**When to use yield instead of return in Python?**

The yield statement suspends a function’s execution and sends a value back to the caller, but retains enough state to enable the function to resume where it left off. When the function resumes, it continues execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list.

Let’s see with an example:

* Python

|  |
| --- |
| # A Simple Python program to demonstrate working  # of yield    # A generator function that yields 1 for the first time,  # 2 second time and 3 third time      **def** simpleGeneratorFun():  **yield** 1  **yield** 2  **yield** 3      # Driver code to check above generator function  **for** value **in** simpleGeneratorFun():  **print**(value) |

**Output:**

1

2

3

**Return** sends a specified value back to its caller whereas **Yield** can produce a sequence of values. We should use yield when we want to iterate over a sequence, but don’t want to store the entire sequence in memory. Yield is used in Python **generators**. A generator function is defined just like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

* Python

|  |
| --- |
| # A Python program to generate squares from 1  # to 100 using yield and therefore generator    # An infinite generator function that prints  # next square number. It starts with 1      **def** nextSquare():      i **=** 1        # An Infinite loop to generate squares  **while** True:  **yield** i**\***i          i **+=** 1  # Next execution resumes          # from this point      # Driver code to test above generator  # function  **for** num **in** nextSquare():  **if** num > 100:  **break**  **print**(num) |

**Output:**

1

4

9

16

25

36

49

64

81

100

# Generators in Python

 [Read](javascript:void(0))

 [Discuss(20+)](javascript:void(0))

 [Courses](javascript:void(0))

 [Practice](javascript:void(0))

A Generator in Python is a function that returns an iterator using the Yield keyword. In this article, we will discuss how the generator function works in Python.

## **Generator Function in Python**

A generator function in Python is defined like a normal function, but whenever it needs to generate a value, it does so with the [yield keyword](https://www.geeksforgeeks.org/python-yield-keyword/) rather than return. If the body of a def contains yield, the function automatically becomes a Python generator function.

### Create a Generator in Python

In Python, we can create a generator function by simply using the def keyword and the yield keyword. The generator has the following syntax in [Python](https://www.geeksforgeeks.org/python-programming-language/):

def function\_name():  
 yield statement

**Example:**

In this example, we will create a simple generator that will yield three integers. Then we will print these integers by using Python [for loop](https://www.geeksforgeeks.org/python-for-loops/).

* Python3

|  |
| --- |
| # A generator function that yields 1 for first time,  # 2 second time and 3 third time  **def** simpleGeneratorFun():  **yield** 1  **yield** 2  **yield** 3    # Driver code to check above generator function  **for** value **in** simpleGeneratorFun():  **print**(value) |

**Output:**

1  
2  
3

## **Generator Object**

Python Generator functions return a generator object that is iterable, i.e., can be used as an [Iterator](https://www.geeksforgeeks.org/iterators-in-python/). Generator objects are used either by calling the next method of the generator object or using the generator object in a “for in” loop.

**Example:**

In this example, we will create a simple generator function in Python to generate objects using the [next() function](https://www.geeksforgeeks.org/python-next-method/).

* Python3

|  |
| --- |
| # A Python program to demonstrate use of  # generator object with next()    # A generator function  **def** simpleGeneratorFun():  **yield** 1  **yield** 2  **yield** 3    # x is a generator object  x **=** simpleGeneratorFun()    # Iterating over the generator object using next    # In Python 3, \_\_next\_\_()  print(next(x))  **print**(next(x))  print(next(x)) |

**Output:**

1  
2  
3

**Example:**

In this example, we will create two generators for Fibonacci Numbers, first a simple generator and second generator using a [for loop](https://www.geeksforgeeks.org/python-for-loops/).

* Python3

|  |
| --- |
| # A simple generator for Fibonacci Numbers  **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)    # Iterating over the generator object using next  # In Python 3, \_\_next\_\_()  **print**(next(x))  **print**(next(x))  print(next(x))  **print**(next(x))  **print**(next(x))    # Iterating over the generator object using for  # in loop.  print("\nUsing for in loop")  **for** i **in** fib(5):      print(i) |

**Output:**

0  
1  
1  
2  
3  
  
Using for in loop  
0  
1  
1  
2  
3

## Python Generator Expression

In Python, generator expression is another way of writing the generator function. It uses the Python [list comprehension](https://www.geeksforgeeks.org/python-list-comprehension/) technique but instead of storing the elements in a list in memory, it creates generator objects.

### Generator Expression Syntax

The generator expression in Python has the following Syntax:

(expression for item in iterable)

**Example:**

In this example, we will create a generator object that will print the multiples of 5 between the range of 0 to 5 which are also divisible by 2.

* Python3

|  |
| --- |
| # generator expression  generator\_exp **=** (i **\*** 5 **for** i **in** range(5) **if** i**%**2**==**0)    **for** i **in** generator\_exp:      print(i) |

**Output:**

0  
10  
20

### **Applications of Generators in Python**

Suppose we create a stream of Fibonacci numbers, adopting the generator approach makes it trivial; we just have to call next(x) to get the next Fibonacci number without bothering about where or when the stream of numbers ends. A more practical type of stream processing is handling large data files such as log files. Generators provide a space-efficient method for such data processing as only parts of the file are handled at one given point in time. We can also use Iterators for these purposes, but Generator provides a quick way (We don’t need to write \_\_next\_\_ and \_\_iter\_\_ methods here).

