Python Functions

In this article, you'll learn about functions, what a function is, the syntax, components, and types of functions. Also, you'll learn to create a function in Python.

Video: Introduction to Python Functions

What is a function in Python?

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function

```
def function_name(parameters):

        """docstring"""

        statement(s)
```

Above shown is a function definition that consists of the following components.

1. Keyword `def` that marks the start of the function header.

2. A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
7. An optional `return` statement to return a value from the function.

Example of a function

```python
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program, or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')
Hello, Paul. Good morning!
```

Try running the above code in the Python program with the function definition to see the output.

```python
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

greet('Paul')
```
Run Code

Note: In python, the function definition should always be present before the function call. Otherwise, we will get an error. For example,

```
# function call
greet('Paul')

# function definition
def greet(name):
    """
    This function greets to
    the person passed in as
    a parameter
    """
    print("Hello, " + name + ". Good morning!")

# Error: name 'greet' is not defined
```

Docstrings

The first string after the function header is called the docstring and is short for documentation string. It is briefly used to explain what a function does.

Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

In the above example, we have a docstring immediately below the function header. We generally use triple quotes so that docstring can extend up to multiple lines. This string is available to us as the `__doc__` attribute of the function.
For example:
Try running the following into the Python shell to see the output.

```
>>> print(greet.__doc__)

    This function greets to
    the person passed in as
    a parameter
```

To learn more about docstrings in Python, visit Python Docstrings.

The return statement

The `return` statement is used to exit a function and go back to the place from where it was called.
Syntax of return

```
return [expression_list]
```

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the `return` statement itself is not present inside a function, then the function will return the `None` object.

For example:

```
>>> print(greet("May"))
Hello, May. Good morning!
None
```

Here, `None` is the returned value since `greet()` directly prints the name and no `return` statement is used.

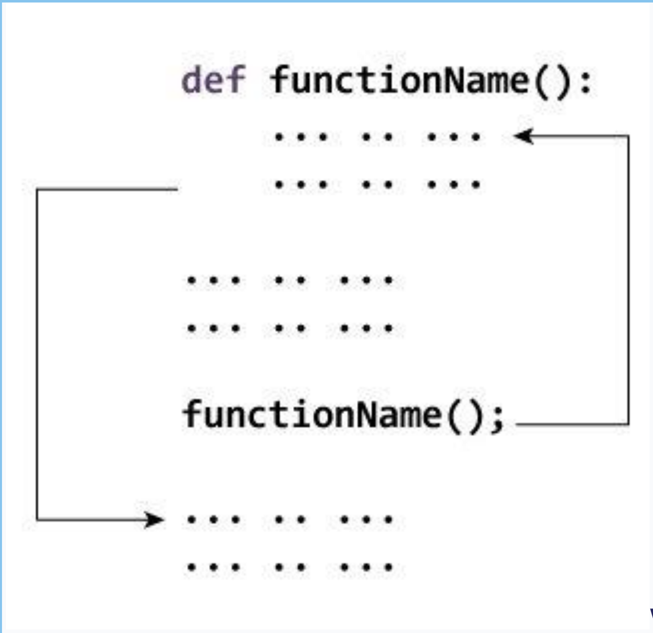
Example of return

```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num


print(absolute_value(2))

print(absolute_value(-4))
```
Run Code

Output

```
2
4
```

How Function works in Python?



Working of functions in Python

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function are not visible from outside the function. Hence, they have a local scope.

The lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Here is an example to illustrate the scope of a variable inside a function.

```
def my_func():
	x = 10
	print("Value inside function:",x)
```

```
x = 20
my_func()
print("Value outside function:",x)
```
Run Code

Output

```
Value inside function: 10
Value outside function: 20
```

Here, we can see that the value of `x` is 20 initially. Even though the function `my_func()` changed the value of `x` to 10, it did not affect the value outside the function.

This is because the variable `x` inside the function is different (local to the function) from the one outside. Although they have the same names, they are two different variables with different scopes.

On the other hand, variables outside of the function are visible from inside. They have a global scope.

We can read these values from inside the function but cannot change (write) them. In order to modify the value of variables outside the function, they must be declared as global variables using the keyword `global`.

Types of Functions

Basically, we can divide functions into the following two types:

1. Built-in functions - Functions that are built into Python.
2. User-defined functions - Functions defined by the users themselves.

