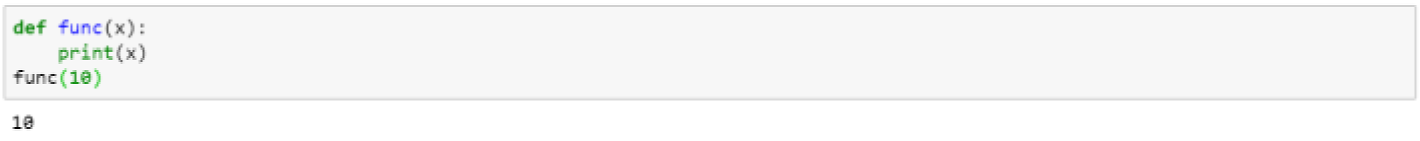
Functions in Python

CONTENTS

In Python, functions are “first-class citizens.” This means that they are on par with any other object (integers, strings, lists, modules, and so on). It can be dynamically created or destroyed, passed to other functions, returned as values, and so forth.

Defining function, calling function

Example:



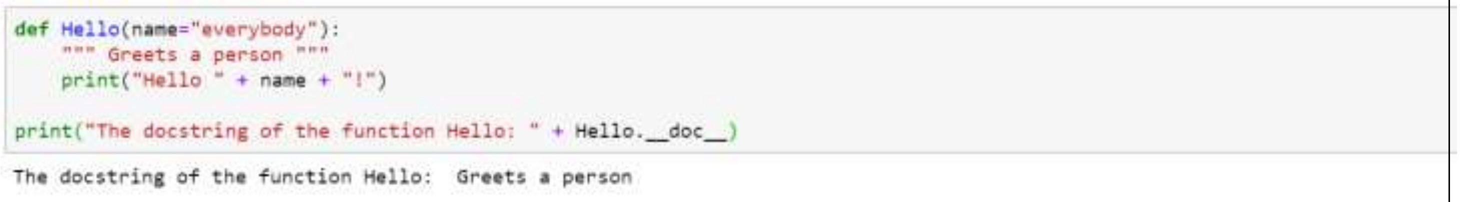
```
def func(x):
    print(x)
func(10)
```

```
10
```

Docstring

The first statement in the body of a function is usually a string, which can be accessed with function_name.__doc__. This statement is called Docstring.

Example:



```
def Hello(name="everybody"):
    """ Greets a person """
    print("Hello " + name + "!")

print("The docstring of the function Hello: " + Hello.__doc__)
```

```
The docstring of the function Hello:  Greets a person
```

Understanding function object using compile and exec functions

The compile() method returns a Python code object from the source (normal string, a byte string, or an AST object). The syntax of compile() is:

```
compile(source, filename, mode, flags=0, dont_inherit=False, optimize=-1)
```

The compile() method is used if the Python code is in string form or is an AST object, and you want to change it to a code object. The code object returned by the compile() method can later be called using methods like: exec() and eval() which will execute dynamically generated Python code.

compile() Parameters

- source - a normal string, a byte string, or an AST object
- filename - file from which the code was read. If it wasn't read from a file, you can give a name yourself
- mode - Either exec or eval or single.
    - eval - accepts only a single expression.
    - exec - It can take a code block that has Python statements, class and functions and so on.
    - single - if it consists of a single interactive statement
- flags (optional) and dont_inherit (optional) - controls which future statements affect the compilation of the source. Default Value: 0
- optimize (optional) - optimization level of the compiler. Default value -1.

Return Value from compile()

- The compile() method returns a Python code object.

The exec() method executes the dynamically created program, which is either a string or a code object.

The syntax of exec();

```
exec(object, globals, locals)
```

exec() Parameters

The exec() takes three parameters:

- **object** - Either a string or a code object
- **globals** (optional) - a dictionary
- **locals** (optional)- a mapping object. Dictionary is the standard and commonly used mapping type in Python.

