**Python’s ‘yield’ Method: Your Guide to Supercharged Loops!**

[[Arjun Prakash](https://medium.com/@arjunprakash027?source=post_page-----743ad6bfe447--------------------------------)](https://medium.com/@arjunprakash027?source=post_page-----743ad6bfe447--------------------------------)

[Arjun Prakash](https://medium.com/@arjunprakash027?source=post_page-----743ad6bfe447--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F376a20930ef1&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40arjunprakash027%2Fpythons-yield-method-your-guide-to-supercharged-loops-743ad6bfe447&user=Arjun+Prakash&userId=376a20930ef1&source=post_page-376a20930ef1----743ad6bfe447---------------------post_header-----------)

7 min read

·

Jul 23

132

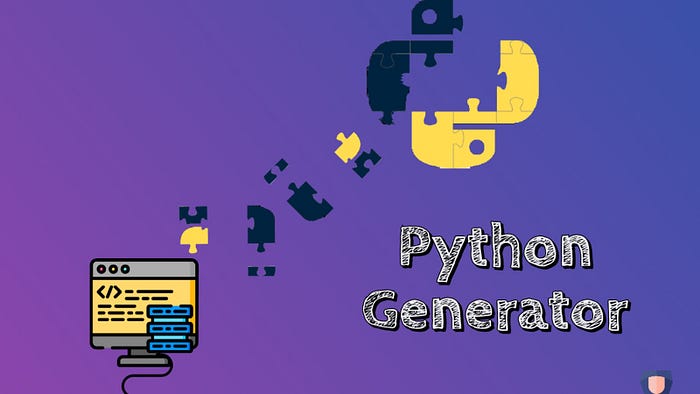


image from [Python Generator — Yield and Next — DEV Community](https://dev.to/divshekhar/python-generator-yield-and-next-306g)

**Introduction**

If you’ve ever found yourself dealing with massive datasets or pondering over infinite sequences, then generators are your secret weapon. In this blog, we’ll unravel the wonders of Python’s yield statement, which allows you to create efficient, on-the-fly data generators.

Whether you’re a Python newbie or an experienced coder, by the end of this guide, you’ll be wielding the power of yield like a pro. Let's get started!

**Generators and Yield statement**

**What are Generators**

*Generator* in python is a function that returns *iterator*using *yield* keyword

An iterator is an object that contains a countable number of values that can be iterated upon, meaning that we can traverse through all the values.

Generator function is like a normal function in python, but instead of using return keyword, it uses yield keyword. If the body of function contains yield, its automatically a python generator.

def function\_name():  
 yield statement

**The need for Generators**

Iterators are objects that can be iterated upon. It is used to abstract a container of data to make it behave like iterable object. List, Dicts and strings are some example of iterable objects.

Iterators don't compute the value of each item when instantiated, but only when we ask for it. This is called lazy evaluation and it can be useful when we have a large dataset to compute.

Iterators can only be iterated once, if we try to iterate over same iterator object, it will give only empty list.

Generators are the easiest way to create a iterable in python.

**Creating Generator Functions: Generator vs Normal Function**

**Basic Generator Function**

We will create a basic fibonacci number function to demonstrate generators in python.

def fib(limit):  
   
 # Initialize first two Fibonacci Numbers  
 a, b = 0, 1  
   
 # One by one yield next Fibonacci Number  
 while a < limit:  
 yield a  
 a, b = b, a + b  
   
# Create a generator object  
x = fib(5)

The function yields the result for 5 Fibonacci numbers in form of an iterable object.

There are 2 ways to traverse through the iterable object returned by yield keyword of a generator function.

*Method 1:*

print(next(x))  
print(next(x))  
print(next(x))  
print(next(x))  
print(next(x))

*Output:*

0  
1  
1  
2  
3

This method uses \_\_next\_\_() method to go though each and every value/ output stored in iterable object.

*Method 2:*

for i in fib(5):  
 print(i)

*Output:*

0  
1  
1  
2  
3

This method uses for loop to go through all the output returned by the iterable object.

**Generator execution flow**

1. A generator function is defined using yield instead of return.
2. Calling the generator function returns a generator object without executing the function’s code.
3. Generator execution starts when iterating over the generator or using \_\_next\_\_()/next().
4. Execution proceeds until the first yield statement is encountered.
5. The generator pauses, yielding the value specified after yield to the caller.
6. Generator state is saved, allowing it to resume from that point on next iteration.
7. Upon subsequent iteration, the generator continues from where it left off.
8. Execution proceeds until the next yield statement or function completion.
9. The generator can have multiple yield statements, each pausing and returning a value.
10. The generator keeps yielding until there are no more yield statements or a return statement is encountered.
11. Exhausted generators raise a StopIteration exception upon further iteration attempts.

**Generators vs Regular Functions**

***Normal Function***

def find\_even\_number\_function(number\_stream):  
 even\_number = []  
 for n in number\_stream:  
 if n % 2 == 0:  
 even\_number.append(n)  
 return even\_number  
  
for i in find\_even\_number\_function(1000000000000000000000000000000000000):  
 print i

The code is quite simple and straightforward, but its builds the full list in memory. This is clearly not acceptable in our case, because we cannot afford to keep all 1000000000000000000000000000000000000 in memory. As you can see the function will never stop. In this case we are going to use generator.