3. Python Function Arguments

4. In Python, you can define a function that takes variable number of arguments. In this article, you will learn to define such functions using default, keyword and arbitrary arguments.

5. Video: Python Function Arguments: Positional, Keywords and Default

6. Arguments

7. In the user-defined function topic, we learned about defining a function and calling it. Otherwise, the function call will result in an error. Here is an example.

```
def greet(name, msg):
    """This function greets to
    the person with the provided message"""
    print("Hello", name + ', ' + msg)

greet("Monica", "Good morning!")
```

Run Code

15. **Output**

```
Hello Monica, Good morning!
```

17. Here, the function `greet()` has two parameters.

18. Since we have called this function with two arguments, it runs smoothly and we do not get any error.

19. If we call it with a different number of arguments, the interpreter will show an error message. Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("Monica")    # only one argument
TypeError: greet() missing 1 required positional argument: 'msg'
>>> greet()    # no arguments
```

```
TypeError: greet() missing 2 required positional arguments: 'name' and
'msg'
```

Variable Function Arguments

Up until now, functions had a fixed number of arguments. In Python, there are other ways to define a function that can take variable number of arguments.

Three different forms of this type are described below.

Python Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=). Here is an example.

```python
def greet(name, msg="Good morning!"):
    """
    This function greets to
    the person with the
    provided message.

    If the message is not provided,
    it defaults to "Good
    morning!"
    """

    print("Hello", name + ', ' + msg)


greet("Kate")
greet("Bruce", "How do you do?")
```
Run Code

Output

```
Hello Kate, Good morning!
Hello Bruce, How do you do?
```

51. In this function, the parameter `name` does not have a default value and is required (mandatory) during a call.

52. On the other hand, the parameter `msg` has a default value of `"Good morning!"`. So, it is optional during a call. If a value is provided, it will overwrite the default value.

53. Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

54. This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
55.def greet(msg = "Good morning!", name):
```

56. We would get an error as:

```
57.SyntaxError: non-default argument follows default argument
```

58.

---

59. Python Keyword Arguments

60. When we call a function with some values, these values get assigned to the arguments according to their position.

61. For example, in the above function `greet()`, when we called it as `greet("Bruce", "How do you do?")`, the value `"Bruce"` gets assigned to the argument `name` and similarly `"How do you do?"` to `msg`.

62. Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed. Following calls to the above function are all valid and produce the same result.

```
63.# 2 keyword arguments
64.greet(name = "Bruce",msg = "How do you do?")
65.
```

```
# 2 keyword arguments (out of order)
greet(msg = "How do you do?",name = "Bruce")

1 positional, 1 keyword argument
greet("Bruce", msg = "How do you do?")
```

71. As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

72. Having a positional argument after keyword arguments will result in errors. For example, the function call as follows:

```
greet(name="Bruce","How do you do?")
```

74. Will result in an error:

```
SyntaxError: non-keyword arg after keyword arg
```

76.

---

77. Python Arbitrary Arguments

78. Sometimes, we do not know in advance the number of arguments that will be passed into a function. Python allows us to handle this kind of situation through function calls with an arbitrary number of arguments.

79. In the function definition, we use an asterisk (*) before the parameter name to denote this kind of argument. Here is an example.

```
def greet(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello", name)


greet("Monica", "Luke", "Steve", "John")
```

90. Run Code

91. **Output**

```
92.Hello Monica
93.Hello Luke
94.Hello Steve
95.Hello John
```

96. Here, we have called the function with multiple arguments. These arguments get wrapped up into a tuple before being passed into the function. Inside the function, we use a `for` loop to retrieve all the arguments back.

Python Recursion

In this tutorial, you will learn to create a recursive function (a function that calls itself).

What is recursion?

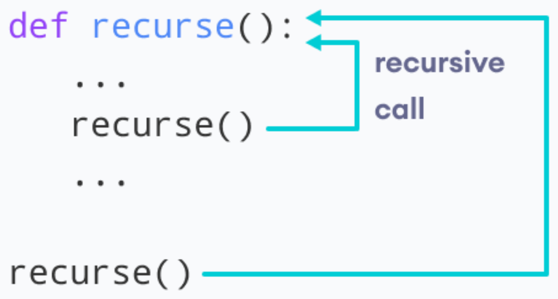
Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

Python Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

The following image shows the working of a recursive function called `recurse`.



```
def recurse():
    ...
    recurse()
    ...

recurse()
```

Recursive Function in Python

Following is an example of a recursive function to find the factorial of an integer.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is `1*2*3*4*5*6 = 720`.

