UNIT III (GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING)

3.1. Boolean values and operators

BOOLEAN VALUES:

Boolean values can be tested for truth value, and used for IF and WHILE condition. There are two values True and False. 0 is considered as False and all other values considered as True.

Boolean Operations:

Consider x=True, y= False

Operator	Example	Description
and	x and y- returns false	Both operand should be true
or	x or y- returns true	Anyone of the operand should be true
not	not x returns false	Not carries single operand

Modulus operator

The **modulus operator** works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
>>> quotient = 7 / 3
>>> print quotient
2

>>> remainder = 7 % 3
>>> print
remainder 1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. For example, you can check whether one number is divisible by another—if x % y is zero, then x is divisible by y.

Also, you can extract the right-most digit or digits from a number. For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```


True and False are special values that belong to the type bool; they are not strings:

```
>>> type(True)
<type 'bool'>

>>> type(False)
<type 'bool'>
```

The == operator is one of the **relational operators**; the others are: x != y # x is not equal to y
x > y # x is greater than y x < y # x is less than y
x >= y # x is greater than or equal to y x <= y # x is less than or equal to y

Logical operators

There are three **logical operators**: and, or, and not. The semantics (meaning) of these operators is similar to their meaning in English. For example, x > 0 and x < 10 is true only if x is greater than 0 and less than 10. n%2 == 0 or n%3 == 0 is true if either of the conditions is true, that is, if the number is divisible by 2 or 3.

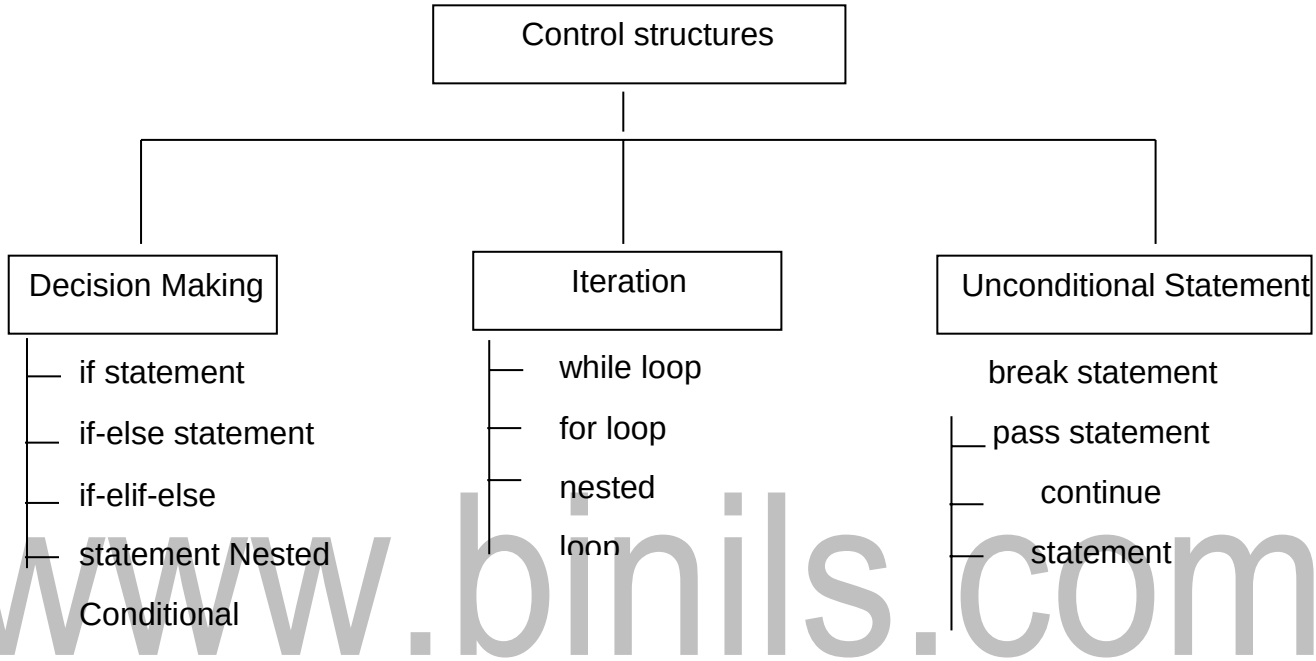
Finally, the not operator negates a boolean expression, so not (x > y) is true if x > y is false, that is, if x is less than or equal to y. The operands of the logical operators should be boolean expressions. Any nonzero number is interpreted as “true.”

```
>>> 17 and True
True
```

UNIT III (GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING)

3.2. Conditional (if), alternative (if-else), chained conditional (if-elif-else)

CONTROL STRUCTURES



Decision Making (or) Conditionals (or) Branching

The execution of the program depends upon the condition. The sequence of the control flow differs from the normal program. The decision making statements evaluate the conditions and produce True or False as outcome.

Types of conditional Statement

1. if statement
2. if-else statement
3. if-elif-else statement
4. Nested Conditional

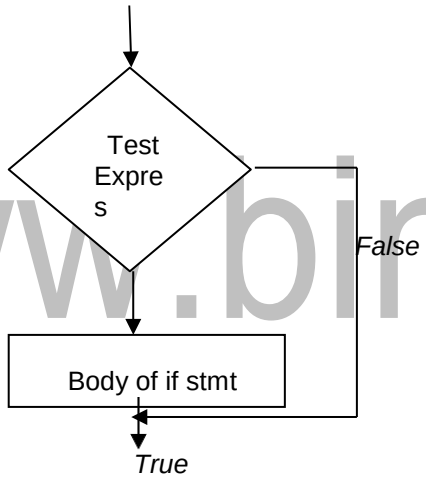
*1.if statement(*conditional)

If statement contains a logical expression using which data is compared and a decision is made based on the result of comparison.

