6.1 Introduction

Python functions

- ➢ A Function is a group of related statements that perform a specific task.
- ➢ Functions help us to break the long code into smaller chunks.
- ➢ As our programs grow larger and larger, the functions make the code look more organized and manageable.
- ➢ It avoids repetition of the code and makes the block of the code reusable.

Syntax

def function_name(parameters):

"""

Doc String

"""

Statement(s)

- ➢ The keyword **'def'** marks the starting of the function header.
- ➢ Parameters (the arguments) are the values which we pass to the function.
- ➢ The colon(:) marks the end of the function header.
- ➢ Docstring is used for describing what the function does. This is optional.
- ➢ The **return** statement is used to return a value from the function.

The below example was discussed at the timestamp 1:35 in the video.

Example:

```
def print_name(name):
    """
    This function prints the name
    """
    print("Hello " + str(name))
```

Function Call

Once we have defined a function, we can call it from anywhere

```
print_name('satish')
```

```
Hello satish
```

We have defined the **print_name()** function and called it with the parameter 'satish'. So this parameter is used in the task performed by the function **print_name()** and the result is returned in the same format as that specified in the **return** statement.

Once after a function is defined, we can call it from anywhere and any number of times.

Doc String

- ➢ The first string after the function header is called the docstring. It is the documentation string and tells us in a short description what all the function is about.
- ➢ Whenever we define any function, it is a good coding practice to mention the docstring.
- ➢ Docstring is usually written in triple quotes as it could extend upt multiple lines.

The syntax to print the doc string of the above function print_name() is

print(print_name.__doc__)

o/p:
This function prints the name

return statement

- ➢ The **return** statement is used to exit a function and go back to the place from where it was called.

Syntax:

return expression

- ➢ Either we can directly return a value or a variable or an expression through the **return** statement.
- ➢ If there is no expression passed through the **return** statement or if there is no return statement in a function, then the function returns **None** object.

The below example was discussed at the timestamp 4:30

```
def get_sum(lst):
    """
    This function returns the sum of all the elements in a list
    """
    #initialize sum
    _sum = 0

    #iterating over the list
    for num in lst:
        _sum += num
    return _sum
```

```
s = get_sum([1, 2, 3, 4])
print(s)
```

```
10
```

```
#print doc string
print(get_sum.__doc__)
```

```
    This function returns the sum of all the elements in a list
```

In the above example, we are passing a list of elements as an input to the **get_sum()** function and the function returns the sum of all the elements in the list.

How Functions Work in Python?

Scope and Lifetime of Variables

- ➢ Scope of a variable is the portion of the program upto which a variable gets recognized.
- ➢ Variables defined inside a function are not seen from the outside. Hence such variables have local scope.
- ➢ Lifetime of a variable is the period through which the variable lies in the memory.

- ➢ The lifetime of variables inside a function is as long as the function executes.
- ➢ Variables are destroyed once we return from the function.

The below example was discussed at the timestamp

```
global_var = "This is global variable"

def test_life_time():
    """
    This function test the life time of a variables
    """
    local_var = "This is local variable"
    print(local_var)       #print local variable local_var

    print(global_var)      #print global variable global_var



#calling function
test_life_time()

#print global variable global_var
print(global_var)

#print local variable local_var
print(local_var)
```

```
This is local variable
This is global variable
This is global variable

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-d5226680661e> in <module>()
     19
     20 #print local variable local_var
---> 21 print(local_var)

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

In the above example, ‘global_var’ is a global variable and it can be accessed either inside a function or outside a function. Whereas ‘local_var’ is a local variable defined inside a function, so we could access it only inside the function. Once the control comes out of the function, this variable becomes invalid.

Program to compute the HCF of two numbers

The below example was discussed at the timestamp 10:30 in the video.

```
def computeHCF(a, b):
    """
    Computing HCF of two numbers
    """
    smaller = b if a > b else a  #consice way of writing if else statement

    hcf = 1
    for i in range(1, smaller+1):
        if (a % i == 0) and (b % i == 0):
            hcf = i
    return hcf

num1 = 6
num2 = 36

print("H.C.F of {0} and {1} is: {2}".format(num1, num2, computeHCF(num1, num2)))
```

```
H.C.F of 6 and 36 is: 6
```

6.2 Types of Functions

There are two types of functions. They are

1) Built-in functions
2) User defined functions

Built-in Functions

1) abs()

The abs() function is used to find out the absolute value of a given number.

The below example was discussed at the timestamp 0:38 in the video.

