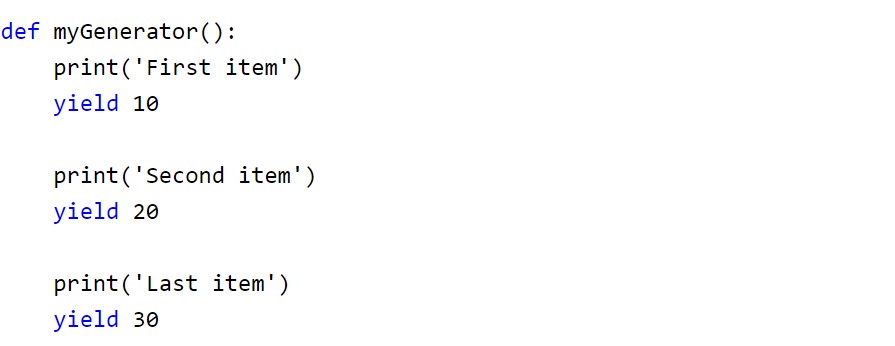
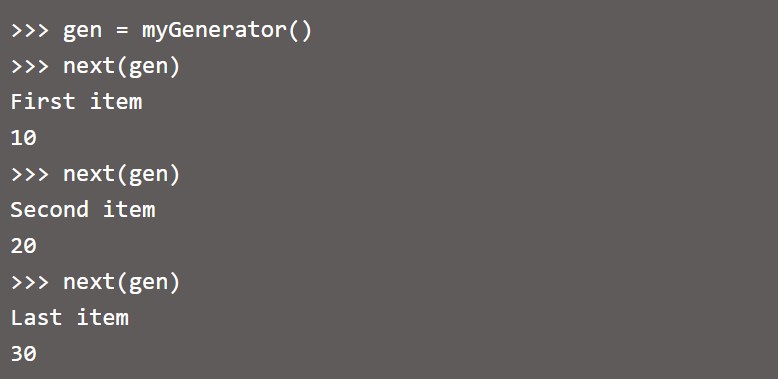
**Generators in Python**

Python provides a generator to create our own [iterator function](https://www.tutorialsteacher.com/python/python-iterator-iter). A generator is a special type of function which does not return a single value, instead it returns an iterator object with a sequence of values. In a generator function, a yield statement is used rather than a return statement. The following is a simple generator function.

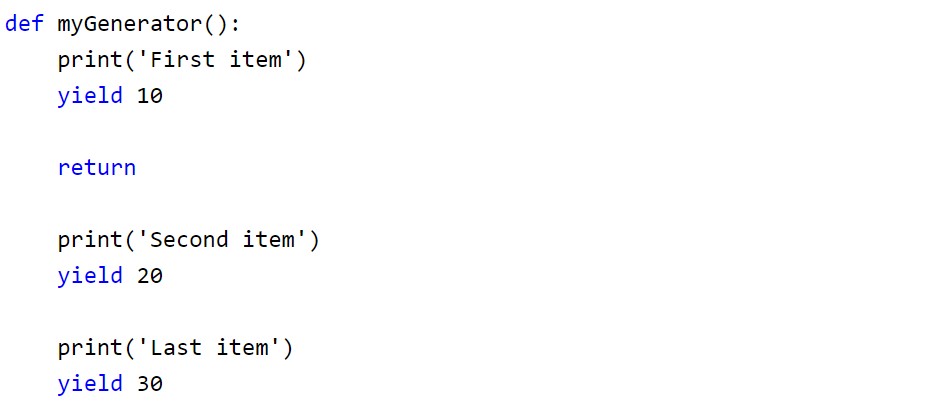


In the above example, myGenerator() is a generator function. It uses yield instead of return keyword. So, this will return the value against the yield keyword each time it is called. However, we need to create an iterator for this function, as shown below.

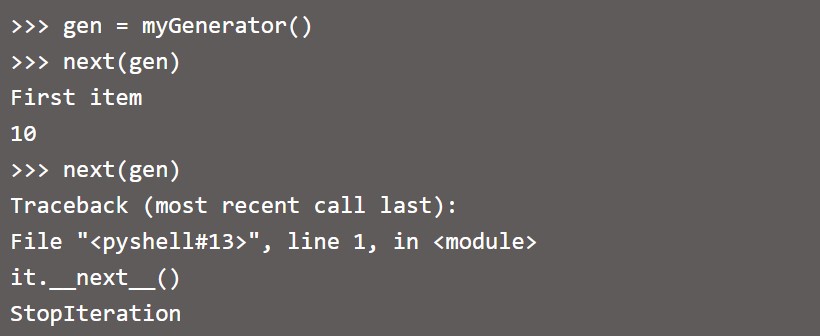


The generator function cannot include the return keyword. If it is included then it will terminate the function. The difference between yield and return is that yield returns a value and pauses the execution while maintaining the internal states, whereas the return statement returns a value and terminates the execution of the function.