Syntax

```
if <expression>:
        true statements
```

Flow Chart



Example:

```
a=10
if a==10:
        print("a is equal to 10")
```

Result:

a is equal to 10

2. Alternative execution (if… else)

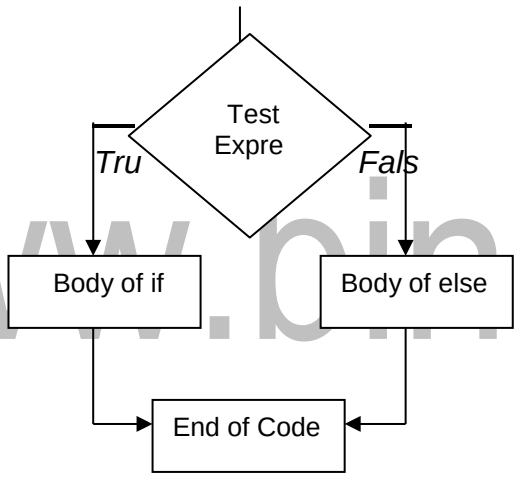
The second form of if statement is “alternative execution” in which there are two possibilities and the condition determines which block of statement executes.

Syntax

```
if
<testexpression>:
        <body_1
        >
```

If the testexpression evaluates to true then <body_1>statements are executed else <body_2> statements are executed.

Flow Chart



Example:

```
a=10
if a==10:
        print(“a is equal to
    10”) else:
        print(“a is not equal to 10”)
```

Result

a is equal to 10

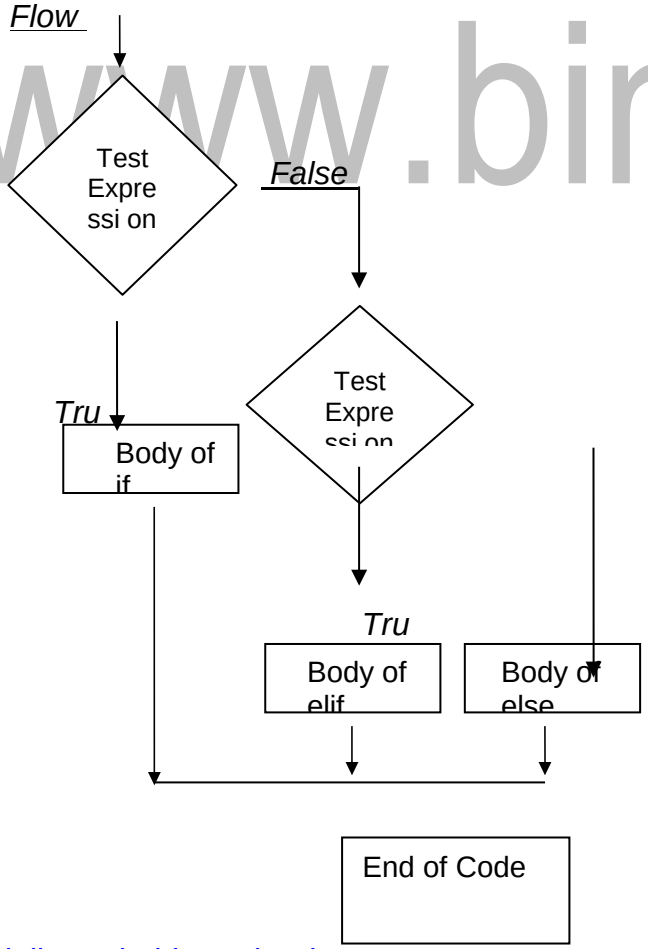
3.elif else Statement(chained

The elif statement or chained conditional allows you to check multiple expressions for true and execute a block of code as soon as one of the conditions evaluates to true. The elif statement has more than one statements and there is only one if condition.

Syntax

```
if
 <bodv
  elif
   <bodv
    elif
     <bodv
```

Flow



Exampl

```
a=
if a==10:
print(“a is equal to
    elif a<10:
        print(“a is lesser than
    10”) elif a>10:
        print(“a is greater than 10”)
   else
        print(“a is not equal to 10”)
```

Result:

a is lesser than 10

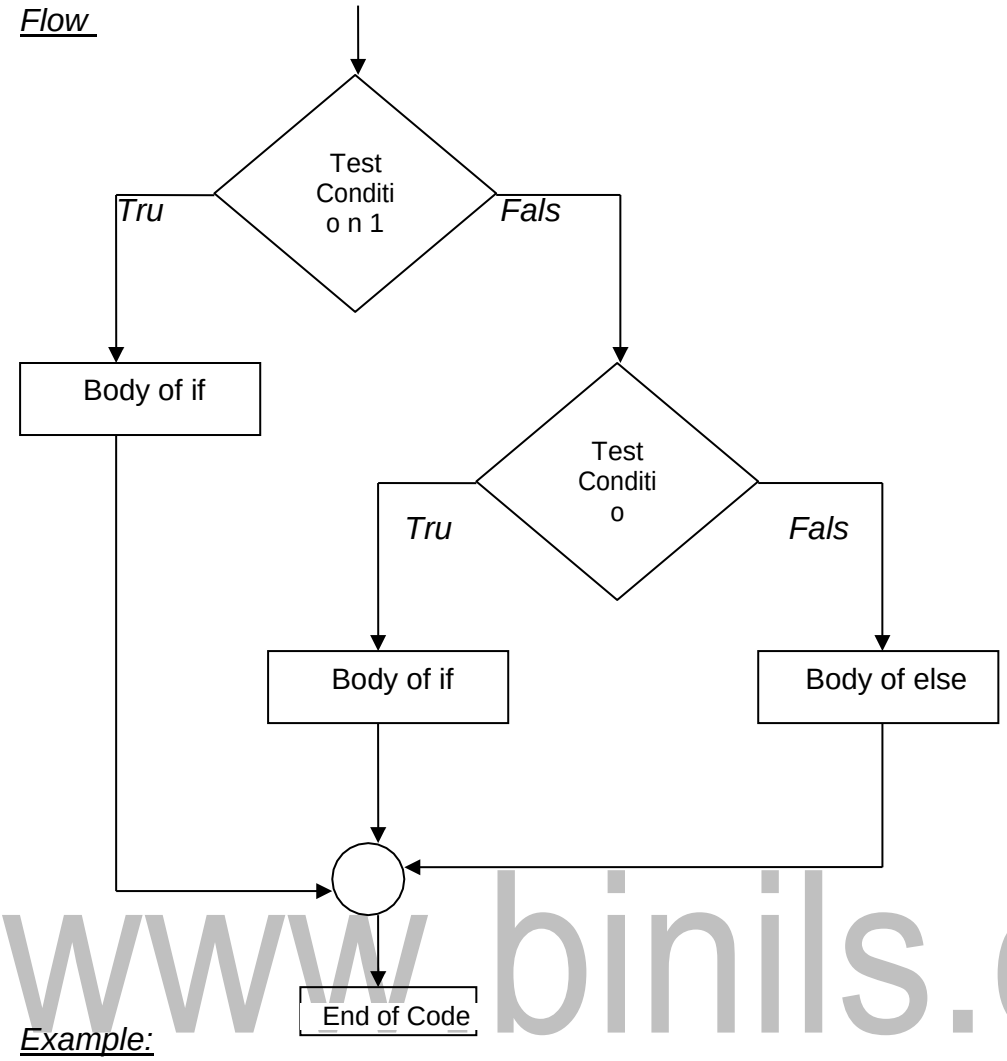
4. Nested Conditionals (or) Nested if-else

