Unit – 2

Function:

A function is a self contained program that carries out some specified, well defined task. Functions are basically used when there are tasks that are to be performed in exactly the same way again and again. A function contains with in some statements to be performed. These intended action can be accessed anywhere at any time of the program by just calling the function name.

Types of Function:

Library Function: The Python language includes built-in functions such as dir, len, and abs. The def keyword is used to create functions that are user specified.

User – Define Function

User-defined functions are the fundamental building block of any programme and are essential for modularity and code reuse because they allow programmers to write their own function with function name that the computer can use.

Creating User Defined Function

A function definition begins with def (short for define). The syntax for creating a user defined function is as follows –

Syntax –

```
def function_name(parameter1, parameter2, …) :
statement_1
statement_2
```

There are two ways to call user defined functions.
Call By Reference: Variables initialized are stored somewhere in the computer's memory. Each variable has a particular address of the location where it is stored. Call by reference refers to the calling of function by passing arguments as the address of these locations of the variables rather than their value.

Call By Value: Calling functions refers to the request of performing the statements within a function named to be called whenever, we call a function, generally we pass some arguments to it these arguments passed are generally the value of variables. This process of calling a function is referred to as call by value.

Arguments and Parameters

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function. User-defined function could potentially take values when it is called. A value received in the matching parameter specified in the function header and sent to the function as an argument is known as an argument.

Parameters are of two types **–** Formal Parameters, Actual Parameters
Formal Parameters are the parameters which are specified during the definition of the function. Actual Parameters are the parameters which are specified during the function call.

Global Variable – A variable that is defined in Python outside of any function or block is referred to as a global variable. It is accessible from any functions defined afterward.
Local Variable – A local variable is one that is declared inside any function or block. Only the function or block where it is defined can access it.

Examples of function:

Call by Reference :

function to add 2 numbers and display their sum.

```
def addnum():
fnum = int(input(“Enter first number: “))
snum = int(input(“Enter second number: “))
sum = fnum + snum
print(“ sum = “,sum)
#function call
addnum()
```

Call by Value:

function calcFact() to calculate and display the factorial of a number num passed as an argument.

```
def calcFact(num):
fact = 1
for i in range(num,0,-1):
```

```
fact = fact * i
print("Factorial of",num,"is",fact)
num = int(input("Enter the number: "))
calcFact(num)
```

function calcPow() that accepts base and exponent as arguments and returns the value Baseexponent where Base and exponent are integers.

```
def calcpow(number,power):
result = 1
for i in range(1,power+1):
result = result * number
return result
base = int(input("Enter the value for the Base: "))
expo = int(input("Enter the value for the Exponent: "))
answer = calcpow(base,expo)
print(base,"raised to the power",expo,"is",answer)
```

String as Parameters

function that accepts the first name and lastname as arguments, concatenate them to get full name and display.

```
def fullname(first,last):
fullname = first + " " + last
print("Hello",fullname)
first = input("Enter first name: ")
last = input("Enter last name: ")
fullname(first,last)
```

Recursion : A function is called recursive, if the body of function calls the function itself until the condition for recursion is true. Thus, a Python recursive function has a termination condition. In other words Recursion means calling a function itself again and again.

Important points related to recursion:

- There must be a terminating condition in the program which will marks the end of the process. (if not the process will get in an infinte loop)
- Each time a new recursive call is made a new memory space is allocated to each local variable used by the recursive function. or we can say that during each recursive call the local variables used in the function gets a different values having the same name.
- The values in the recursive function are pushed onto the stack with each call to the function and all these values are available to the function call when they are poped from the stack.

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Examples Of Recursion

Program to print the fibonacci series upto n_terms using Recursive function

```
def recursive_fibonacci(n):
if n <= 1:
    return n
else:
    return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))
n_terms = 10
# check if the number of terms is valid
if n_terms <= 0:
print("Invalid input ! Please input a positive value")
else:
print("Fibonacci series:")
for i in range(n_terms):
```

```
    print(recursive_fibonacci(i))
```

2.

```
# Program to calculate factorial of a number Recursive function.

def Recur_facto(n):
    if (n == 0):
        return 1
    return n * Recur_facto(n-1)

print(Recur_facto(6))
```

Tail-Recursion:

A unique type of recursion where the last procedure of a function is a recursive call. The recursion may be automated away by performing the request in the current stack frame and returning the output instead of generating a new stack frame. The tail-recursion may be optimized by the compiler which makes it better than non-tail recursive functions.

```
# Program to calculate factorial of a number using a Tail-Recursive function.

def Recur_facto(n, a = 1):
    if (n == 0):
        return a
    return Recur_facto(n - 1, n * a)

# print the result

print(Recur_facto(6))
```

Lambda Function

Lambda Functions in Python are anonymous functions, implying they don't have a name. The def keyword is needed to create a typical function in Python, as we already know. We can also use the lambda keyword in Python to define an unnamed function.

