Unit –IV Functions

What Are Functions?, Calling Functions, Creating Functions, Passing Functions, Formal Arguments, Positional Arguments, Default Arguments, Why Default Arguments?, Default Function Object Argument Example, Variable-length Arguments, Non-keyword Variable Arguments (Tuple) , Keyword Variable Arguments (Dictionary)

Functions:

Functions are the structured or procedural programming way of organizing the logic in your programs.

The idea is to put some commonly or repeatedly done task together and make a function, so that instead of writing the same code again and again for different inputs, we can call the function. Python provides built-in functions like print(), etc. but we can also create your own functions. These functions are called user-defined functions.

Functions are often compared to procedures. Procedures are simply special cases, functions that do not return a value.

Functions may return a value back to their callers and those that are more procedural in nature do not explicitly return anything at all. Languages that treat procedures as functions usually have a special type or value name for functions that "return nothing." These functions default to a return type of "void" in C, meaning no value returned. In Python, the equivalent return object type is None.

```
>>>def hello():
        print("hello! Welcome to python")
>>>hello()
hello! Welcome to python
>>>test=hello()
hello! Welcome to python
>>>print(test)
None
```

The hello() function acts as a procedure in the code below, returning no value. If the return value is saved, you will see that its value is None:
When no items are explicitly returned or if None is returned, then Python returns None. If the function returns exactly one object, then that is the object that Python returns and the type of that object stay the same. If the function returns multiple objects, Python gathers them all together and returns them in a tuple.

Return Values	**Types**
0	None
1	object
>1	tuple

Calling Functions:

Functions are called using the same pair of parentheses that you are used to. In fact, some consider(()) to be a **two-character operator**, the **function operator**. As you are probably aware, any inputparameters or arguments must be placed between these calling parentheses. Parentheses are also usedas part of function declarations to define those arguments.

The concept of keyword arguments applies only to function invocation. The idea now is for the caller toidentify the arguments by parameter name in a function call. This specification allows for arguments tobe missing or out-of-order because the interpreter is able to use the provided keywords to match valuesto parameters.

For a simple example, imagine a functiontest(), which has the following definition.

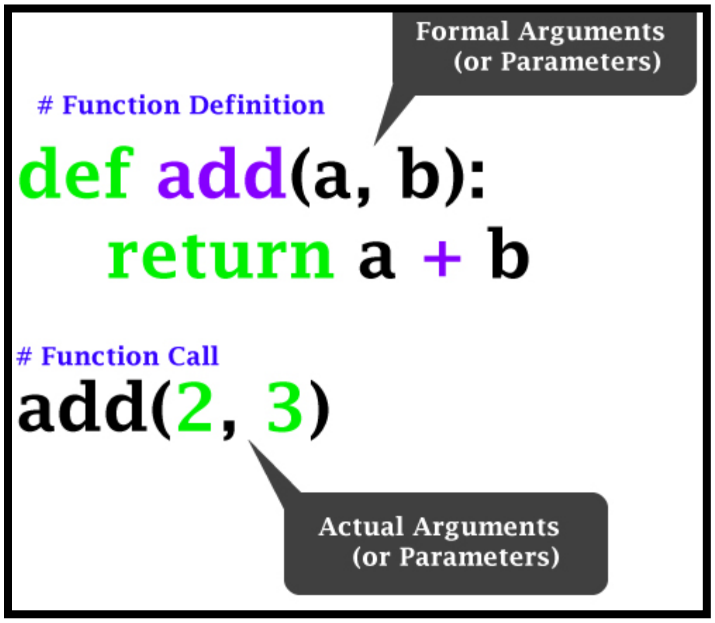
```
def test(x):
print(x)
```

then the calling of function test () is having forms:

```
>>>test(44)
44
>>>test(x=44)
```

```
44
>>>test(x='me')
me
```

Formal Arguments and actual arguments:



```
# Function Definition
def add(a, b):
    return a + b

# Function Call
add(2, 3)
```

Formal Arguments (or Parameters)
Actual Arguments (or Parameters)

```
Example:

>>>defaddme(a,b):

        c=a+b

        return c

>>>addme(3,4)

7
```

Default Arguments: Default arguments are parameters that are defined to have a default value if one is not provided in thefunction call for that argument.

The syntax of default argument in Python is the argument name is followed by an "assignment" of its default value. This assignment is merely a syntactical way of indicating that this assignment will occur if no value is passed in for that argument.

The syntax for declaring variables with default values in Python is such that all positional argumentsmust come before any default arguments:

```
defbillcalc(cost, gst=0.18):
return cost + (cost * gst)
```

Each default argument is followed by an assignment statement of its default value. If no value is givenduring a function call, then this assignment is realized.

```
>>>billcalc(100)
118.0
>>>billcalc(100,0.28)
128.0
```

Positional or Keyword Arguments:

In function, the values passed through arguments are assigned to parameters in order, by their position.

With Keyword arguments, we can use the name of the parameter irrespective of its position while calling the function to supply the values. All the keyword arguments must match one of the arguments accepted by the function.

Example:

```
defprint_name(name1, name2):
print (name1 + " and " + name2 + " are friends")
```

Calling:

```
print_name('Riyaj','Vishwanath')
Riyaj and Vishwanath are friends
```

Variable-length Arguments: Non-keyword Variable Arguments (Tuple) , Keyword Variable Arguments (Dictionary):

Sometimes you may need more arguments to process function then you mentioned in the definition. If we don’t know in advance about the arguments needed in function, we can use variable-length arguments also called arbitrary arguments.

To do this an asterisk (*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyworded variable-length arguments.

If we use one asterisk (*) like *var, then all the positional arguments from that point till the end are collected as a tuple called ‘var’ and if we use two asterisks (**) before a variable like **var, then all the positional arguments from that point till the end are collected as a dictionary called ‘var’.

```
Example:
def display(*name, **address):
        for items in name:
            print (items)
        for items in address.items():
            print (items)
```

```
Calling:
>>>display('a','b','c')
a
b
c
>>>display('a','b','c',a='Aman',c='Chandra',b='bhairav')
a
b
c
('a', 'Aman')
('c', 'Chandra')
('b', 'bhairav')
```

As you can see in the example above, *name takes all the non-keyworded arguments a, b, and c wrapped into a tuple, whereas **address takes all the keyworded arguments a='Aman', c ='Chandra', and b='bhairav' wrapped into a dictionary.