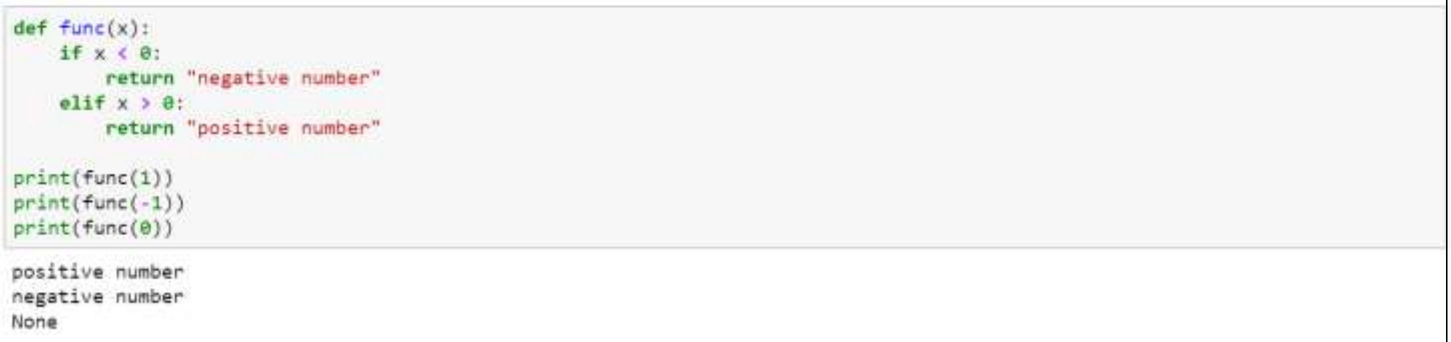
Return Value from exec()

- The exec() doesn't return any value, it returns None.

Return statement

Function bodies can contain one or more return statement. They can be situated anywhere in the function body. A return statement ends the execution of the function call and "returns" the result, i.e. the value of the expression following the return keyword, to the caller. If the return statement is without an expression, the special value None is returned. If there is no return statement in the function code, the function ends, when the control flow reaches the end of the function body and the value "None" will be returned.

Example:



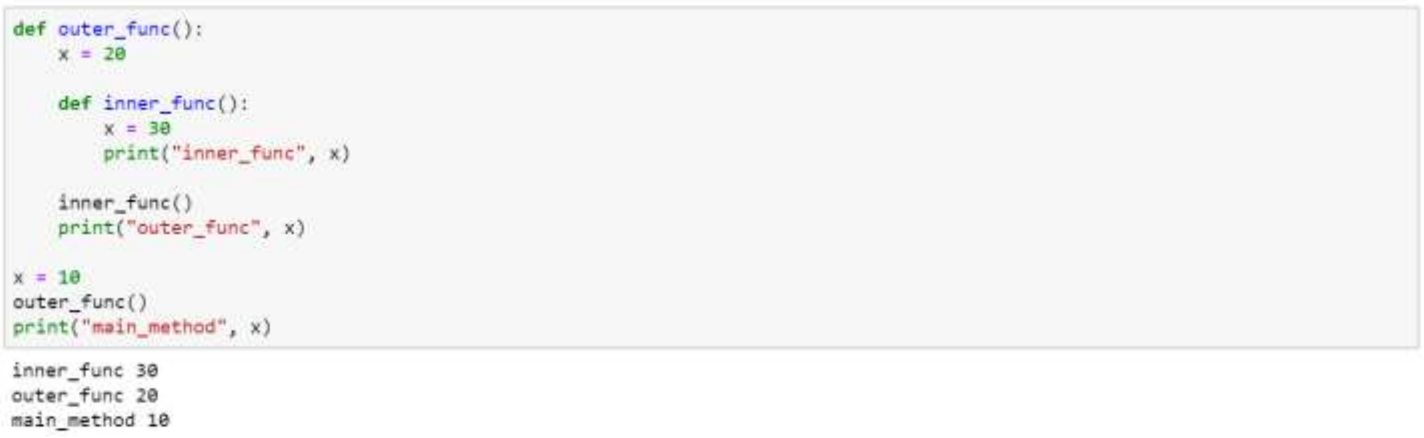
```
def func(x):
    if x < 0:
        return "negative number"
    elif x > 0:
        return "positive number"

print(func(1))
print(func(-1))
print(func(0))
```

```
positive number
negative number
None
```

Nested functions

A function which is defined inside another function is known as nested function. Nested functions are able to access variables of the enclosing scope. In Python, these non-local variables can be accessed only within their scope and not outside their scope. As we can see inner_func() can easily be accessed inside the outer_func body but not outside of it’s body. Hence, here, inner_func() is treated as nested Function.

