which you can pass data. These are useful for constructing data pipelines, but as you'll see soon, they aren't necessary for building them. (If you're looking to dive deeper, then [this course on coroutines and concurrency](#) is one of the most comprehensive treatments available.)

Now that you've learned about `.send()`, let's take a look at `.throw()`.

How to Use `.throw()`

`.throw()` allows you to throw exceptions with the generator. In the below example, you raise the exception in line 6. This code will throw a `ValueError` once `digits` reaches 5:

Python

```
1 pal_gen = infinite_palindromes()
2 for i in pal_gen:
3     print(i)
4     digits = len(str(i))
5     if digits == 5:
6         pal_gen.throw(ValueError("We don't like large palindromes"))
7     pal_gen.send(10 ** (digits))
```

This is the same as the previous code, but now you'll check if `digits` is equal to 5. If so, then you'll `.throw()` a `ValueError`. To confirm that this works as expected, take a look at the code's output:

Python

>>>

```
11
111
1111
10101
Traceback (most recent call last):
  File "advanced_gen.py", line 47, in <module>
    main()
  File "advanced_gen.py", line 41, in main
    pal_gen.throw(ValueError("We don't like large palindromes"))
  File "advanced_gen.py", line 26, in infinite_palindromes
    i = (yield num)
ValueError: We don't like large palindromes
```

`.throw()` is useful in any areas where you might need to catch an [exception](#). In this example, you used `.throw()` to control when you stopped iterating through the generator. You can do this more elegantly with `.close()`.

How to Use `.close()`

As its name implies, `.close()` allows you to stop a generator. This can be especially handy when controlling an infinite sequence generator. Let's update the code above by changing `.throw()` to `.close()` to stop the iteration:

Python

```
1 pal_gen = infinite_palindromes()
2 for i in pal_gen:
3     print(i)
4     digits = len(str(i))
5     if digits == 5:
6         pal_gen.close()
7     pal_gen.send(10 ** (digits))
```

Instead of calling `.throw()`, you use `.close()` in line 6. The advantage of using `.close()` is that it raises `StopIteration`, an exception used to signal the end of a finite iterator:

Python

>>>

```
11
111
1111
10101
Traceback (most recent call last):
  File "advanced_gen.py", line 46, in <module>
    main()
  File "advanced_gen.py", line 42, in main
    pal_gen.send(10 ** (digits))
StopIteration
```

Now that you've learned more about the special methods that come with generators, let's talk about using generators to build data pipelines.

Creating Data Pipelines With Generators

Data pipelines allow you to string together code to process large datasets or streams of data without maxing out your machine's memory. Imagine that you have a large CSV file:

CSV

```
permalink,company,numEmps,category,city,state,fundedDate,raisedAmt,raisedCurrency,round
digg,Digg,60,web,San Francisco,CA,1-Dec-06,8500000,USD,b
digg,Digg,60,web,San Francisco,CA,1-Oct-05,2800000,USD,a
facebook,Facebook,450,web,Palo Alto,CA,1-Sep-04,500000,USD,angel
facebook,Facebook,450,web,Palo Alto,CA,1-May-05,12700000,USD,a
```

photobucket, Photobucket, 60, web, Palo Alto, CA, 1-Mar-05, 3000000, USD, a

This example is pulled from the TechCrunch Continental USA set, which describes funding rounds and dollar amounts for various startups based in the USA. Click the link below to download the dataset:

Download Dataset: [Click here to download the dataset you'll use in this tutorial](#) to learn about generators and yield in Python.

It's time to do some processing in Python! To demonstrate how to build pipelines with generators, you're going to analyze this file to get the total and average of all series A rounds in the dataset.

Let's think of a strategy:

1. Read every line of the file.
2. Split each line into a list of values.
3. Extract the column names.
4. Use the column names and lists to create a dictionary.
5. Filter out the rounds you aren't interested in.
6. Calculate the total and average values for the rounds you are interested in.

