

UNIT 2

Python has two primitive loop commands:

- while loop
- for loop

The while loop: With the while loop we can execute a set of statements as long as a condition is true.

Syntax

```
while <condition>:  
    { code block }
```

#Using else statement in while loop

```
while condition:
```

```
    # execute these statements
```

```
else:
```

```
    # execute these statements
```

Example 1:

```
i = 1  
while i < 6:  
    print(i)  
    i += 1
```

Output:

```
1  
2  
3  
4  
5
```

Example 2:

```
counter = 0  
while (counter < 10):  
    counter = counter + 3  
    print("Python Loops")
```

else:

```
print("Code block inside the else statement")
```

Output:

```
Python Loops
Python Loops
Python Loops
Python Loops
Code block inside the else statement
```

The for loop: A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

Syntax

for value in sequence:

```
{ code block }
```

Example 1:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

Output:

```
apple
banana
cherry
```

Example 2: Looping through a string

```
for x in "banana":
    print(x)
```

Output:

```
b
a
n
a
n
a
```

The range() Function : With the help of the range() function, we may produce a series of numbers. range(10) will produce values between 0 and 9. (10 numbers).

We can give specific start, stop, and step size values in the manner range(start, stop, step size). If the step size is not specified, it defaults to 1.

Example 1: # Python program to show the working of range() function

```
print(range(15))
print(list(range(15)))
print(list(range(4, 9)))
print(list(range(5, 25, 4)))
```

Output:

```
range(0, 15)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
[4, 5, 6, 7, 8]
[5, 9, 13, 17, 21]
```

Example 2:

```
for x in range(6):
    print(x)
```

Output:

```
0
1
2
3
4
5
```

Example 3:

The range() function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: range(2, 6), which means values from 2 to 6 (but not including 6):

```
for x in range(2, 6):  
    print(x)
```

Output:

```
2  
3  
4  
5
```

Example 4:

The range() function defaults to increment the sequence by 1, however it is possible to specify the increment value by adding a third parameter: range(2, 30, 3).

```
for x in range(2, 30, 3):  
    print(x)
```

Output:

```
2  
5  
8  
11  
14  
17  
20  
23  
26  
29
```

Nested loop: A nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop".

Example:

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]  
for x in adj:  
    for y in fruits:  
        print(x, y)
```

Output:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

Loop Control Statements

- i. **Continue Statement:** With the continue statement we can stop the current iteration of the loop, and continue with the next.

Example :

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

Output:

```
apple
cherry
```

- ii. **Break Statement:** With the break statement we can stop the loop before it has looped through all the items:

Example:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

Output:

```
apple
banana
```

- iii. **Pass Statement:** for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

Example:

```
for x in [0, 1, 2]:
    pass          # having an empty for loop like this, would raise an error without the pass statement
```

Numpy Array

Python does not have built-in support for arrays, but python list can be used instead. However, to work with arrays in python you will have to import a library, like the Numpy.

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).

Create a NumPy ndarray Object

Numpy is used to work with arrays. The array object in Numpy is called ndarray. We can create a Numpy ndarray object by using the array() function.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

Output:

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

type(): This built-in python functions tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Example: Use a tuple to create a Numpy array.

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

Output:

```
[1 2 3 4 5]
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example:

```
import numpy as np
arr = np.array(42)
print(arr)
```

Output:

```
42
```

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array. These are the most common and basic arrays.

Example: Creating a 1-D array containing the values 1,2,3,4,5.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Output:

```
[1 2 3 4 5]
```

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array. These are often used to represent matrix or 2nd order tensors.

Example: Create a 2D array containing two arrays with the values 1,2,3 and 4,5,6.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

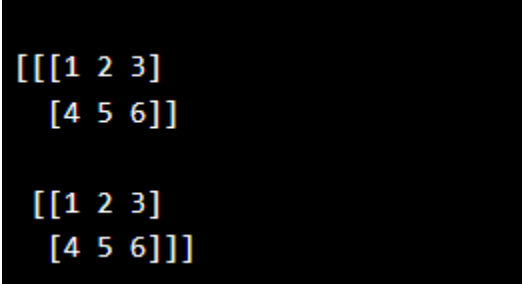
3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array. These are often used to represent a 3rd order tensor.