Example



```
def outer_func():
    x = 20

    def inner_func():
        x = 30
        print("inner_func", x)

    inner_func()
    print("outer_func", x)

x = 10
outer_func()
print("main_method", x)
```

```
inner_func 30
outer_func 20
main_method 10
```

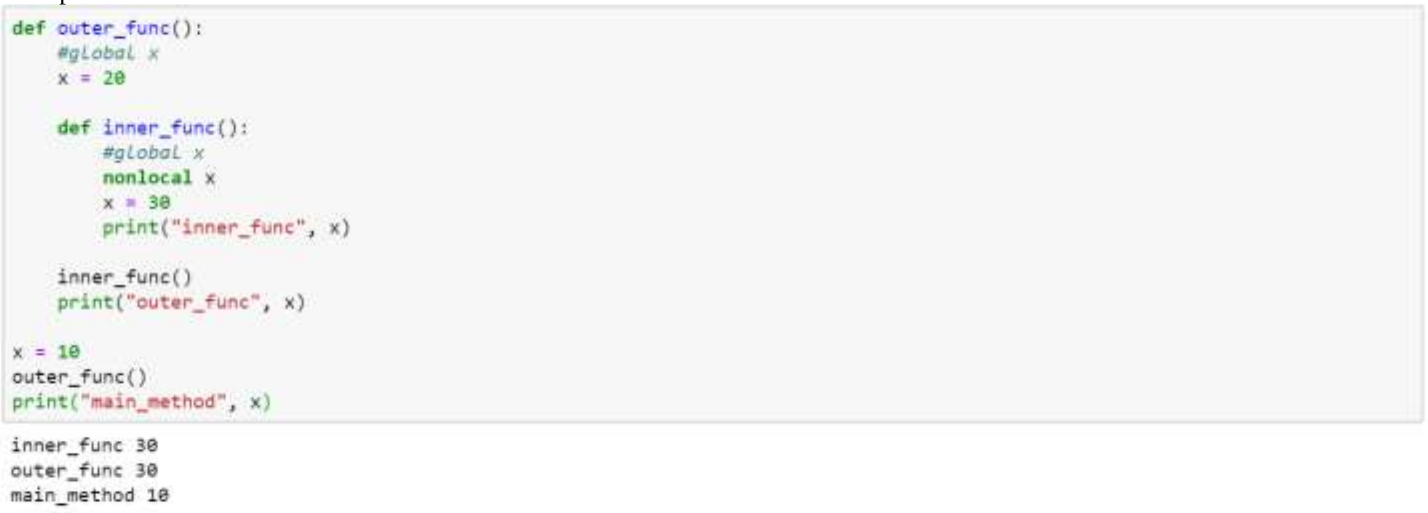
Scope of variables (globals, locals, nonlocal, built-in functions)

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

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

Nonlocal Variables: Nonlocal variable are used in nested function whose local scope is not defined. This means, the variable can be neither in the local nor the global scope.

Example:



```
def outer_func():
    #global x
    x = 20

    def inner_func():
        #global x
        nonlocal x
        x = 30
        print("inner_func", x)

    inner_func()
    print("outer_func", x)

x = 10
outer_func()
print("main_method", x)
```