Normally, you can do this with a package like [pandas](#), but you can also achieve this functionality with just a few generators. You'll start by reading each line from the file with a generator expression:

Python

```
1 file_name = "techcrunch.csv"
2 lines = (line for line in open(file_name))
```

Then, you'll use another generator expression in concert with the previous one to split each line into a list:

Python

```
3 list_line = (s.rstrip().split(",") for s in lines)
```

Here, you created the generator `list_line`, which iterates through the first generator `lines`. This is a common pattern to use when designing generator pipelines. Next, you'll pull the column names out of `techcrunch.csv`. Since the column names tend to make up the first line in a CSV file, you can grab that with a short `next()` call:

Python

```
4 cols = next(list_line)
```

This call to `next()` advances the iterator over the `list_line` generator one time. Put it all together, and your code should look something like this:

Python

```
1 file_name = "techcrunch.csv"
2 lines = (line for line in open(file_name))
3 list_line = (s.rstrip().split(",") for s in lines)
4 cols = next(list_line)
```

To sum this up, you first create a generator expression `lines` to yield each line in a file. Next, you iterate through that generator within the definition of *another* generator expression called `list_line`, which turns each line into a list of values. Then, you advance the iteration of `list_line` just once with `next()` to get a list of the column names from your CSV file.

Note: Watch out for trailing newlines! This code takes advantage of `.rstrip()` in the `list_line` generator expression to make sure there are no trailing newline characters, which can be present in CSV files.

To help you filter and perform operations on the data, you'll create dictionaries where the keys are the column names from the CSV:

Python

```
5 | company_dicts = (dict(zip(cols, data)) for data in list_line)
```

This generator expression iterates through the lists produced by `list_line`. Then, it uses `zip()` and `dict()` to create the dictionary as specified above. Now, you'll use a *fourth* generator to filter the funding round you want and pull `raisedAmt` as well:

Python

```
6 | funding = (
7 |     int(company_dict["raisedAmt"])
8 |     for company_dict in company_dicts
9 |     if company_dict["round"] == "a"
10 | )
```

In this code snippet, your generator expression iterates through the results of `company_dicts` and takes the `raisedAmt` for any `company_dict` where the `round` key is A.

Remember, you aren't iterating through all these at once in the generator expression. In fact, you aren't iterating through anything until you actually use a for loop or a function that works on iterables, like `sum()`. In fact, call `sum()` now to iterate through the generators:

Python

```
11 | total_series_a = sum(funding)
```

Putting this all together, you'll produce the following script:

Python

```
1 | file_name = "techcrunch.csv"
2 | lines = (line for line in open(file_name))
3 | list_line = (s.rstrip().split(",") for s in lines)
4 | cols = next(list_line)
5 | company_dicts = (dict(zip(cols, data)) for data in list_line)
6 | funding = (
7 |     int(company_dict["raisedAmt"])
8 |     for company_dict in company_dicts
9 |     if company_dict["round"] == "A"
10 | )
11 | total_series_a = sum(funding)
12 | print(f"Total series A fundraising: ${total_series_a}")
```

This script pulls together every generator you've built, and they all function as one big data pipeline. Here's a line by line breakdown:

- **Line 2** reads in each line of the file.
- **Line 3** splits each line into values and puts the values into a list.
- **Line 4** uses `next()` to store the column names in a list.
- **Line 5** creates dictionaries and unites them with a `zip()` call:
 - **The keys** are the column names `cols` from line 4.
 - **The values** are the rows in list form, created in line 3.
- **Line 6** gets each company's series A funding amounts. It also filters out any other raised amount.
- **Line 11** begins the iteration process by calling `sum()` to get the total amount of series A funding found in the CSV.

When you run this code on `techcrunch.csv`, you should find a total of \$4,376,015,000 raised in series A funding rounds.

Note: The methods for handling CSV files developed in this tutorial are important for understanding how to use generators and the Python `yield` statement. However, when you work with CSV files in Python, you should instead use the [csv](#) module included in Python's standard library. This module has optimized methods for

handling CSV files efficiently.

To dig even deeper, try figuring out the average amount raised *per company* in a series A round. This is a bit trickier, so here are some hints:

- Generators exhaust themselves after being iterated over fully.
- You will still need the `sum()` function.

Good luck!

Conclusion

In this tutorial, you’ve learned about **generator functions** and **generator expressions**.



You now know:

- How to use and write generator functions and generator expressions
- How the all-important **Python yield statement** enables generators
- How to use **multiple** Python yield statements in a generator function
- How to use `.send()` to send data to a generator
- How to use `.throw()` to raise generator exceptions
- How to use `.close()` to stop a generator’s iteration
- How to **build a generator pipeline** to efficiently process large CSV files

You can get the dataset you used in this tutorial at the link below:

Download Dataset: [Click here to download the dataset you'll use in this tutorial](#) to learn about generators and yield in Python.

How have generators helped you in your work or projects? If you’re just learning about them, then how do you plan to use them in the future? Did you find a good solution to the data pipeline problem? Let us know in the comments below!

 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 Kyle Stratis