The following generator function includes the return keyword.



Now, executing the above function as given below.



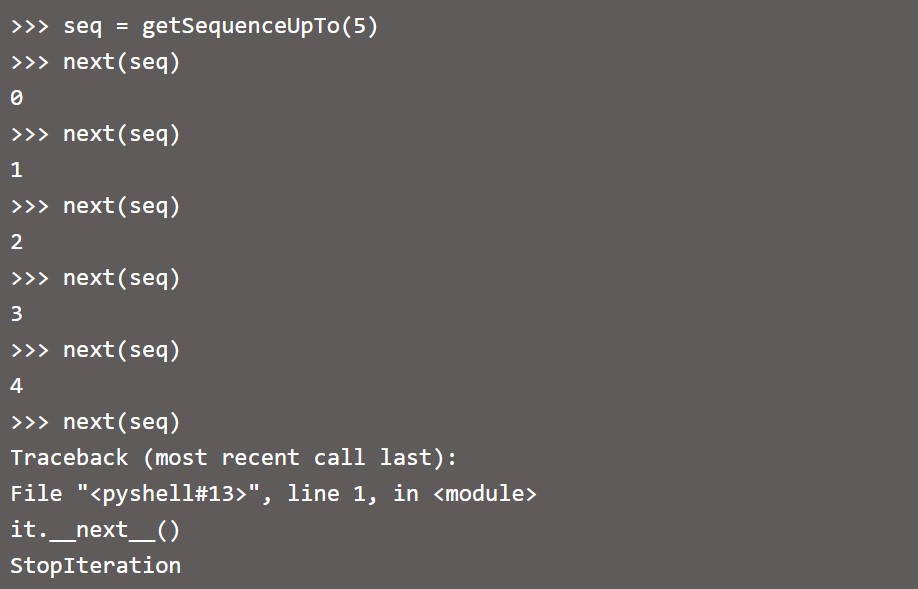
As we can see, the above generator stops executing after getting the first item because the return keyword is used after yielding the first item.

The generator function can also use the for loop.



As we can see above, the getSequenceUpTo function uses the yield keyword. The generator is called just like a normal function. However, its execution is paused on encountering the yield keyword. This sends the first value of the iterator stream to the calling environment. However, local variables and their states are saved internally.

The above generator function myGenerator can be called as below.

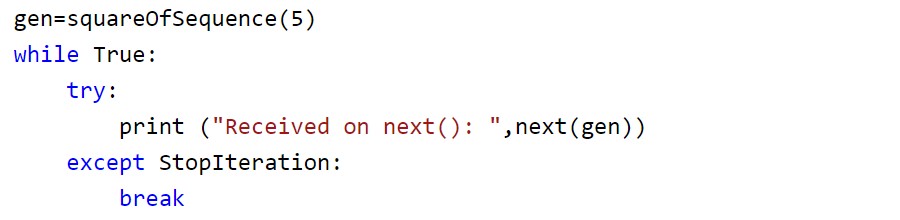


The function resumes when next() is issued to the iterator object. The function finally terminates when next() encounters the StopIteration error.

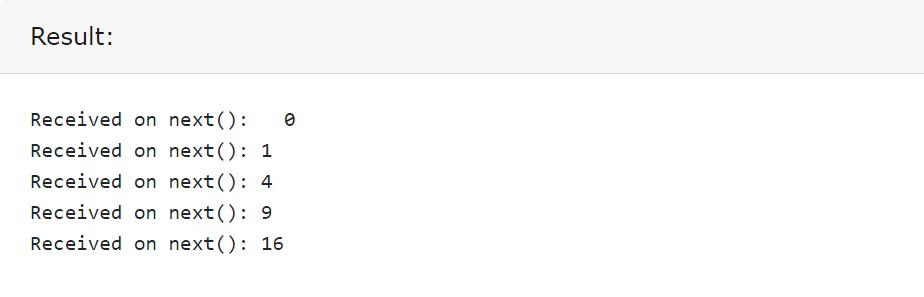
In the following example, the function squareOfSequence() acts as a generator. It yields the square of a number successively on every call of next().



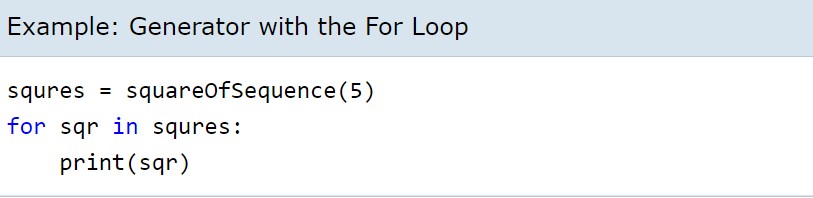
The following script shows how to call the above generator function.