# Python lambda

n Python, an anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.

## **Python lambda Syntax:**

lambda arguments : expression

## Python lambda Example:

* Python3

|  |
| --- |
| calc **=** **lambda** num: "Even number" **if** num **%** 2 **==** 0 **else** "Odd number"    print(calc(20)) |

**Output:**

Even number

## Python lambda properties:

* This function can have any number of arguments but only one expression, which is evaluated and returned.
* One is free to use lambda functions wherever function objects are required.
* You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
* It has various uses in particular fields of programming, besides other types of expressions in functions.

### **Example 1: Program to demonstrate return type of Python lambda keyword**

* Python3

|  |
| --- |
| string **=** 'GeeksforGeeks'    # lambda returns a function object  print(**lambda** string: string) |

**Output**

<function <lambda> at 0x7fd7517ade18>

**Explanation:** In this above example, the lambda is not being called by the print function, but simply returning the function object and the memory location where it is stored. So, to make the print to print the string first, we need to call the lambda so that the string will get pass the print.

### **Example 2:** Invoking lambda return value to perform various operations

Here we have passed various types of arguments into the different lambda functions and printed the result generated from the lambda function calls.

* Python3

|  |
| --- |
| filter\_nums **=** **lambda** s: ''.join([ch **for** ch **in** s **if** **not** ch.isdigit()])  print("filter\_nums():", filter\_nums("Geeks101"))    do\_exclaim **=** **lambda** s: s **+** '!'  print("do\_exclaim():", do\_exclaim("I am tired"))    find\_sum **=** **lambda** n: sum([int(x) **for** x **in** str(n)])  print("find\_sum():", find\_sum(101)) |

**Output:**

filter\_nums(): Geeks

do\_exclaim(): I am tired!

find\_sum(): 2

### **Example 3:** Difference between lambda and normal function call

The main difference between **lambda** function and other functions defined using **def** keyword is that, we cannot use multiple statements inside a lambda function and allowed statements are also very limited inside lambda statements. Using lambda functions to do complex operations may affect the readability of the code.

* Python3

|  |
| --- |
| **def** cube(y):      print(f"Finding cube of number:{y}")  **return** y **\*** y **\*** y      lambda\_cube **=** **lambda** num: num **\*\*** 3    # invoking simple function  **print**("invoking function defined with def keyword:")  print(cube(30))  # invoking lambda function  print("invoking lambda function:", lambda\_cube(30)) |

**Output:**

invoking function defined with def keyword:

Finding cube of number:30

27000

invoking lambda function: 27000

### **Example 4:** The lambda function gets more helpful when used inside a function.

We can use lambda function inside map(), filter(), sorted() and many other functions. Here, we have demonstrated how to use lambda function inside some of the most common Python functions.

* Python3

|  |
| --- |
| l **=** ["1", "2", "9", "0", "-1", "-2"]  # sort list[str] numerically using sorted()  # and custom sorting key using lambda  print("Sorted numerically:",        sorted(l, key**=lambda** x: int(x)))    # filter positive even numbers  # using filter() and lambda function  **print**("Filtered positive even numbers:",        list(filter(**lambda** x: **not** (int(x) **%** 2 **==** 0 **and** int(x) > 0), l)))    # added 10 to each item after type and  # casting to int, then convert items to string again  **print**("Operation on each item using lambda and map()",        list(map(**lambda** x: str(int(x) **+** 10), l))) |

**Output**

Sorted numerically: ['-2', '-1', '0', '1', '2', '9']

Filtered positive even numbers: ['1', '9', '0', '-1', '-2']

Operation on each item using lambda and map() ['11', '12', '19', '10', '9', '8']

#### using mylist:

In this example, we define a lambda function that takes an argument x and adds 10 to it. We then use the map() function to apply the lambda function to each element in the list my\_list. Finally, we convert the result to a list and print it.

Here’s another example that uses a lambda function to filter out even numbers from a list:

* Python3

|  |
| --- |
| # Example list  my\_list **=** [1, 2, 3, 4, 5]    # Use lambda to filter out even numbers from the list  new\_list **=** list(filter(**lambda** x: x **%** 2 !**=** 0, my\_list))    # Print the new list  print(new\_list) |

**Output**

[1, 3, 5]

Approach:

Define a list ‘my\_list’ with some numbers.  
Use lambda function with filter to check whether a number in the list is even or not.  
Convert the filter object into a list using the list() function and store it in a new\_list.  
Print the new list with odd numbers.  
Time Complexity:  
The time complexity of the filter function is O(n) where n is the number of elements in the list. The lambda function does not affect the time complexity, as it is a simple check that takes constant time.

Space Complexity:  
The space complexity of this code is O(n) because it creates a new list that contains only odd numbers from the original list. The original list is not modified, so it remains the same size.