```
# find the absolute value

num = -100

print(abs(num))
```

```
100
```

2) all()

The all() function takes an iterable(either a list or a tuple or a set) as an input and returns **True** if all the elements in the iterable are **True**. If any element in the iterable is **False**, then it returns **False**.

Note: The values **0** and **None** are also considered as **False**. Here if the given iterable contains any element as **0** or **False** or **None**, then the all() function returns the result as False. Otherwise it returns **True**.

The below examples were discussed at the timestamp 1:40.

```
lst = [1, 2, 3, 4]
print(all(lst))
```

```
True
```

```
lst = (0, 2, 3, 4)    # 0 present in list
print(all(lst))
```

```
False
```

```
lst = []              #empty list always true
print(all(lst))
```

```
True
```

```
lst = [False, 1, 2]   #False present in a list so all(lst) is False
print(all(lst))
```

```
False
```

3) dir()

➢ The dir() returns a list of the valid attributes of the object.

➢ If the object supports the dir() method, whenever the method is called with this object as an argument, then it must return a list of all the attributes of that object.

➢ If the object doesn't support the dir() function, whenever the method is called with this object as an argument, then this method tries to find information from the **dict** attribute (if defined), and from the type object. In this case, the list returned from dir() may not be complete.

The below example was discussed at the timestamp 4:20 in the video.

```
: numbers = [1, 2, 3]

  print(dir(numbers))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__',
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rm
ul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

4) **divmod()**

The divmod() function takes two values(say **a** and **b**) as an input and returns a pair of values in the form of a tuple. The first value in the tuple would be the **quotient** obtained when **a** is divided by **b** and the second value would be the **remainder** obtained when **a** is divided by **b**.

The below example was discussed at the timestamp 5:23.

```
print(divmod(9, 2)) #print quotient and remainder as a tuple

#try with other number
```

```
(4, 1)
```

5) **enumerate()**

The enumerate() adds a counter to an iterable and returns it. So whenever we loop through the iterable (with counter added), we get each item in the form of a tuple with a pair of values. The first value would be the index and the second value would be the corresponding element in the iterable.

The below example was discussed at the timestamp 6:08

```
numbers = [10, 20, 30, 40]

for index, num in enumerate(numbers,10):
    print("index {0} has value {1}".format(index, num))
```

```
index 10 has value 10
index 11 has value 20
index 12 has value 30
index 13 has value 40
```

The first argument in enumerate() should be the iterable to which we want to add the counter. The second argument should be the starting index of the counter. If we want to manually assign the starting index, then we have to pass the required starting value. Otherwise, the default starting index would be 0.

6) **filter()**

The filter() method constructs an iterator from elements of an iterable for which a function returns True.

The below example was discussed at the timestamp 9:10

```
def find_positive_number(num):
    """
    This function returns the positive number if num is positive
    """
    if num > 0:
        return num
```

```
number_list = range(-10, 10) #create a list with numbers from -10 to 10
print(list(number_list))

positive_num_lst = list(filter(find_positive_number, number_list))

print(positive_num_lst)
```

```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In the above example, we are defining the function **find_positive_number()** which returns the input number if it is greater than 0. Otherwise, it returns None.

So now the filter() function will loop through the input iterator and passes each element of the iterator through the function **find_positive_number()** and creates an iterator with only those elements that do not return **None** when passed through the function **find_positive_number().**

We are passing a range of values in the interval [-10,9] as an input to the filter() and only those values that are greater than 0 are returned.

7) isinstance()

The isinstance() function checks if the **object** (first argument) is an instance or subclass of **class_name** class (second argument).

Syntax

isinstance(object, class_name)

The below example was explained at the timestamp 12:45.

```
: lst = [1, 2, 3, 4]
print(isinstance(lst, list))

#try with other datatypes tuple, set
t = (1,2,3,4)
print(isinstance(t, list))
```

```
True
False
```

In the first example, the **isinstance()** is checking if the object 'lst' is an instance of 'list' class. If yes, then it returns **True**. Otherwise, it returns **False**. As 'lst' is a list, the isinstance() is returning **True** in this case.

In the second example, the **isinstance()** is checking if the object 't' is an instance of 'list' class. If yes, then it returns **True**. Otherwise, it returns **False**. As 't' is a tuple, but not a list, the isinstance() is returning **False** In this case.

8) **map()**

Syntax:

map(function_name, iterable_name)

The map() will apply the function 'function_name' to each and every element in the iterable 'iterabl_name'. The result would be of 'map' class. If we want to get the result in the form of a list (or) a tuple (or) a set, then we have to apply the list() (or) tuple() (or) set() functions on the map() function output.

The below example was discussed at the timestamp 13:50.