Example

1.

```
Lambda : print (‘Hello class’)
```

2.

```
X=2
Print ( lambda  y : x+1)
```

3.

```
a = lambda x, y : (x * y)
print(a(4, 5))
```

4.

```
code used to filter the odd numbers from the given list
list_ = [35, 12, 69, 55, 75, 14, 73]
odd_list = list(filter( lambda num: (num % 2 != 0) , list_ ))
print('The list of odd number is:',odd_list)
```

inbuilt functions

The Python built-in functions are defined as the functions whose functionality is pre-defined in Python.

str()

str() function is used to convert an object to its string representation. It is a built-in function that can be used to convert objects of different data types, such as integers, and floats.

```
val=10
val_str= str(val)
print(val_str)
```

globals()

globals() function in Python returns the dictionary of current global symbol table.

```
a = 5
def func():
  c = 10
d = c + a
# Calling globals()
globals()['a'] = d
print (a)
# Driver Code
func()
```

locals()

locals() function returns the dictionary of the current local symbol table.

```
def demo1():
print("Here no local variable  is present : ", locals())
  # here local variables are present
def demo2():
name = "Ankit"
print("Here local variables are present : ", locals())
  # driver code
demo1()
demo2()
```

vars()

The vars() method takes only one parameter and that too is optional. It takes an object as a parameter which may be can **a module, a class, an instance, or any object having __dict__ attribute.**

```
class  abc:

def __init__(self, name1 = "Arun", num2 = 46, name3 = "Rishab"):

    self.name1 = name1

    self.num2 = num2

    self.name3 = name3

 xyz = abc()

print(vars(xyz))
```

eval()

eval() function parse the expression argument and evaluate it as a Python expression and runs Python expression (code) within the program.

eval() is not much used due to security reasons, as we explored above. Still, it comes in handy in some situations like:You may want to use it to allow users to enter their own “scriptlets”: small expressions (or even small functions), that can be used to customize the behavior of a complex system. eval() is also sometimes used in applications needing to evaluate math expressions. This is much easier than writing an expression parser.

```
def function_creator():

        # expression to be evaluated

        expr = input("Enter the function(in terms of x):")

        # variable used in expression

        x = int(input("Enter the value of x:"))

        # evaluating expression

        y = eval(expr)

        # printing evaluated result
```

```
    print("y =", y)

    if __name__ == "__main__":

    function_creator()
```

- **Output**

Enter the function(in terms of x):x*(x+1)*(x+2)

Enter the value of x:3

y = 60

exec()

exec() function is used for the dynamic execution of Python programs which can either be a string or object code.

Example 1:

program = 'a = 5\nb=10\nprint("Sum =", a+b)' exec(program)

Output: Sum = 15

Example 2:

from math import *

exec("print(dir())")

execfile()

Function executes the contents of a file.

execfile(“abc.py")

repr()

repr() function returns a printable representation of the object by converting that object to a string.

print(repr(['a', 'b', 'c']))

Output: ['a', 'b', 'c']

var = ‘ABC'

print(repr(var))

ABC

ascii()

Returns a string as a printable representation of the object passed, escaping the non-ASCII characters.

x = ascii("My name is Ståle")

print(x)

Output

'My name is St\e5le'

String:

The string is a data type in python, which is used to handle an array of characters. The string is a sequence of Unicode characters that may be a combination of letters, numbers, or special symbols enclosed within single, double, or even triple quotes.

String Methods : Done programs in class.

Tuples:

Python Tuple is a collection of objects separated by commas. In some ways, a tuple is similar to a Python list in terms of indexing, nested objects, and repetition but the main difference between both is Python tuple is immutable, unlike the Python list which is mutable.

There are various ways by which you can create a tuple in Python. They are as follows:

- Using round brackets
- With one item
- Tuple Constructor

To create a tuple we will use () operators.

```
var = ("amit", "raj", "deepak")

print(var)
```

To create a tuple With one item
Python 3.11 provides us with another way to create a Tuple.

```
values : tuple[int | str, ...] = (1,2,4,"ashish")

print(values)
```

Tuple Constructor in Python
To create a tuple with a Tuple constructor, we will pass the elements as its parameters.

```
tuple_constructor = tuple(("dsa", "developement", "deep learning"))

print(tuple_constructor)
```

Immutable in Tuples?
Tuples in Python are similar to Python lists but not entirely. Tuples are immutable and ordered and allow duplicate values. Some Characteristics of Tuples in Python.

- We can find items in a tuple since finding any item does not make changes in the tuple.
- One cannot add items to a tuple once it is created.
- Tuples cannot be appended or extended.
- We cannot remove items from a tuple once it is created.