***Generator Function***

def find\_even\_number\_generator(number\_stream):  
 for n in number\_stream:  
 if n % 2 == 0:  
 yield n  
  
for i in find\_even\_number\_generator(1000000000000000000000000000000000000):  
 print i

Generator does not store all the 1000000000000000000000000000000000000 in memory, but instead only execute each step when next() is called, saving memory, but to run all the 1000000000000000000000000000000000000, it will take a lot of time.

So in simple words, if you use a normal function for the above case, you will run out of memory. Alternatively if you use generator function you will run out of time.

**Sending values to Generator**

The generator.send() method is used to send a value into a generator function while it is paused at a yield statement. It allows you to not only get values from the generator but also send values back into it, influencing its behavior dynamically. This capability adds more flexibility and interactivity to generator functions.

*Example*

def squared\_numbers\_generator():  
 num = 0  
 while True:  
 initial\_value = yield num\*\*2  
 if initial\_value is not None:  
 num = initial\_value  
 num += 1  
  
# Create the generator object  
generator = squared\_numbers\_generator()  
  
# Start the generator by calling next()  
next(generator)  
  
# Get the first squared number (0)  
print(generator.send(0)) # Output: 0  
  
# Get the next squared number (1)  
print(generator.send(None)) # Output: 1  
  
# Get the next squared number (4)  
print(generator.send(None)) # Output: 4  
  
# Change the starting value to 5 and get the next squared number (25)  
print(generator.send(5)) # Output: 25  
  
# Get the next squared number (36)  
print(generator.send(None)) # Output: 36

* We define the squared\_numbers\_generator() function, which is an infinite generator that produces squared numbers.
* Inside the generator, we use yield num\*\*2 to generate the squared value of num in each iteration.
* When yield is encountered, the generator pauses, and the yielded value is returned.
* Using generator.send(value), we send a new initial value into the generator, which becomes the value of initial\_value.
* The generator checks if initial\_value is not None. If it is not None, the generator updates num to the new initial value.
* The generator continues running, and the updated num is used to calculate the next squared number when it encounters the next yield statement.

**Yield From keyword**

Yield from keyword is used to work with nested generators.

def sub\_generator():  
 yield 1  
 yield 2  
  
def main\_generator():  
 yield 0  
 yield from sub\_generator()  
 yield 3  
  
# Using the main generator  
for num in main\_generator():  
 print(num)

* We have two generator functions, sub\_generator() and main\_generator().
* sub\_generator() yields two values, 1 and 2.
* main\_generator() yields 0 and then uses yield from sub\_generator() to delegate the generation of values to sub\_generator().
* The yield from statement effectively "flattens" the nested structure, and the values yielded by sub\_generator() (1 and 2) become part of the output of main\_generator().
* Finally, main\_generator() yields 3.
* When we use a for loop to iterate over the main\_generator(), we get the output: 0 1 2 3.

**Chaining and Piping generators**

**Chaining Generators:**

Chaining generators involves combining multiple generators one after the other, allowing you to create a single continuous sequence of values from the output of multiple generator functions. This is achieved using the yield from statement to delegate the generation of values from one generator to another.

def generator1():  
 for i in range(1, 4):  
 yield i  
  
def generator2():  
 for j in range(4, 7):  
 yield j  
  
def chain\_generators():  
 yield from generator1()  
 yield from generator2()  
  
# Using the chained generator  
for num in chain\_generators():  
 print(num)

**Piping Generators:**

Piping generators involves passing the output of one generator as input to another generator, creating a processing pipeline of generators. Instead of using yield from, the output of the first generator is used in the second generator's logic, and so on.

def generator1():  
 for i in range(1, 4):  
 yield i  
  
def generator2(numbers):  
 for num in numbers:  
 yield num \*\* 2  
  
# Using the piped generator  
numbers\_generator = generator1()  
squared\_numbers\_generator = generator2(numbers\_generator)  
  
for num in squared\_numbers\_generator:  
 print(num)

*output:*

1  
4  
9

**Real World Usecases of Generators**

**Large Data Processing:**When dealing with large datasets that do not fit entirely into memory, generators can efficiently process data in chunks, one piece at a time, without the need to store the entire dataset in memory.

**Infinite Sequences:** Generators are perfect for representing infinite sequences, such as generating an endless stream of random numbers, counting numbers, or iterating over an open file.

**Data Streaming:** Generators are commonly used in web scraping and data streaming applications, where data is fetched or received incrementally over a network.

**Parsing and Tokenization:** Generators are useful for parsing large files or streams, where they can process data line by line or token by token.

**Python Inbuilt Generators**

1. range(): The range() function in Python is a built-in generator that efficiently generates a sequence of numbers within a specified range.
2. enumerate(): The enumerate() function returns an iterator of index-value pairs, allowing you to loop over elements while keeping track of their indices.
3. zip(): The zip() function combines multiple iterables element-wise, creating an iterator of tuples containing elements from each iterable.
4. map(): The map() function applies a given function to each item in an iterable and returns an iterator with the results.
5. filter(): The filter() function creates an iterator that filters elements of an iterable based on a given function's condition.
6. os.walk(): The os.walk() function generates file names in a directory tree, efficiently walking through the directory and its subdirectories.
7. dict.items(), dict.keys(), dict.values(): These built-in methods return iterators for dictionary keys, values, and key-value pairs, respectively.
8. itertools: The itertools module in Python provides a collection of powerful iterators and generators for various combinatorial functions, such as permutations, combinations, and infinite iterators.