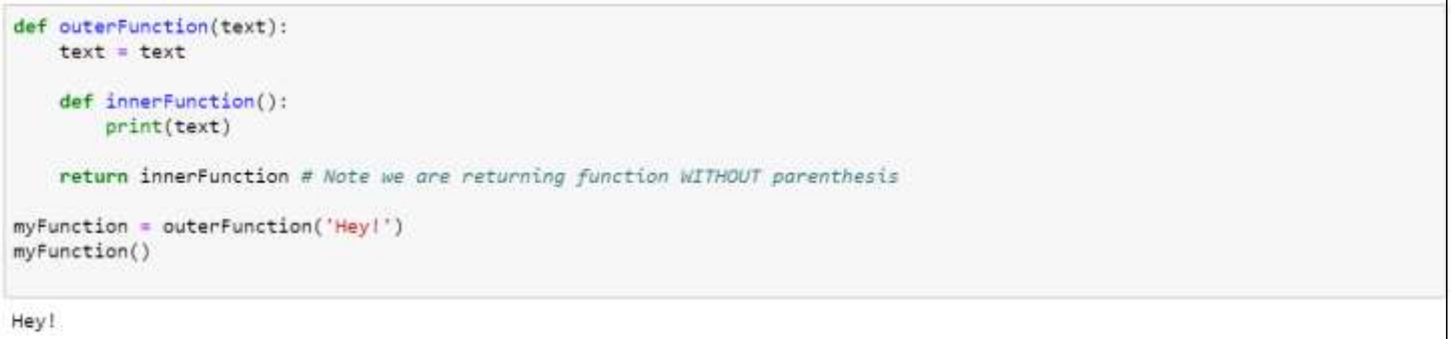
```
inner_func 30
outer_func 30
main_method 10
```

Closure

A Closure is a function object that remembers values in enclosing scopes even if they are not present in memory.

- It is a record that stores a function together with an environment: a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created.
- A closure—unlike a plain function—allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

Example:



```python
def outerFunction(text):
    text = text

    def innerFunction():
        print(text)

    return innerFunction # Note we are returning function WITHOUT parenthesis

myFunction = outerFunction('Hey!')
myFunction()
```

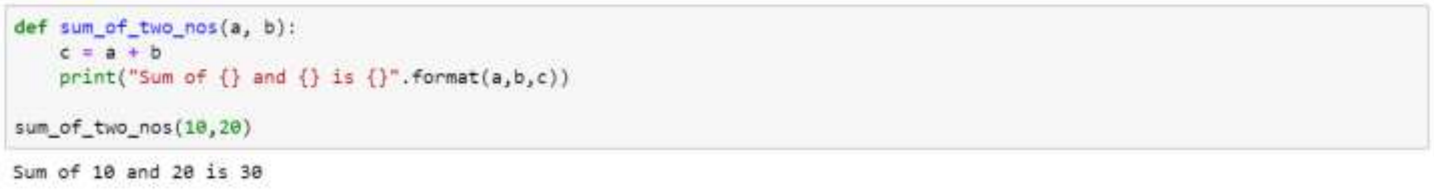
```
Hey!
```

Parameter passing mechanisms in Python

The parameter list consists of none or more parameters. Parameters are called arguments, if the function is called. The function body consists of indented statements. The function body gets executed every time the function is called. Parameter can be mandatory or optional. The optional parameters (zero or more) must follow the mandatory parameters.

Positional arguments

Example:



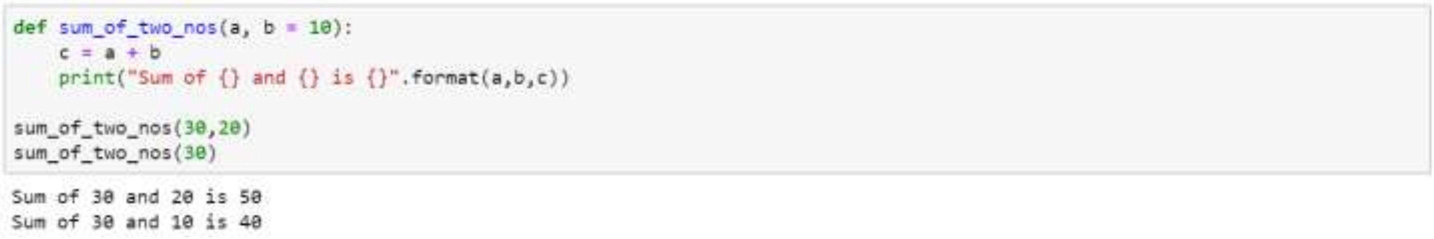
```python
def sum_of_two_nos(a, b):
    c = a + b
    print("Sum of {} and {} is {}".format(a,b,c))

sum_of_two_nos(10,20)
```

```
Sum of 10 and 20 is 30
```

Default arguments

Functions can have optional parameters, also called default parameters. Default parameters are parameters, which don't have to be given, if the function is called. In this case, the default values are used.