One conditional can also be nested with another condition.(ie) we can have if…elif….else statement inside another if …elif…else statements.

Syntax

```
if expression
            true
      else
            if expression 2:
                  true
                  statements
            else:
```

Flow



Example:

```
num = float(input("Enter a number:
")) if num >= 0:
    if num == 0:
        print("Zero")
    else
    :
        print("Positive number")
else
:
    print("Negative
number") Result:
Enter a
number:50
Positive number
```

UNIT III (GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING)

3.4. Fruitful functions: return values, parameters, local and global scope,

function composition, recursion

Fruitful Functions

Function that returns value are called as fruitful functions.The return statement is followed by an expression which is evaluated, its result is returned to the caller as the "fruit" of calling this function.

Input the value→fruitful function→return the result

len(variable) – which takes input as a string or a list and produce the length of string or a list as an output.

In a fruitful function the return statement includes a return value. This statement means: Return immediately from this function and use the following expression as a return value. The expression provided can be arbitrarily complicated, so we could have written this function more concisely:


```
def

return 3.14159 *
```

On the other hand, temporary variables like temp often make debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional.

We have already seen the built-in abs, now we see how to write our own:

```
def
absolute_value(x): if
x < 0:
        return -x
else:
        return x
```

Since these return statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements. Another way to write the above function is to leave out the else and just follow the if condition by the second return statement.

```
def
absolute_value(x): if
x < 0:
        return -
x return x
```

Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. The following version of absolute_value fails to do this:


```
def
 absolute_value(x):
 if x < 0:
        return -
x elif x > 0:
        return x
```

This version is not correct because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. In this case, the return value is a special value called **None**:

```
>>>print absolute
value(0) None
```

None is the unique value of a type called the NoneType:

```
>>>type(None)
```

All Python functions return None whenever they do not return another value.

Example:

Write a python program to find distance between two points:

```
import math
def distance(x1,y1,x2,y2):                    # Defining the Function Distance
    dx=x2-
    x1
    dy=y2-
    y1
    print("The value of dx is",
    dx) print("The value of dy
    is", dy) d= (dx**2 + dy**2)
    dist=math.sqrt(d)
    return dist


x1 = float(input("Enter the first Number: "))        #Getting inputs from  user
x2 = float(input("Enter the Second
Number: ")) y1 = float(input("Enter the third
number: ")) y2 = float(input("Enter the forth
number: "))
print("The distance between two points are",distance(x1,x2,y1,y2))
                                                     #Calling the function distance
```


Output:

```
>>> Enter the first Number: 2
    Enter the Second
    Number: 4 Enter the third
    number: 6 Enter the forth
    number: 12 The value of
    dx is 4.0
    The value of dy is 8.0

   The distance between two points are 8.94427190999916

>>>
```

Explanation for Example 2:

Function Name – ‘distance()’

Function Definition – def

distance(x1,y1,x2,y2) Formal Parameters -

x1, y1, x2, y2

Actual Parameter – dx, dy

Return Keyword – return the output value

'dist' Function Calling –

distance(x1,y1,x2,y2)

Parameter in fruitful function

A function in python

- Take input data,called parameter
- Perform computation
- Return result

```
def
funct(param1,param2):
    statements
    return value
```


Once the function is defined,it can be called from main program or from another function.

Functioncall statement syntax

```
Result=function_name(param1,param
2
```

Parameter is the input data that is sent from one function to another.The parameters are of two types

1.Formal parameter

- The parameter defined as part of the function definition.
- The actual parameter is received by the formal parameter. 2.Actual parameter
- The parameter is defined in the function call Example:

```
def cube(x):
    return x*x*x                    #x is the formal
parameter a=input("Enter the number=")
```

b=cube(a) #a is the actual parameter

print”cube of given

number=”,b Result:

Enter the number=2

Cube of given

number=8

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory.

The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Eg:

```
def my_func():
    x = 10

    print("Value inside function:",x)
x = 20
my_func()
print("Value outside
function:",x) Output:
Value inside function: 10
Value outside function: 20
```

Local Scope and Local Variables

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope** .

```
def func1():
    n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
n in func1 =  10
n in func2 after call to func1 =  20
```

Global Variables and Global Scope

A **global variable** is a variable that is defined outside of any function definition. Such variables are said to have **global scope** .

```
max = 100  # global variable

def func1(count):
  if count < max:
  .

def func2(count):
  for i in range(1, max):
  .
```

global variable max accessed

global variable max accessed

Variable max is defined outside func1 and func2 and therefore "global" to each.