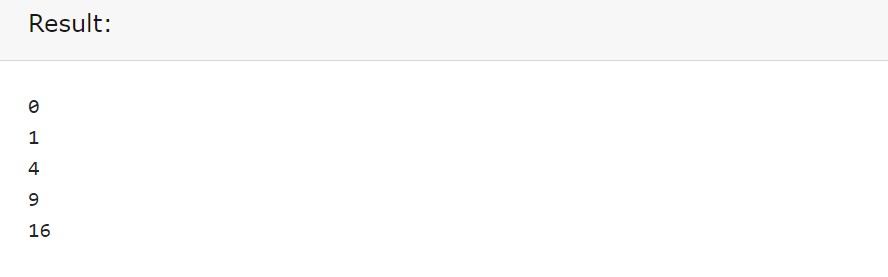


The above script uses the try..except block to handle the StopIteration error. It will break the while loop once it catches the StopIteration error.



We can use the for loop to traverse the elements over the generator. In this case, the next() function is called implicitly and StopIteration is also automatically taken care of.



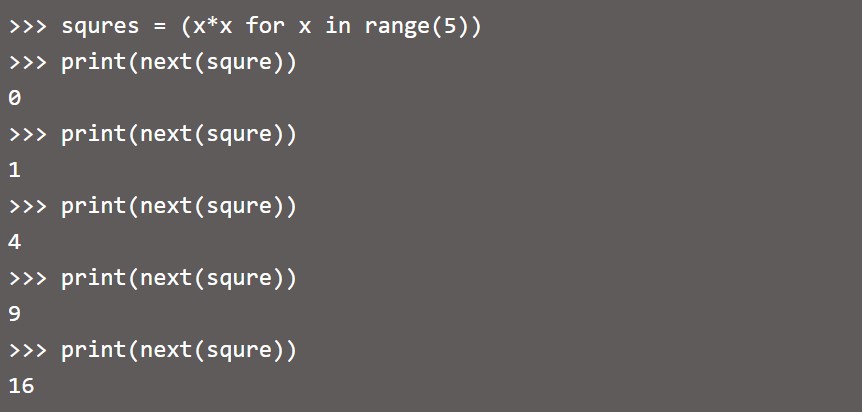


**Note:One of the advantages of the generator over the iterator is that elements are generated dynamically. Since the next item is generated only after the first is consumed, it is more memory efficient than the iterator.**

## 

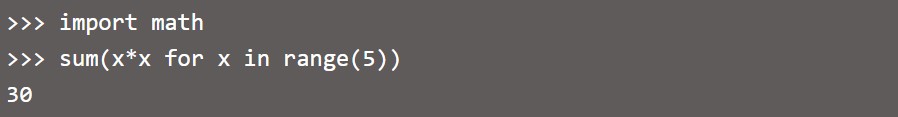
## **Generator Expression**

Python also provides a generator expression which is a shorter way of defining simple generator functions. Generator expression is an anonymous generator. The following is a generator expression for the squareOfSequence() function.



In the above example, (x\*x for x in range(5)) is a generator expression. The first part of an expression is the yield value and the second part is the for loop with the collection.

The generator expression can also be passed in a function. It should be passed without parentheses, as shown below.



In the above example, a generator expression is passed without parentheses into the built-in function sum.

### 

### 

### **Benefits of Generators**

1. Simplified code

Generators simplify code in a very elegant manner. These code simplification and elegance are even more evident in generator expressions, where a single line of code replaces an entire block of code.

1. Better performance

Generators work on lazy (on-demand) generation of values. This results in two advantages. First, lower memory consumption. However, this memory saving will work in our benefit if we use the generator only once. If we use the values several times, it may be worthwhile to generate them at once and keep them for later use.

The on-demand nature of generators also means we may not have to generate values that won't be used, and thus would have been wasted cycles if they were generated. This means your program can use only the values needed without having to wait until all of them have been generated.

### 

### 

### 

### 

### **When to use Generators**

Generators are an advanced tool present in Python. There are several programming cases where generators can increase efficiency. Some of these cases are:

* Processing large amounts of data: generators provide calculation on-demand, also called lazy evaluation. This technique is used in stream processing.
* Piping: stacked generators can be used as pipes, in a manner similar to Unix pipes.
* Concurrency: generators can be used to generate (simulate) concurrency.

### 

### 

### 

### 

### 

### **Conclusion**

Generators are a type of function that generate a sequence of values. As such they can act in a similar manner to iterators.These aspects are even more evident in generator expressions, where one line of code can summarize a sequence of statements.

Generators' working capacity has been improved with new methods, such as send(), and enhanced statements, such as yield from.

As a result of these properties, generators have many useful applications, such as generating pipes, concurrent programming, and helping in creating streams from large amounts of data.

As a consequence of these improvements, Python is becoming more and more the language of choice in data science.