Example:



```python
def sum_of_two_nos(a, b = 10):
    c = a + b
    print("Sum of {} and {} is {}".format(a,b,c))

sum_of_two_nos(30,20)
sum_of_two_nos(30)
```

```
Sum of 30 and 20 is 50
Sum of 30 and 10 is 40
```

Keyword arguments

Example:

```
def sum_of_two_nos(a, b = 10):
    c = a + b
    print("Sum of {} and {} is {}".format(a,b,c))

sum_of_two_nos(a = 30,b = 20)
sum_of_two_nos(a = 30)
```

```
Sum of 30 and 20 is 50
Sum of 30 and 10 is 40
```

Variable length arguments

Example:

```
def sum_of_nos(*nos):
    print("Sum of {} is {}".format(nos, sum(nos)))

sum_of_nos(10,20,30,40,50)
```

```
Sum of (10, 20, 30, 40, 50) is 150
```

Extra-non-keyword arguments

Example:

```
def f(nos):
    for i in nos:
        print(i, end = " ")
p = (47,11,12)
f(p)
```

```
47 11 12
```

```
def f(x,y,z):
     print(x,y,z)
p = (47,11,12)
f(*p)
```

```
47 11 12
```

Extra-keyword arguments

Example:

```
def f(**kwargs):
    print(kwargs)

f()
f(de="German",en="English",fr="French")
```

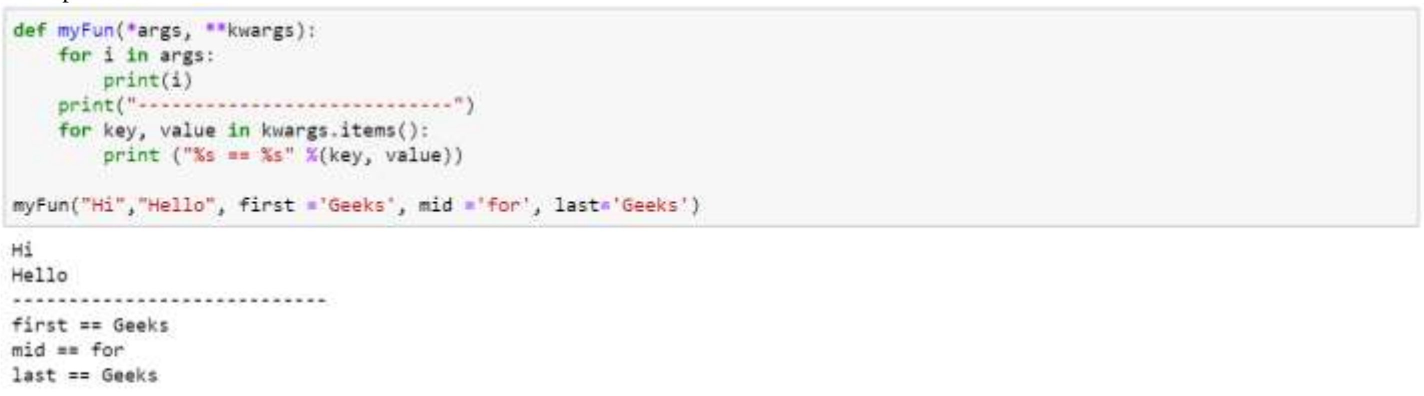
```
{}
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

Example:

```
def f(a,b,x,y):
     print(a,b,x,y)
d = {'a':'append', 'b':'block','x':'extract','y':'yes'}
f(**d)
```

```
append block extract yes
```

Example:



```python
def myFun(*args, **kwargs):
    for i in args:
        print(i)
    print("----------------------------")
    for key, value in kwargs.items():
        print ("%s == %s" %(key, value))

myFun("Hi","Hello", first ='Geeks', mid ='for', last='Geeks')
```

```
Hi
Hello
----------------------------
first == Geeks
mid == for
last == Geeks
```