Example: Create a 3D array with two 2D arrays, both containing two arrays with the values 1,2,3 and 4,5,6.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(arr)
```

Output:



```
[[[1 2 3]
    [4 5 6]]
 [[1 2 3]
    [4 5 6]]]
```

Check Number of Dimensions

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

Example: Check how many dimensions the arrays have:

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

Output:



```
0
1
2
3
```

Higher Dimensional Arrays

An array can have any number of dimensions. When the array is created, you can define the number of dimensions by using the **`ndmin`** argument.

Example: Create an array with 5 dimensions and verify that it has 5 dimensions.

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

Output:

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

Access Array Elements

Array indexing is the same as accessing an array element. You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example: Get the first element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Output:

```
1
```

Example: Get the second element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[1])
```

Output:

```
2
```

Example: Get third and fourth elements from the following array and add them.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
```

```
print(arr[2] + arr[3])
```

Output:

```
7
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element. Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Example: Access the element on the first row, second column.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

Output:

```
2nd element on 1st dim:  2
```

Example: Access the element on the 2nd row, 5th column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd row: ', arr[1, 4])
```

Output:

```
5th element on 2nd dim:  10
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example: Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Output:

```
6
```

Example Explained

`arr[0, 1, 2]` prints the value 6.
And this is why:

The first number represents the first dimension, which contains two arrays:

`[[1, 2, 3], [4, 5, 6]]`

and:

`[[7, 8, 9], [10, 11, 12]]`

Since we selected 0, we are left with the first array:

`[[1, 2, 3], [4, 5, 6]]`

The second number represents the second dimension, which also contains two arrays:

`[1, 2, 3]`

and:

`[4, 5, 6]`

Since we selected **1**, we are left with the second array:

`[4, 5, 6]`

The third number represents the third dimension, which contains three values:

4

5

6

Since we selected 2, we end up with the third value:

6

Negative Indexing

Use negative indexing to access an array from the end.

Example: Print the last element from the 2nd dim:

```
import numpy as np
```

```
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
```

```
print('Last element from 2nd dim: ', arr[1, -1])
```

Output:

```
Last element from 2nd dim: 10
```

Numpy Array Slicing

Slicing in python means taking elements from one given index to another given index.

- We pass slice instead of index like this: `[start:end]`.
- We can also define the step, like this: `[start:end:step]`.
- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension

- If we don't pass step its considered 1

Example: Slice elements from index 1 to index 5 from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
```

Output:

```
[2 3 4 5]
```

Example: Slice elements from index 4 to the end of the array:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
```

Output:

```
[5 6 7]
```

Negative Slicing:

Use the minus operator to refer to an index from the end:

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
print(arr[1:5:2])
print(arr[::-2])
```

Output:

```
[5 6]
[2 4]
[1 3 5 7]
```

Slicing 2D Array

```
import numpy as np
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
print(arr[0:2, 2])
print(arr[0:2, 1:4])
```

Output:

```
[7 8 9]
[3 8]
[[2 3 4]
 [7 8 9]]
```

Numpy Array Shape and Reshape

The shape of an array is the number of elements in each dimension. NumPy arrays have an attribute called **shape** that returns a tuple with each index having the number of corresponding elements.

Example:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
```

The example above returns (2, 4), which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

Output:

```
(2, 4)
```

Reshaping Arrays

Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

Example: Convert the following 1-D array with 12 elements into a 2-D array. The outmost dimension will have 4 arrays, each with 3 elements.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

Output:

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Example 2: Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)
print(newarr)
```

Output:

```
[[[ 1  2]
   [ 3  4]
   [ 5  6]]

 [[ 7  8]
   [ 9 10]
  [11 12]]]
```

Iterating Arrays

Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python. If we iterate on a 1-D array it will go through each element one by one.

Example: Iterate on the elements of the following 1-D array:

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
    print(x)
```