Example of a recursive function

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))
```

Run Code

Output

```
The factorial of 3 is 6
```

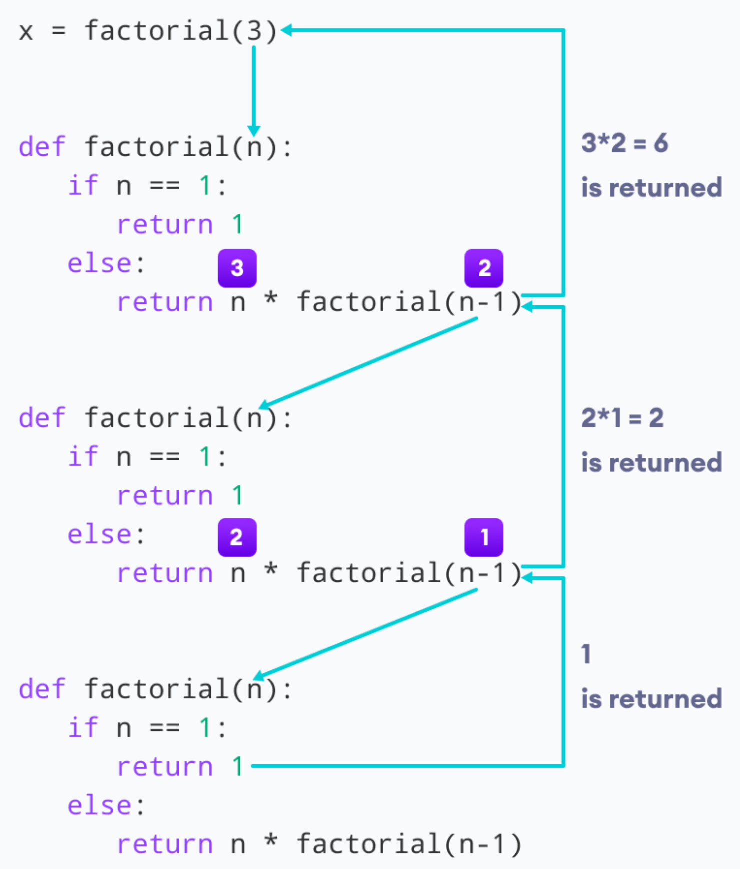
In the above example, `factorial()` is a recursive function as it calls itself.

When we call this function with a positive integer, it will recursively call itself by decreasing the number.

Each function multiplies the number with the factorial of the number below it until it is equal to one. This recursive call can be explained in the following steps.

```
factorial(3)          # 1st call with 3

3 * factorial(2)      # 2nd call with 2

3 * 2 * factorial(1)  # 3rd call with 1

3 * 2 * 1             # return from 3rd call as number=1

3 * 2                 # return from 2nd call

6                     # return from 1st call
```

Let's look at an image that shows a step-by-step process of what is going on:


```
x = factorial(3)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Working of a recursive factorial function

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

The Python interpreter limits the depths of recursion to help avoid infinite recursions, resulting in stack overflows.

By default, the maximum depth of recursion is `1000`. If the limit is crossed, it results in `RecursionError`. Let's look at one such condition.

```python
def recursor():
    recursor()
recursor()
```

Output

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  File "<string>", line 2, in a
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

3. Recursive functions are hard to debug.

4. Python Anonymous/Lambda Function

5. In this article, you'll learn about the anonymous function, also known as lambda functions. You'll learn what they are, their syntax and how to use them (with examples).

6. Video: Python Lambda

7. What are lambda functions in Python?

8. In Python, an anonymous function is a function that is defined without a name.

9. While normal functions are defined using the `def` keyword in Python, anonymous functions are defined using the `lambda` keyword.

10. Hence, anonymous functions are also called lambda functions.

11.

---

12. How to use lambda Functions in Python?

13. A lambda function in python has the following syntax.

14. Syntax of Lambda Function in python

```
15.lambda arguments: expression
```

16. Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned. Lambda functions can be used wherever function objects are required.

17.