```
numbers = [1, 2, 3, 4]

#normal method of computing num^2 for each element in the list.
squared = []
for num in numbers:
    squared.append(num ** 2)

print(squared)
```

```
[1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]

def powerOfTwo(num):
    return num ** 2

#using map() function
squared = list(map(powerOfTwo, numbers))
print(squared)
```

```
[1, 4, 9, 16]
```

In the above example, we want to compute the square of each element in the given input list. One way is to iterate through the whole list and perform power of 2 operation on each element.

The other way is to define a function **powerOfTwo()** which performs square on the given element. We use this function as the first argument and the list 'numbers' as the second argument in the map() function.

9) reduce()

➢ reduce() function is for performing some computation on a list and returning the result.

➢ It applies a rolling computation to sequential pairs of values in a list.

The below example was discussed at the timestamp 17:10

```
#product of elemnts in a list
product = 1
lst = [1, 2, 3, 4]

# traditional program without reduce()
for num in lst:
    product *= num
print(product)
```

```
24
```

```
#with reduce()
from functools import reduce # in Python 3.

def multiply(x,y):
    return x*y;

product = reduce(multiply, lst)
print(product)
```

```
24
```

In the above example, in the first code snippet, we are performing the product of all the elements in the list.

In the second code snippet, we are performing the same task using the reduce() function.

User Defined Functions

- Functions that we define ourselves to do certain specific task are referred as user-defined functions
- If we use functions written by others in the form of a library, it can be termed as library functions.

Advantages of User Defined Functions

- User-defined functions help to decompose a large program into small segments which makes the program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmers working on large projects can divide the workload by making different functions.

The below example was discussed at the timestamp 21:34

```
def product_numbers(a, b):
    """
    this function returns the product of two numbers
    """
    product = a * b
    return product

num1 = 10
num2 = 20
print "product of {0} and {1} is {2} ".format(num1, num2, product_numbers(num1, num2))
```

```
product of 10 and 20 is 200
```

In the above example, **product_numbers()** is a user defined function to perform the product operation between two numbers.

Python program to make a simple calculator that can add, subtract, multiply and division

```
def add(a, b):
    """
    This function adds two numbers
    """
    return a + b

def multiply(a, b):
    """
    This function multiply two numbers
    """
    return a * b

def subtract(a, b):
    """
    This function subtract two numbers
    """
    return a - b

def division(a, b):
    """
    This function divides two numbers
    """
    return a / b
```

```
: print("Select Option")
  print("1. Addition")
  print ("2. Subtraction")
  print ("3. Multiplication")
  print ("4. Division")

  #take input from user
  choice = int(input("Enter choice 1/2/3/4"))

  num1 = float(input("Enter first number:"))
  num2 = float(input("Enter second number:"))
  if choice == 1:
      print("Addition of {0} and {1} is {2}".format(num1, num2, add(num1, num2)))
  elif choice == 2:
      print("Subtraction of {0} and {1} is {2}".format(num1, num2, subtract(num1, num2)))
  elif choice == 3:
      print("Multiplication of {0} and {1} is {2}".format(num1, num2, multiply(num1, num2)))
  elif choice == 4:
      print("Division of {0} and {1} is {2}".format(num1, num2, division(num1, num2)))
  else:
      print("Invalid Choice")
```

```
Select Option
1. Addition
2. Subtraction
3. Multiplication
4. Division
Enter choice 1/2/3/43
Enter first number:12.2
Enter second number:2.3
Multiplication of 12.2 and 2.3 is 28.059999999999995
```

In the above program, we are defining the functions to perform addition, subtraction, multiplication and division operations separately for each.

We have to enter a number of our choice(in between 1 to 4 inclusive) and the corresponding operation would be performed and the result gets printed.

6.3 Function Arguments

The below example was discussed at the timestamp 0:20

```
def greet(name, msg):
    """
    This function greets to person with the provided message
    """
    print("Hello {0} , {1}".format(name, msg))