Function Composition

We can call one function from within another. This ability is called composition.

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the **center point** is stored in the variables **xc** and **yc**, and the **perimeter** point is in **xp** and **yp**. The first step is to find the radius of the circle, which is the distance between the two points.

radius = distance(xc, yc, xp, yp)

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
result =
area(radius) return
result
```

Wrapping that up in a function, we get:

```
def area2(xc, yc, xp,

radius = distance(xc, yc, xp,

result = area(radius) return
```

We called this function area2 to distinguish it from the area function defined earlier. There can only be one function with a given name within a given module. The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:


```
def area2(xc, yc, xp,

return area(distance(xc, yc, xp,
```

Example:

Write a python program to add three numbers by using function:

```
def addition(x,y,z):                                    #function 1

        add=x+y+
        z return
        add
def get():                                              #function 2

        a=int(input("Enter first number:"))
        b=int(input("Enter second
        number:")) c=int(input("Enter third
        number:"))
        print("The addition is:",addition(a,b,c))    #Composition function calling

get()                                                   #function calling
```

Output:

Enter first number:5
Enter second
number:10 Enter third
number:15 The
addition is: 30

Recursion:

A Recursive function is the one which calls itself again and again to repeat the code. The recursive function does not check any condition. It executes like normal function definition and the particular function is called again and again

Syntax:

```
def function(parameter):
        #Body of function
```

Example-1:

Write a python program to find factorial of a number using Recursion:

(*Positive value of n ,then n! can be calculated as n!=(n-1)….2.1 it ncan be written as (n-1)! Hence n! is the product of n and (n-1)!* n!=n.(n-1)!)

```
def fact(n):
        if(n<=1):
            return n
        else:
            return n*fact(n-
    1) n=int(input("Enter a
    number:")) print("The
    Factorial is", fact(n))
```

Output:

```
>>> Enter a
    number:5 The
    Factorial is 120
>>>
```

Explanatio

First Iteration - 5*fact(4)

Second Iteration - 5*4*

fact(3) Third Iteration -

5*4*3*fact(2)

Fourth Iteration - 5*4*3*2*

fact(1) Fifth Iteration - 5*4*3*2*1

Example-2:

Write a python program to find the sum of a ‘n’ natural number using

Recursion: def nat(n):

if(n<=1):

return n

else:

return n+nat(n-

1) n=int(input("Enter a

number:")) print("The Sum

is", nat(n))

Output:

>>> Enter a

number: 5 The

Sum is 15

Explanation:

First Iteration – 5+nat(4)

Second Iteration –

5+4+nat(3) Third Iteration –

5+4+3+nat(2)

Fourth Iteration –

5+4+3+2+nat(1) Fifth Iteration

– 5+4+3+2+1

The Advantages of

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

The Disadvantages of recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

UNIT III (GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING)

3.6. Illustrative programs: square root, gcd, exponentiation, sum an array of numbers, linear search, binary search.

ILLUSTRATIVE EXAMPLES

1. Program to find the square root using Newton Method.

```
def newtonSqrt(n,
  howmany): approx = 0.5
  * n
  for i in range(howmany):

    betterapprox = 0.5 * (approx +
    n/approx) approx = betterapprox
  return betterapprox
print(newtonSqrt(10, 3))
print(newtonSqrt(10, 5))

print(newtonSqrt(10, 10))
```


OUTPUT :

Newton Sqrt Value is =.3.16231942215 Newton Sqrt Value is .=3.16227766017 Newton Sqrt Value is .=3.16227766017

2.Program to find the GCD of two numbers

```
d1=int(raw_input("Enter a number:"))
d2=int(raw_input("Enter another
number"))
```

```
rem=d1%d2
while rem!=0 :
  d1=d2
  d2=rem
  rem=d1%d
  2
  print "gcd of given numbers is : %d" %(d2)
```

OUTPUT :

```
Enter a number:54
Enter another number
:24 GCD of given
number is: 6
```


3.Program to find the exponential of a number

```
def power(base,exp):
if(exp==1):
return(base
) if(exp!=1):
    return(base*power(base,exp-1))
base=int(input("Enter base: "))
exp=int(input("Enter exponential value:
")) print("Result:",power(base,exp))
```

OUTPUT :

```
Enter the base:3
Enter exponential
value:2 Result: 9
```

4.Program to find the sum of array of numbers

```
arr = [1, 2, 3, 4, 5];
sum = 0;
for i in range(0, len(arr)):
  sum = sum + arr[i];
   print("Sum of all the elements of an array: " + str(sum));
```

OUTPUT :

Sum of all the elements of an array:15

5.Program to find the maximum and minimum in a list

```
list=[]

print("Enter the limit")
n=int(input())
print("Enter
numbers") for i in
range(0,n):
   a=int(input()
   )
   list.append(
   a)
maxno=list[0
]
minno=list[0]
for i in
   range(len(list)): if
   list[i]>maxno:
      maxno=list[i
                 ] if
     list[i]<minno:
      minno=list[i]
```

```
print("Maximum no of the

print("Minimum no of the
```

OUTPUT :

```
Enter the limit:5

Enter numbers:1 2 3 4

5 Maximum no of the

list:5 Minimum no of

the list:1
```

6.Program to perform the linear search

```
list = []

n=int(input("enter the no of elements in

list")) for i in range(0,n):

  a=int(input("enter the list

  elements")) list.append(a)

x = int(input("Enter number to search:

")) found = False

for i in range(0,n):

 if(list[i] == x):

  found = True

  print("%d found at %d

  position"%(x,i+1)) break

if(found == False):
```

```
 print("%d is not in list"%x)
```

OUTPUT :