---

18. Example of Lambda Function in python

19. Here is an example of lambda function that doubles the input value.

```
20.# Program to show the use of lambda functions
```

```
21.double = lambda x: x * 2
22.
23.print(double(5))
```
24. Run Code

25. **Output**

```
26.10
```

27. In the above program, `lambda x: x * 2` is the lambda function. Here `x` is the argument and `x * 2` is the expression that gets evaluated and returned.

28. This function has no name. It returns a function object which is assigned to the identifier `double`. We can now call it as a normal function. The statement

```
29.double = lambda x: x * 2
```

30. is nearly the same as:

```
31.def double(x):
32.   return x * 2
```

33.

---

34. Use of Lambda Function in python

35. We use lambda functions when we require a nameless function for a short period of time.

36. In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). Lambda functions are used along with built-in functions like `filter()`, `map()` etc.

37. Example use with filter()

38. The `filter()` function in Python takes in a function and a list as arguments.

39. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to `True`.

40. Here is an example use of `filter()` function to filter out only even numbers from a list.

```
# Program to filter out only the even items from a list
my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(filter(lambda x: (x%2 == 0) , my_list))

print(new_list)
```

47. Run Code

48. **Output**

```
[4, 6, 8, 12]
```

50. Example use with map()

51. The `map()` function in Python takes in a function and a list.

52. The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

53. Here is an example use of `map()` function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]

new_list = list(map(lambda x: x * 2 , my_list))

print(new_list)
```

61. Run Code

62. **Output**

```
[2, 10, 8, 12, 16, 22, 6, 24]
```

Python Global, Local and Nonlocal variables

In this tutorial, you'll learn about Python Global variables, Local variables, Nonlocal variables and where to use them.

Video: Python Local and Global Variables

Global Variables

In Python, a variable declared outside of the function or in global scope is known as a global variable. This means that a global variable can be accessed inside or outside of the function.

Let's see an example of how a global variable is created in Python.

Example 1: Create a Global Variable

```
x = "global"

def foo():
    print("x inside:", x)


foo()
print("x outside:", x)
```
Run Code

Output

```
x inside: global
x outside: global
```

In the above code, we created `x` as a global variable and defined a `foo()` to print the global variable `x`. Finally, we call the `foo()` which will print the value of `x`.

What if you want to change the value of `x` inside a function?

```
x = "global"

def foo():
    x = x * 2
    print(x)

foo()
```
Run Code

Output

```
UnboundLocalError: local variable 'x' referenced before assignment
```

The output shows an error because Python treats `x` as a local variable and `x` is also not defined inside `foo()`.

To make this work, we use the `global` keyword. Visit Python Global Keyword to learn more.

Local Variables

A variable declared inside the function's body or in the local scope is known as a local variable.

Example 2: Accessing local variable outside the scope

```
def foo():
    y = "local"


foo()
```

```
print(y)
```
Run Code

Output

```
NameError: name 'y' is not defined
```

The output shows an error because we are trying to access a local variable `y` in a global scope whereas the local variable only works inside `foo()` or local scope.

Let's see an example on how a local variable is created in Python.

Example 3: Create a Local Variable

Normally, we declare a variable inside the function to create a local variable.

```
def foo():
    y = "local"
    print(y)

foo()
```
Run Code

Output

```
local
```

Let's take a look at the earlier problem where `x` was a global variable and we wanted to modify `x` inside `foo()`.

Global and local variables

Here, we will show how to use global variables and local variables in the same code.

Example 4: Using Global and Local variables in the same code

```python
x = "global "

def foo():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

foo()
```

Run Code

Output

```
global global
local
```

In the above code, we declare `x` as a global and `y` as a local variable in the `foo()`. Then, we use multiplication operator `*` to modify the global variable `x` and we print both `x` and `y`.
After calling the `foo()`, the value of `x` becomes `global global` because we used the `x * 2` to print two times `global`. After that, we print the value of local variable `y` i.e `local`.

Example 5: Global variable and Local variable with same name

```python
x = 5
```

```
def foo():
    x = 10
    print("local x:", x)


foo()
print("global x:", x)
```
Run Code

Output

```

local x: 10
global x: 5
```

In the above code, we used the same name `x` for both global variable and local variable. We get a different result when we print the same variable because the variable is declared in both scopes, i.e. the local scope inside `foo()` and global scope outside `foo()`.
When we print the variable inside `foo()` it outputs `local x: 10`. This is called the local scope of the variable.
Similarly, when we print the variable outside the `foo()`, it outputs `global x: 5`. This is called the global scope of the variable.

Nonlocal Variables

Nonlocal variables are used in nested functions whose local scope is not defined. This means that the variable can be neither in the local nor the global scope.

Let's see an example of how a nonlocal variable is used in Python.

We use `nonlocal` keywords to create nonlocal variables.

Example 6: Create a nonlocal variable

