Functions

in Python

What is a function?

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions*.

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python:

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Formal and Actual arguments

You can call a function by using the following types of formal arguments –

- ➢ Positional arguments / Required arguments
- ➢ Keyword arguments
- ➢ Default arguments
- ➢ Variable-length arguments

Positional / Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.
To call the function **printme()**, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
# function definition is here
def printme(str):
        “““This prints a passed string passed into this function”””
        print(str)
        return

#Now calling the function
printme()
```

Try this and observe the output!

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the printme() function in the following ways –

```
def printme(str):  # function definition is here
        """This prints a passed string passed into this function"""
        print(str)
        return

printme(str  = "This is a keyword argument")  #Now calling the function
#Try this with two or more arguments without passing them in the right order as it was defined.
```

Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed –

```
#function definition
def printinfo(name, age = 30)
        """This function just simply prints what is passed from its arguments."""
        print(f"My name is {name} and I am {age} years old!")
        #return

#Now calling the function
printinfo(age = 23, name = "Alice")
printinfo(name = "Tom")
```

Try this yourself and observe the output!

Variable-length Arguments (*args)

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments. Syntax for a function with non-keyword variable arguments is given below –

```
#function definition
def personal_info(name, age, *args) # *args is passed in the form of a tuple. [len(0 - n)]
        """This function prints whatever the value is passed from its argumets."""
        print(f"My name is {name} and I am {age} years old.")
        print(f"All the rest of arguments {args}")

personal_info("Bob", 22, male, "09123456780")
```

NOTE : If you want to pass FORMAL arguments, you have to pass them before passing variable-length arguments!
Try this and observe the output

Keyword Variable-length Arguments (**kwargs)

A keyword variable length argument is an argument that can accept any number of values provided in the format of keys and values. If we want to use a keyword variable length argument, we can declare it with ‘ ** ‘ before the argument as:

#def display(farg, **kwargs):

Here ‘**kwargs’ is called keyword variable argument. This argument internally represents a dictionary object. A dictionary stores data in the form of key and value pairs. It means, when we provide values for ‘**kwargs’, we can pass multiple pairs of values by using it.

Let’s see an example in the next slide…

Example : (**kwargs)

```
def display(farg, **kwargs):
        print(f“This is a formal argument : {farg}”)

        for key, value in kwargs.items():
                print(f“Key = {key}, Value = {value}”)

display(5, name=“Bob”, age=22)
```

Try this!

NOTE : *args takes all the informal arguments and store them into a tuple while **kwargs takes all the keywards arguments and store them into a dictionary.

The Anonymous Functions

These functions are called anonymous because they are not declared in the standard manner by using the **def** keyword. You can use the **lambda** keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambdas are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is to stack allocation by passing function, during invocation for performance reasons.

```
#Syntax
lambda [arg1 [,arg2, ....argn]]:expression  # Example in the next slide
```

Example : Anonymous function (A lambda function)

```
# This anonymous function just returns the sum of two numbers

sum = lambda x, y : x + y

#Now you can call it like this
print(f“Sum of x and y is {sum(3, 4)}”)
```

And also try this and see if this works or not.

```
x = 3, y = 5
print(f”Sum of x and y is {lambda x, y : x + y}”)
```

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier.

There are two basic scopes of variables in Python –

- Global variables
- Local variables

Global VS Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Let’s do an example –

```
total = 0
def sum(a, b):
	total = a + b
	print(f“Printing the local total : {total}”)
sum(2, 5)
print(f“Printing the global total : {total}”)  # Try and see the outputs carefully!

#Can we use a global variable inside a function?
```

The Global keyword

Sometimes, the global variable and the local variable may have the same name. In that case, the function, by default, refers to the local variable and ignores the global variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

When the programmer wants to use the global variable inside a function, he can use the keyword 'global' before the variable in the beginning of the function body as:

```
total = 0
def sum(a, b):
        global total
        total = a + b
        return total
print(sum(5, 6))
print(f"Printing the global total : {total}")
#Try this!
```

What if we want to use both a global and a local variable with same name?

When the global variable name and local variable names are same, the programmer will face difficulty to differentiate between them inside a function. For example there is a global variable ‘a’ with some value declared above the function. The programmer is writing a local variable with the same name ‘a’ with some other value inside the function. Consider the following code :

```
a = 1 # a is a global variable
def myfunction():
        a = 2 # a is a local variable
```

Now, if the programmer wants to work with global variable, how is it possible? If he uses ‘global’ keyword, then he can access only global variable and the local variable is no more available.

The globals() function

The globals() function will solve this problem. This is a built in function which returns a table of current global variables in the form of a dictionary. Hence, using this function, we can refer to the global variable ‘a’, as: globals()['a']. Now, this value can be assigned to another variable, say ‘x’ and the programmer can work with that value. This is shown here :

```
a = 1 # a is a global variable
def myfunction():
        a = 2 # a is a local variable
        x = globals()[‘a’] # get the value of global variable ‘a’ into x
        print(f“Local : {a}, Global : {x}”)
myfunction()
print(f“Global : {a}”)
#Try this!
```

Functions are First Class Objects

In Python, functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the Python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function. Also, it is possible to return a function from another function. This is similar to returning an object (or value) from a function. The following possibilities are noteworthy:

- It is possible to assign a function to a variable.
- It is possible to define one function inside another function.
- It is possible to pass a function as parameter to another function.
- It is possible that a function can return another function.

Let’s try each of the above in VS Code.

Pass by Object Reference

In the languages like C and Java, when we pass values to a function, we think about two ways:

- Pass by value or Call by value
- Pass by reference or Call by reference

Pass by value represents that a copy of the variable value is passed to the function and any modifications to that value will not reflect outside the function. Pass by reference represents sending the reference or memory address of the variable to the function. The variable value is modified by the function through memory address and hence the modified value will reflect outside the function also.

Neither of these two concepts is applicable in Python. In Python, the values are sent to functions by means of object references. We know everything is considered as an object in Python. All numbers are objects, strings are objects, and the datatypes like tuples, lists, and dictionaries are also objects. If we store a value into a variable as: x = 5.

How any variable is stored in Python?

In the case of other programming languages, a variable with a name ‘x’ is created and some memory is allocated to the variable. Then the value 10 is stored into the variable ‘x’. We can imagine ‘x’ as a box where 10 is stored. This is not the case in Python. In Python, everything is an object. An object can be imagined as a memory block where we can store some value. In this case, an object with the value ‘10’ is created in memory for which a name ‘x’ is attached. So, 10 is the object and ‘x’ is the name or tag given to that object. Also, objects are created on heap memory which is a very huge memory that depends on the RAM of our computer system. Heap memory is available during runtime of a program.

To know the location of an object in heap, we can use id() function that gives identity number of an object. For example, consider the following code snippet:

```
x = 10
print(id(x))  # Test every data types and observe the output. [Try also with functions]
```

NOTE: In python, int, float, string and tuple are immutable. List and Dictionaries are mutable. [Try to modify both mutable and immutable arguments, see what happens…]

Thank You 2

Customize this Template

Template Editing Instructions and Feedback