```
enter the no of elements in
list:5 enter the list elements:1
2 3 4 5 Enter number to
search:2
2 found at 1 position
```

7.Program to perform Binary search

```
def
  binary_search(list,n,x)
  : start = 0
  end = n - 1
  while(start <=
  end):
    mid = (start +
    end)/2 if (x ==
    list[mid]):
      return mid
    elif(x <
    list[mid]):
      end = mid -
    1 else:
      start = mid +
  1 return -1
n = input("Enter the size of the
list: ") list = []
```

```
for i in range(n):
   list.append(input("Enter %d element:
"%i)) x = input("Enter the number to
search: ") position = binary_search(list, n,
x)
if(position != -1):
   print("Entered number %d is present at position:
%d"%(x,position+1)) else:
   print("Entered number %d is not present in the list"%x)
```

OUTPUT :

```
Enter the size of the
list:5 Enter 1 element:1
Enter 2 element:2
Enter 3 element:3
Enter 4 element:4
Enter 5 element:5
Enter the number to search:7
Entered number 7 is not present in the list
```

ADDITIONAL PROGRAMS

1. **Write a python Program to Check if a Number is Positive, Negative or 0**

Using if...elif...else

```
num = float(input("Enter a number:
")) if num > 0:
        print("Positive
number") elif num == 0:
        print("Zero")
else
:
        print("Negative number")
```

Output:

```
>>> Enter a
   number: 5
   Positive number
>>>
```


Using Nested if

```
num = float(input("Enter a number:
")) if num >= 0:
        if num == 0:
                print("Zero")
        else:
```

```
                print("Positive number")
    else
    :
        print("Negative number")
```

Output:

```
>>> Enter a
    number: 5
    Positive number
>>>
```

2. Write a Python Program to Check a year is Leap Year or not.

```
year = int(input("Enter a year: "))          # To get year (integer input) from the user
if (year % 4) == 0:
    if (year % 100) == 0:
        if (year % 400) == 0:
                print("{0} is a leap year".format(year))
        else
        :
            print("{0} is not a leap year".format(year))
    else:
        print("{0} is a leap year".format(year))
else
:
    print("{0} is not a leap year".format(year))
```


Output:

```
>>>
Enter a year:
2000 2000 is a
leap year
>>>
Enter a year: 1991
1991 is not a leap
year
```

3. Write a Python Program to Print the Fibonacci sequence

```
nterms = int(input("How many terms? "))
                                        # first two terms
n1 = 0
n2 = 1
count = 0
                                        # check if the number of terms is
                                        valid
if nterms <= 0:
        print("Please enter a positive
integer") elif nterms == 1:
         print("Fibonacci sequence
         upto",nterms,":") print(n1)
else
:
         print("Fibonacci sequence upto",nterms,":")
         while count < nterms:          # Starting of While loop
              print(n1,end=' ,
             ') nth = n1 + n2
                                          # update
                                            values
                n1 =
             n2 n2 =
                  nth
             count += 1                 # Ending of While loop
```


Output:

How many terms? 10

Fibonacci sequence upto

10 :

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

>>>

4. Write a Python Program to Check a number is Armstrong Number or not.

```
num = int(input("Enter a number: "))
sum = 0                                   # initialize sum
temp = num                                # find the sum and cube of each
digit
while temp > 0:
        digit  =  temp  %
        10 sum += digit
        ** 3 temp //= 10
if num == sum:                            # display the result
        print(num,"is an Armstrong number")
else
:
        print(num,"is not an Armstrong number")
```


Output:

```
>>> Enter a number: 121
    121 is not an Armstrong number
>>>
```

5. Write a Python Program to Find LCM of two numbers

```
def lcm(x, y):
        if x > y:
                greater = x
        else:
                greater = y
        while(True):
                if((greater % x == 0) and (greater % y == 0)):
```

```
                    lcm =
                    greater
                    break
                greater +=
            1 return lcm
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number:
"))
print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))
```

Output:

```
>>> Enter first number: 10
    Enter second number:
    15
    The L.C.M. of 10 and 15 is 30
>>>
```


6. Write a Python Program to Add Two Matrices

```
X = [[12,7,3],
     [4 ,5,6],
     [7 ,8,9]]
Y = [[5,8,1],
     [6,7,3],
     [4,5,9]]
result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

for i in range(len(X)):                    # iterate through rows

    for j in range(len(X[0])):             # iterate through columns
        result[i][j] = X[i][j] + Y[i][j]
for r in result:
```

```
        print(r)
```

Output:

```
>>>
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
>>>
```

7. Write a Python Program to Transpose a Matrix

```
X = [[12,7],
     [4 ,5],
     [3 ,8]]
result = [[0,0,0],
          [0,0,0]]                    # iterate through rows
for i in range(len(X)):
                                      # iterate through columns
    for j in range(len(X[0])):
        result[j][i] = X[i][j]

for r in result:
    print(r)
```


Output:

```
>>>
[12, 4, 3]
[7, 5, 8]
>>>
```

8. Python Program to Multiply Two Matrices

```
                                                    # 3x3 matrix
X = [[12,7,3],
     [4 ,5,6],
     [7 ,8,9]]
                                                    # 3x4 matrix
Y = [[5,8,1,2],
     [6,7,3,0],
     [4,5,9,1]]
                                                    # result is 3x4
result = [[0,0,0,0],
          [0,0,0,0],
          [0,0,0,0]]
                                                    # iterate through rows of X
for i in range(len(X)):
                                                    # iterate through column  Y
    for j in range(len(Y[0])):
                                                    # iterate through rows of Y
        for k in range(len(Y)):
            result[i][j] += X[i][k] *
            Y[k][j]
for r in result:
    print(r)
```


Output:

```
>>> [114, 160, 60, 27]
    [74, 97, 73, 14]
    [119, 157, 112, 23]
```

9. Write a Python Program to Check Whether a String is Palindrome or Not

```
my_str = 'madame'
```

```
my_str = my_str.casefold()                    # it suitable for caseless
comparison
rev_str = reversed(my_str)                    # reverse the string
if list(my_str) == list(rev_str):             # check the string is equal to its
        reverse print("It is palindrome")
else
:
         print("It is not palindrome")
```

Output:

```
>>>
 It is not palindrome
>>>
```

10. Write a Python Program to count the number of each vowel in a string.