#call the function with arguments
greet("satish", "Good Morning")
```

```
Hello satish , Good Morning
```

```
#suppose if we pass one argument

greet("satish") #will get an error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-b48ea98044bf> in <module>()
      1 #suppose if we pass one argument
      2
----> 3 greet("satish") #will get an error

TypeError: greet() missing 1 required positional argument: 'msg'
```

- In this example, we are defining the function greet() which takes two parameters.
- Here whenever we want to call the function greet(), then we have to pass the values as well. The values passed while calling the function will be assigned to the arguments in the function header in the same order as they are present.
- The number of arguments passed while calling the function should be the same as the number of arguments defined in the function.
- In our example, as we have two parameters defined in the greet() function, we are also passing the two values ‘satish’ and ‘Good Morning’. The value ‘satish’ gets assigned to the ‘name’ parameter and the value ‘Good Morning’ gets assigned to the ‘msg’ parameter and the function gets executed.
- When we are calling the greet() function for the second time, we are passing only 1 value. The first value ‘satish’ gets assigned to the first argument ‘name’ and there would be no value for the ‘msg’ argument. The method signature now is different from the one that was defined. Hence it throws an error. In such cases, whenever we want certain arguments to

hold the same value for multiple function calls and change only a few of the arguments, we have to go with default arguments.

Different Forms of Arguments

1) **Default Arguments**

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

At the time of the function call, if we again pass any value for that argument, then the newly passed value will be assigned to the argument. Otherwise, the default value that was assigned in the function definition would be assigned to this argument.

Such arguments with default values assigned at the time of defining the function itself are called **Default Arguments**.

Below is an example that was discussed at the timestamp 1:32

```
def greet(name, msg="Good Morning"):
    """
    This function greets to person with the provided message
    if message is not provided, it defaults to "Good Morning"
    """
    print("Hello {0} , {1}".format(name, msg))

greet("satish", "Good Night")
```

```
Hello satish , Good Night
```

```
#with out msg argument
greet("satish")
```

```
Hello satish , Good Morning
```

```
Once we have a default argument, all the arguments to its right must also have default values.

def greet(msg="Good Morning", name)

#will get a SyntaxError : non-default argument follows default argument
```

In this example, we are assigning the value ‘Good Morning’ to the ‘msg’ argument. This is the default value assigned to this argument.

At the time of the first function call, we are passing the values ‘satish’ and ‘Good Night’. So these two values get assigned to the two arguments.

But in the second function call, we are passing only the value ‘satish’ which gets assigned to the ‘name’ argument. Now there is no value passed

to the ‘msg’ argument. In such cases, the default value ‘Good Morning’ that was assigned in the function definition, will now be assigned to this ‘msg’ argument.

In this example, ‘msg’ is a default argument and ‘name’ is a non-default argument. At the time of function definition and function call, we must make sure the non default arguments come first and then the default arguments, as it is a good programming practice.

2) Keyword Arguments

kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function.

Below is an example discussed at the timestamp 4:45

```
def greet(**kwargs):
    """
    This function greets to person with the provided message
    """
    if kwargs:
        print("Hello {0} , {1}".format(kwargs['name'], kwargs['msg']))
greet(name="satish", msg="Good Morning")
```

```
Hello satish , Good Morning
```

In the above example, whenever we want to manually assign values to each of the parameters in the key-value format, then we call them Keyword arguments.

We have to assign each value with the help of the argument names as the keys while calling the function. We have to mention only **kwargs as an argument in the function definition.

In the function body, this **kwargs would be considered as a dictionary and each of the arguments in it will behave like the keys. The accessing of the values associated with these keys is the same as how we access the values in a dictionary.

Here ‘name’ and ‘msg’ are the keys and kwargs is considered as a dictionary in the function body. The values associated with these keys will be accessed as kwargs[‘name’] and kwargs[‘msg’].

3) **Arbitrary Arguments**

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 arbitrary number of arguments.

The below example was discussed at the timestamp 6:50

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

    for name in names:
        print("Hello,  {0} ".format(name))

greet("satish", "murali", "naveen", "srikanth")
```

```
('satish', 'murali', 'naveen', 'srikanth')
Hello,  satish
Hello,  murali
Hello,  naveen
Hello,  srikanth
```

In this example, we do not have any idea about the number of values we use at runtime. In the function call, we have passed 4 values. In the next function call, we might pass more or less values. In such cases, when we do not have idea about the number of values we pass at the runtime, we go with these arbitrary arguments. It means the argument having an arbitrary length.