Output:

```
1
2
3
```

Example: Iterate on the elements of the following 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
    print(x)
```

Output:

```
[1 2 3]
[4 5 6]
```

Iterate on each scalar element of the 2-D array:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
for x in arr:
    for y in x:
        print(y)
```

Output:

```
1
2
3
4
5
6
```

Example: Iterate down to the scalars

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
    for y in x:
        for z in y:
            print(z)
```

Output:

```
1
2
3
4
5
6
7
8
9
10
11
12
```

Iterating Arrays Using `nditer()`

The function **`nditer()`** is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

Iterating on Each Scalar Element

In basic **`for`** loops, iterating through each scalar of an array we need to use n **`for`** loops which can be difficult to write for arrays with very high dimensionality.

Example:

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
    print(x)
```

Output:

```
1
2
3
4
5
6
7
8
```

Enumerated Iteration Using `ndenumerate()`

Enumeration means mentioning sequence number of somethings one by one. Sometimes we require corresponding index of the element while iterating, the **`ndenumerate()`** method can be used for those usecases.

Example:

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

Output:


```
(0,) 1
(1,) 2
(2,) 3
```

Example:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

Output:

```
(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
```

Numpy Array Methods

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match. To search an array, use the **where()** method.

Example: Find the index where the value is 4.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

Output:

```
(array([3, 5, 6]),)
```

Sorting Arrays

Sorting means putting elements in an *ordered sequence*. *Ordered sequence* is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending. The NumPy ndarray object has a function called **sort()**, that will sort a specified array.

Example:

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

Output:

```
[0 1 2 3]
```

Functions

- Python Functions is a block of statements that return the specific task.
- The idea is to put some commonly or repeatedly done tasks together and make a function so that instead of writing the same code again and again for different inputs, we can do the function calls to reuse code contained in it over and over again.
- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- The functions are broad of two types, user-defined and built-in 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`.

Syntax

```
def functionname( parameters ):
    "function_docstring"
    function_suite
    return [expression]
```

#Creating a Function

In Python a function is defined using the `def` keyword.

Example:

```
def my_function():
    print("Hello from a function")
```

#Calling a Function

To call a function, use the function name followed by parenthesis.

Example:

```
def my_function():  
    print("Hello from a function")  
my_function()
```

Output:



```
Hello
```

#Arguments

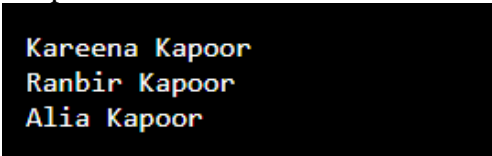
Information can be passed into functions as arguments. Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name.

Example:

```
def my_function(fname):  
    print(fname + " Kapoor")  
my_function("Kareena")  
my_function("Ranbir")  
my_function("Alia")
```

Output:



```
Kareena Kapoor  
Ranbir Kapoor  
Alia Kapoor
```

#Arguments or Parameters

The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function. From a function's perspective: A parameter is the variable listed inside the parentheses in the function definition. And an argument is the value that is sent to the function when it is called.

#Number of Arguments

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Example: This function expects 2 arguments, and get 2 arguments. If you try to call the function with 1 or 3 arguments, you will get an error:

```
def my_function(fname, lname):  
    print(fname + " " + lname)  
my_function("Emma", "Watson")
```

Output:

```
Emma Watson
```

#Arbitrary Arguments, *args

If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition. This way the function will receive a *tuple* of arguments, and can access the items accordingly.

Example: If the number of arguments is unknown, add a * before the parameter name.

```
def my_function(*kids):  
    print("The youngest child is " + kids[2])  
my_function("Emil", "Tobias", "Linus")
```

Output:

```
The youngest child is Linus
```

#Keyword Arguments (kwargs)

You can also send arguments with the *key = value* syntax.