```
vowels = 'aeiou'                              # string of vowels
ip_str = 'Hello, have you tried our turorial section yet?'
                                              # change this value for a different result
ip_str = input("Enter a string:
") ip_str = ip_str.casefold()
                                              # make it suitable for caseless
                                              comparisions
count =
{}.fromkeys(vowels,0)
                                  # make a dictionary with each vowel a key and value 0
for char in ip_str:               # count the
   vowels if char in count:
        count[char] +=
1 print(count)
```

Output:

```
>>>{'o': 5, 'i': 3, 'a': 2, 'e': 5, 'u': 3}
```

UNIT III (GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING)

3.3. Iteration: state, while, for, break, continue, pass

Iteration (or) Looping Statement

An Iterative statement allows us to execute a statement or group of statement multiple times. Repeated execution of a set of statements is called iteration or looping.

Types of Iterative Statement

1. while loop
2. for loop
3. Nested loop

1. while loop

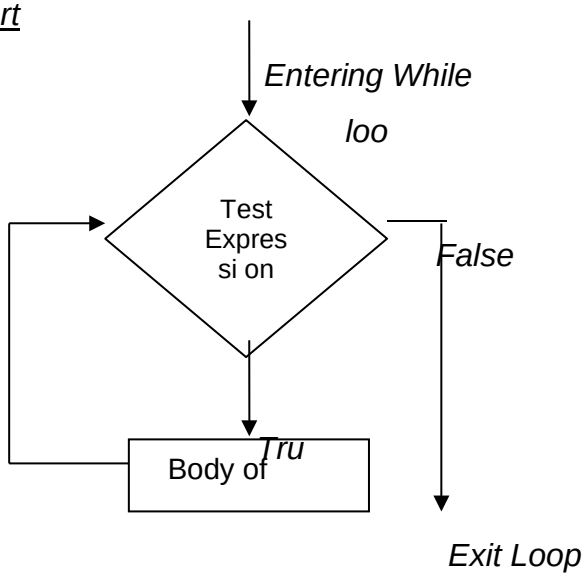
A while loop executes a block of statements again and again until the condition gets false.

The while keyword is followed by test expression and a colon. Following the header is an indented body.

Syntax

```
while expression:
    true statements
```

Flow Chart



Example-1:

1.Write a python program to print the first 100 natural

numbers i=1

while (i<=100):

```
    print(i)
    i=i+1
```

Result:

Print numbers from 1 to 100

Example-2:

2. Write a python program to find factorial of n

```
    numbers. n=int(input("Enter the number:"))
    i=1
    fact=
    1
    while(i<=n):
        fact=fact*
    i i=i+1
    print("The factorial is",fact)
```

Result:

Enter the number:

5 The factorial is

120 ***2. for loop***

The for loop is used to iterate a sequence of elements (list, tuple, string) for a specified number of times.

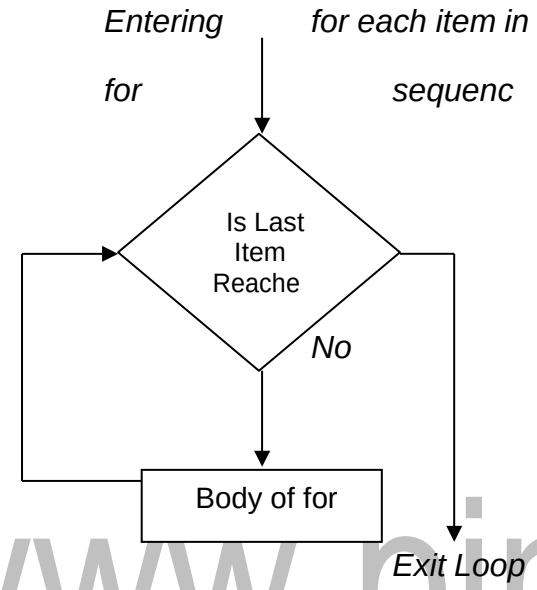
For loop in python starts with the keyword *"for"* followed by an arbitrary variable name,which holds its value in the following sequence objects.

Syntax

```
    for iterating_variable in sequence:
        statements
```

A sequence represents a list or a tuple or a string. The iterating variable takes the first item in the sequence. Next, the statement block is executed. Each item in the list is assigned to the iterating variable and the statements will get executed until the last item in the sequence get assigned.

Flow Chart



Example-1:

```
for letter in"python":
    print("The current letter:", letter)
```

Output:

The current letter: p The current letter: y The current letter: t The current letter: h The current letter: o The current letter: n

Example-2:

```
>>>fruit=['apple', 'orange', 'mango']
>>>for f in fruit:
        print("The current fruit", f)
>>>print("End of for")
```

Output:

The current fruit: apple

The current fruit:

orange The current

fruit: mango End of for

3. *Nested loop*

Python Programming allows using one loop inside another loop. For example using a while loop or a for loop inside of another while or for loop.

Syntax- nested for loop