```
def outer():
    x = "local"

    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner:", x)

    inner()
    print("outer:", x)


outer()
```
Run Code

Output

```
inner: nonlocal
outer: nonlocal
```

In the above code, there is a nested `inner()` function. We use `nonlocal` keywords to create a nonlocal variable. The `inner()` function is defined in the scope of another function `outer()`.

Note : If we change the value of a nonlocal variable, the changes appear in the local variable.

Python Global Keyword

In this article, you'll learn about the global keyword, global variable and when to use global keywords.

Before reading this article, make sure you have got some basics of Python Global, Local and Nonlocal Variables.

What is the global keyword

In Python, `global` keyword allows you to modify the variable outside of the current scope. It is used to create a global variable and make changes to the variable in a local context.

Rules of global Keyword

The basic rules for `global` keyword in Python are:

- When we create a variable inside a function, it is local by default.
- When we define a variable outside of a function, it is global by default. You don't have to use `global` keyword.
- We use `global` keyword to read and write a global variable inside a function.
- Use of `global` keyword outside a function has no effect.

Use of global Keyword

Let's take an example.

Example 1: Accessing global Variable From Inside a Function

```
c = 1 # global variable

def add():
    print(c)

add()
```
Run Code

When we run the above program, the output will be:

```
1
```

However, we may have some scenarios where we need to modify the global variable from inside a function.

Example 2: Modifying Global Variable From Inside the Function

```
c = 1 # global variable

def add():
    c = c + 2 # increment c by 2
    print(c)

add()
```
Run Code

When we run the above program, the output shows an error:

```
UnboundLocalError: local variable 'c' referenced before assignment
```

This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the `global` keyword.

Example 3: Changing Global Variable From Inside a Function using global

```
c = 0 # global variable

def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)

add()
print("In main:", c)
```
Run Code

When we run the above program, the output will be:

```
Inside add(): 2
In main: 2
```

In the above program, we define `c` as a global keyword inside the `add()` function.

Then, we increment the variable `c` by **2**, i.e `c = c + 2`. After that, we call the `add()` function. Finally, we print the global variable `c`.

As we can see, change also occurred on the global variable outside the function, `c = 2`.

Global Variables Across Python Modules

In Python, we create a single module `config.py` to hold global variables and share information across Python modules within the same program.
Here is how we can share global variables across the python modules.

Example 4: Share a global Variable Across Python Modules

Create a `config.py` file, to store global variables

```
a = 0
b = "empty"
```

Create a `update.py` file, to change global variables

```
import config

config.a = 10
config.b = "alphabet"
```

Create a `main.py` file, to test changes in value

```
import config
import update

print(config.a)
print(config.b)
```

When we run the `main.py` file, the output will be

```
10
alphabet
```

In the above, we have created three files: `config.py`, `update.py`, and `main.py`. The module `config.py` stores global variables of `a` and `b`. In the `update.py` file, we import the `config.py` module and modify the values of `a` and `b`. Similarly, in the `main.py` file, we import both `config.py` and `update.py` module. Finally, we print and test the values of global variables whether they are changed or not.

Global in Nested Functions

Here is how you can use a global variable in nested function.

Example 5: Using a Global Variable in Nested Function

```
def foo():
    x = 20

    def bar():
        global x
        x = 25

    print("Before calling bar: ", x)
    print("Calling bar now")
    bar()
    print("After calling bar: ", x)

foo()
print("x in main: ", x)
```
Run Code

The output is :

```
Before calling bar: 20
Calling bar now
After calling bar: 20
x in main: 25
```

In the above program, we declared a global variable inside the nested function `bar()`. Inside `foo()` function, `x` has no effect of the global keyword. Before and after calling `bar()`, the variable `x` takes the value of local variable i.e `x = 20`. Outside of the `foo()` function, the variable `x` will take value defined in the `bar()` function i.e `x = 25`. This is because we have used `global` keyword in `x` to create global variable inside the `bar()` function (local scope).

If we make any changes inside the `bar()` function, the changes appear outside the local scope, i.e. `foo()`.