Example:

```
def my_function(child3, child2, child1):  
    print("The youngest child is " + child3)  
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Output:

```
The youngest child is Linus
```

#Arbitrary Keyword Arguments (**kwargs)

If you do not know how many keyword arguments that will be passed into your function, add two asterisk: ** before the parameter name in the function definition. This way the function will receive a *dictionary* of arguments, and can access the items accordingly.

Example:

```
def my_function(**kid):  
    print("His last name is " + kid["lname"])
```

```
my_function(fname = "Tobias", lname = "Refsnes")
```

Output:

```
His last name is Refsnes
```

#Default Parameter Value

If we call the function without argument, it uses the default value.

Example:

```
def my_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

#Return Values

To let a function return a value, use the return statement.

Example:

```
def func(y):
    x = 10
    z = x+y
    return z
sum = func(10)
print("the sum of x and y is:", sum)
```

Output:

```
the sum of x and y is: 20
```

Local and Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables. Global variables can be used by everyone, both inside of functions and outside. If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function.

Example:

```
x = "awesome"                #Global variable
def func():
    x = "fantastic"          #Local variable
    print("Python is " + x)
func()                        #It will access x that is local to func()
print("Python is " + x)      #It will access global x
```

Recursion

The term Recursion can be defined as the process of defining something in terms of itself. In simple words, it is a process in which a function calls itself directly or indirectly.

Example:

```
# Program to print the fibonacci series upto n_terms
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))
```

Output:

Fibonacci series:

```
0
1
1
2
3
5
8
13
21
34
```

Some Programming Exercises

1. Write a program to check a number is even or odd.

```
num = 5
if (num%2 == 0):
    print(num, "is an even number")
else:
    print(num, "is an odd number")
```

2. Write a program to weekdays for the corresponding number.

```
num = 3
if num==1:
    print('Sunday')
elif num == 2:
    print('Monday')
elif num == 3:
    print('Tuesday')
elif num == 4:
    print('Wednesday')
elif num == 5:
    print('Thursday')
elif num == 6:
    print('Friday')
elif num == 7:
    print('Saturday')
```

3. Write a program for printing the star triangle using nested for loop.

```
for i in range(3):
    for j in range(i+1):
        print("*", end = " ")
    print()
```

4. Write a program to check a number is Armstrong or not.

```
num = 153
sum = 0
temp = num
while temp > 0:
    digit = temp % 10
    sum += digit ** 3
    temp //= 10
if num == sum:
```

```
        print(num, "is an armstrong number")
    else:
        print(num, "is not an Armstrong number")
```

5. Write a program to reverse a string.

```
s = "MMMUT"
print(s[::-1])
```

6. Write a program to check a string is whether a palindrome or not.

```
string = "MALAYALAM"
if (string == string[-1:10:-1]):
    print("The string is a palindrome")
else:
    print("The string is not a palindrome")
```

7. Write a program to transpose a matrix.

```
X = [[1,2],
      [3,4]
      [5,6]]
result = [[0,0,0],
           [0,0,0]]
for i in range(len(X)):
    for j in range(len(X[0])):
        result[j][i] = X[i][j]
for r in result:
    print(r)
```

8. Write a program for matrix multiplication in python.

```
X = [[1,2],
      [3,4]
      [5,6]]
Y = [[1,2,3],
      [4,5,6]]
result = [[0,0,0],
           [0,0,0],
           [0,0,0]]
for i in range(len(X)):
    for j in range(len(Y[0])):
        for k in range(len(Y[0])):
            result[j][i] = X[i][j]
for r in result:
```



```
print(r)
```

- 9.** Write a program to print factorial using function.

```
def fact(n):  
    f=1  
    for i in range(1,n+1):  
        f = f*i  
    print("The factorial of",n, "is", f)  
num = 3  
if num<0:  
    print("Sorry, factorial does not exist for negative numbers")  
elif num == 0:  
    print("The factorial of 0 is 1")  
else:  
    fact(num)
```

- 10.** Write a program to print Fibonacci series using recursion.

```
def fib(n):  
    if n <= 1:  
        return n  
    else:  
        return(fib(n-1) + fib(n-2))  
terms = 10  
if terms < =0:  
    print("Please enter positive integers")  
else:  
    print("Fibonacci Sequence:")  
    for i in range(terms):  
        print(fib(i))
```