```
for iterating_variable in sequence:
        for iterating_variable in
                sequence: Innerloop
                statements
Outer loop statements
```


Unconditional Statement

A situation in which need to exit a loop completely when an external condition is triggered or need to skip a part of the loop. In such situation python provide unconditional statements.

Types of Unconditional looping Statement

1. break statement
2. continue statement
3. pass statement

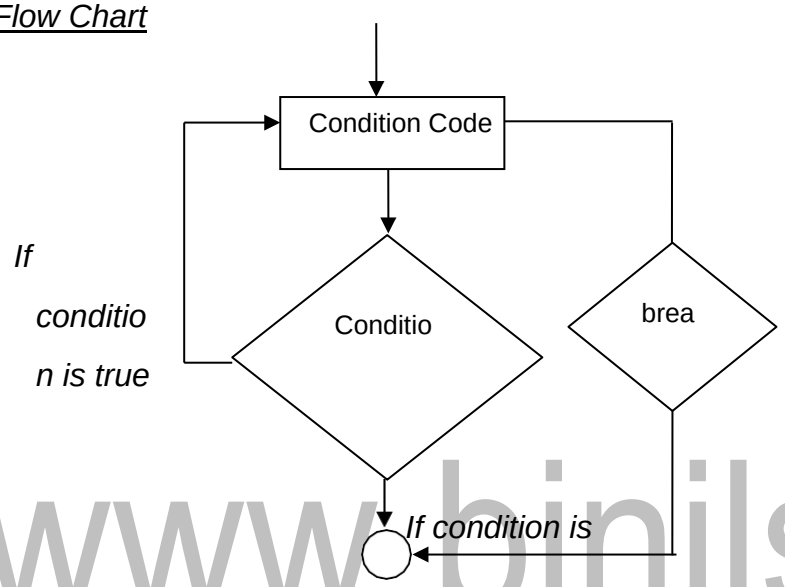
1.break statement

A break statement terminates the current loop and transfers the execution to statement immediately following the loop. The break statement is used when some external condition is triggered.

Syntax

```
break
```

Flow Chart



Example:

```
for letter in"python":
    if letter=='h':
        break
        print(letter)
print("bye")
```

Output:

```
pyt
by
e
```

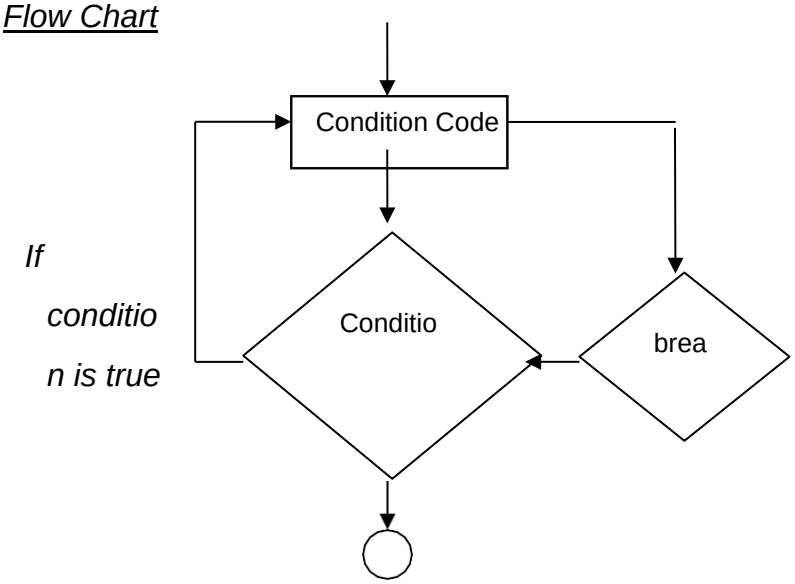
2. continue statement

A continue statement returns the control to the beginning of the loop statement. The continue statement rejects all remaining statement and moves back to the top of the loop.

Syntax

```
continue
```

Flow Chart



If condition is False

Example:

```
for letter in"python":
        if letter=='h':
                continue
                print(letter)
    print("bye")
```


Output:

```
pyton
bye
```

3. pass statement

A pass statement is a null operation, and nothing happens when it executed. It can be used when a statement is required syntactically but the program requires no action.

Syntax

```
pass
```

Example:

```
for letter in"python":
```

```
        if letter==’h’:
                pass
                print(letter)
    print(“bye”)
```

Output:

```
        pytho
        n bye
```

The range() function

If you do need to iterate over a sequence of numbers, the built-in function range() comes in handy. It generates arithmetic progressions:

Eg:

```
# Prints   out    the
           numbers 0,1,2,3,4
for x in range(5):
   print(x)
```


This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

UNIT III (GE8151 PROBLEM SOLVING AND PYTHON PROGRAMMING)

3.5.Strings: string slices, immutability, string functions and methods, string module, Lists as arrays

A string is a **sequence** of characters. We can access the characters one at a time with the bracket operator:

```
>>>  fruit = 'banana'
>>>  letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an **index**. The index indicates which character in the sequence is required .To print the character we use


```
>>> print letter
a
```

For most people, the first letter of 'banana' is b, not a. But for computer scientists, the index is an offset from the beginning of the string, and the offset of the first letter is zero.

```
>>> letter = fruit[0]
>>> print letter
b
```

So b is the 0th letter (“zero-eth”) of 'banana', a is the 1th letter (“one-eth”), and n is the 2th(“two-eth”) letter.

We can use any expression, including variables and operators, as an index, but the value of the index has to be an integer. Otherwise the result will be :

```
>>> letter = fruit[1.5]

TypeError: string indices must be integers, not float
```

Len( )

len is a built-in function that returns the number of characters in a string:

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

To get the last letter of a string, you might be tempted to try something like this:

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```


The reason for the IndexError is that there is no letter in 'banana' with the index 6. Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, you have to subtract 1 from length:

```
>>> last = fruit[length-1]
>>> print
last a
```

Alternatively, you can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

String Slices:

A segment of a string is called a slice. Selecting a slice is similar to selecting a character.