Example:

```
mytuple = (1, 2, 3, 4, 5)

print(mytuple[1])

print(mytuple[4])

mytuple = (1, 2, 3, 4, 2, 3)
```

```
print(mytuple)

mytuple[1] = 100

print(mytuple)
```

Tuples in Python provide two ways by which we can access the elements of a tuple.

- Using a positive index
- Using a negative index

Python Access Tuple using a Positive Index

Using square brackets we can get the values from tuples in Python.

```
var = ("dinesh", "ajeet", "vikas")

print("Value in Var[0] = ", var[0])

print("Value in Var[1] = ", var[1])

print("Value in Var[2] = ", var[2])
```

Access Tuple using Negative Index

```
var = (1, 2, 3)

print("Value in Var[-1] = ", var[-1])

print("Value in Var[-2] = ", var[-2])

print("Value in Var[-3] = ", var[-3])
```

Different operations related to tuples in Python:

- Concatenation
- Nesting
- Repetition
- Slicing
- Deleting
- Finding the length
- Multiple Data Types with tuples
- Conversion of lists to tuples
- Tuples in a Loop

To Concatenation of Python Tuples, we will use plus operators(+).

```
# Code for concatenating 2 tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'JAVA')
print(tuple1 + tuple2)
```

A nested tuple in Python means a tuple inside another tuple.

```
# Code for creating nested tuples
tuple1 = (0, 1, 2, 3)
tuple2 = ('python', 'java')
tuple3 = (tuple1, tuple2)
print(tuple3)
```

We can create a tuple of multiple same elements from a single element in that tuple.

```
# Code to create a tuple with repetition
tuple3 = ('python',)*3
print(tuple3)
```

Slicing a Python tuple means dividing a tuple into small tuples using the indexing method.

```
# code to test slicing
tuple1 = (0 ,1, 2, 3)
print(tuple1[1:])
print(tuple1[::-1])
print(tuple1[2:4])
```

move individual tuple elements is not possible, but we can delete the whole Tuple using Del keyword.

```
# Code for deleting a tuple
tuple3 = ( 0, 1)
del tuple3
print(tuple3)
```

To find the length of a tuple, we can use Python's len() function and pass the tuple as the parameter.

```
# Code for printing the length of a tuple
tuple2 = ('python', 'java')
print(len(tuple2))
```

Tuples in Python are heterogeneous in nature. This means tuples support elements with multiple datatypes.

```
# tuple with different datatypes
tuple_obj = ("immutable",True,23)
print(tuple_obj)
```

We can convert a list in Python to a tuple by using the tuple() constructor and passing the list as its parameters.

```
# Code for converting a list and a string into a tuple
list1 = [0, 1, 2]
print(tuple(list1))
# string 'python'
print(tuple('python'))
```

We can also create a tuple with a single element in it using loops.

```
# python code for creating tuples in a loop
tup = ('raj',)

# Number of time loop runs
n = 5
for i in range(int(n)):
    tup = (tup,)
    print(tup)
```

List in Python:

List in python language is used to store multiple elements in a single variable. Lists are one of the 4 data types in python language along with tuple, set, and dictionary. List in Python is created by putting elements inside the square brackets, separated by commas. A list can have any number of elements and it can be of multiple data types. Also, all the operation of the string is similarly applied on list data type such as slicing, concatenation, etc. Also, we can create the nested list i.e list containing another list.

We can access the data simply from lists as ordered; unlike in sets, the data will be unordered. To access the data, we can use several ways to iterate through each element inside a list.

1. Using for loop

```
1. list1 = [3, 5, 7, 2, 4]
2. for i in list1:
3.     print (i)
```

Using for and range:

```
1. list1 = [3, 5, 7, 2, 4]
2. length = len (list1)
3. for i in range (0, len (list1)):
4.     print (list1 [i])
```

2. Using List Comprehension

This is the simple and suggested way to iterate through a list in Python.

```
1. list1 = [3, 5, 7, 2, 4]
2. [print (i) for i in list1]
3. print ("\n")
4. [print (list1 [i]) for i in range (0, len (list1))]
5. print ("\n")
```

3. Using enumerate():

The enumerate function converts the given list into a list of tuples. Another important fact about this function is that it keeps count of the iterations. This is a built-in function in Python.

```
1. list1 = [3, 5, 7, 2, 4]
2. for i, j in enumerate (list1):
3.     print ("index = ", i, "value: ", j)
```

4. Using lambda function and map():

These are anonymous functions. There is a function map () in Python that can accept a function as an argument, and it calls the function with every element in the iterable, and a new list with all the elements from the iterable will be returned.

```
1. list1 = [3, 6, 1, 8, 7]
2. result = list (map (lambda num: num, list1))
3. print (result)
```

5.Using Numpy function

```
import numpy as np

a = np.arange(5)

for x in np.nditer(a):
    print(x)
```