Syntax:

variable [start:stop]

<String_name> [start:stop]

Example

Char	a=	“B	A	N	A	N	A”
Index from Left		0	1	2	3	4	5
Index from Right		-6	-5	-4	-3	-2	-1

>>>print(a[0]) → prints B #Prints B Alone 0th Position

>>>print(a[5]) → prints A #Prints A Alone Last Position

>>>print(a[-4]) → print N #Print From Backwards -4th Position

>>>a[:] → 'BANANA' #Prints All

>>>print(a[1:4]) → print ANA #Print from 1st Position to 4th Position

>>> print(a[1:-2]) → ANA #Prints from 1st position to -3th Position

Strings are immutable

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item
```

The “object” in this case is the string and the “item” is the character you tried to assign. For now, an **object** is the same thing as a value, but we will refine that definition later. An **item** is one of the values in a sequence.

The reason for the error is that strings are **immutable**, which means we can’t change an existing string. The best we can do is create a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

This example concatenates a new first letter onto a slice of greeting. It has no effect on the original string.

String methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the method upper takes a string and returns a new string with all uppercase letters:

Instead of the function syntax upper(word), it uses the method syntax word.upper().

```
>>> word = 'banana'
>>> new_word =
word.upper()

>>> print new_word
BANANA
```

This form of dot notation specifies the name of the method, upper, and the name of the string to apply the method to, word. The empty parentheses indicate that this method takes no argument.

A method call is called an **invocation**; in this case, we would say that we are invoking upper on the word.

As it turns out, there is a string method named find that is remarkably similar to the function we wrote:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print
index 1
```

‘In this example, we invoke find on word and pass the letter we are looking for as a parameter.

Actually, the find method is more general than our function; it can find substrings, not just characters:

```
>>>

word.find('na') 2
```

It can take as a second argument the index where it should start:

```
>>> word.find('na', 3)
4
```

And as a third argument the index where it should stop:

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
-1
```

This search fails because b does not appear in the index range from 1 to 2 (not including 2).

String methods &Descriptions

S.No	Method	Syntax	Description	Example
1.	len()	len(String)	Returns the length of the String	len(a) →5
2.	centre()	centre(width,fullchar)	The String will be centred along with the width specified and the charecters will fill the space	a.centre(20,+) → ++++Hello++++
3.	lower()	String.lower()	Converts all upper case into lower case	b.lower() → hello
4.	upper()	String.upper()	Converts all lower case into upper case	a.upper() → HELLO
5.	capitalize()	String.capitalize()	It converts the first letter into capital	b.capitalize() → HELLO
6.	split()	String.split(“Char”)	splits according to the character which is present inside the function	c.split(“+”) → 1+2+3+4+5
7.	join()	String1.join(String2)	It concatenates the string with the sequence	a.join(b) → Hello hELLO
8.	isalnum()	String.isalnum()	It checks the string is alpha numeric or not. If the string contains 1 or more	d=”a-b” d.isalnum() → returns 1

			alphanumeric characters it returns 1, else its returns 0	
9.	isalpha()	String.isalpha()	Returns true if it has at least 1 or more alphabet characters, else it return false	b.isalpha() →returns 1
10.	isdigit()	String.isdigit()	Returns true if it has at least 1 or more digits, else it return false	b.isdigit() →returns 0
11.	islower()	String.islower()	Returns true if the string has at least 1 or more Lower case characters, else it return false	b.islower( ) returns 1 →
12.	isupper()	String.isupper()	Returns true if the string has at least 1 or more Upper case characters, else it return false	b.isupper() → returns 1
13.	isnumeric()	String.isnumeric()	Returns true if the string contains only numeric character or false otherwise	a.isnumeric() → returns 0
14.	isspace()	String.isspace()	Returns true if the string contains only wide space character or false otherwise	a.isspace () returns 0 → e=”a b” ; e.isspace () returns 1 →
15.	istitle()	String.istitle()	Returns true if the string is properly titled or false otherwise	d=”Hello How R U” d.istitle() →returns 1
16.	isdecimal()	String.isdecimal()	Returns true if the string contains decimal value or false otherwise	c.isdecimal () returns 0 →
17.	title()	String.title()	Returns title cased, all the characters begin with upper case	d=”hello how h u” d.title() → ”Hello How R U”

18.	find()	String.find(String, start,end)	If the string is found it returns index position or it returns -1	a.find("He",0,4) → returns 1 (index position)
19.	endswith()	String.endswith("text", beg, end)	The string will check for whether the character is ending with the specified character. If it found it returns true, else false	a.endswith(i,0) → returns false
20.	index()	String.index('text',beg, end)	It is same as find(). but it raises exception when the string is not found	a.index('i',0) → returns 1
21.	count()	String.count('text',beg, end)	It counts howmany times a string appears	a.count('i',0) → returns 2
22.	rfind()	String.rfind('text',beg, end)	It finds a string from right to left	a.rfind('i') →-1
23.	rindex()	String.rindex('text',be g, end)	Same as index() but moves from right to left	a.rindex('l',0) → returns 3
24.	rjust()	String.rjust(width,str, fi llchar)	It will justify the character into right and fill with the character	a.rjust(10,a,'-')→ -----Hello
25.	ljust()	String.ljust(width,str, fi llchar)	It will justify the character into left and fill with the character	a.ljust(10,a,'+')→ Hello+++++
26.	rstrip()	rstrip()	It removes all the spaces at the end	rstrip(a) → it returns -1
27.	startswith()	startswith(text, beg, end)	It checks whether the character starts with the specified one	a.stratswith(H,0) → returns true

Strings Modules:

This module contains a number of functions to process standard python strings.

Using import string' we can invoke string functions.

Example: Using the string module

```
import string
text = "Monty Python's Flying
Circus" print "upper", "=>",
string.upper(text) print "lower",
"=>", string.lower(text) print "split",
"=>", string.split(text)
print "join", "=>", string.join(string.split(text), "+")
print "replace", "=>", string.replace(text, "Python", "Java")
print "find", "=>", string.find(text, "Python"), string.find(text,
"Java") print "count", "=>", string.count(text, "n")
```

Output:


```
upper => MONTY PYTHON'S FLYING CIRCUS
lower => monty python's flying circus
split => ['Monty', "Python's", 'Flying',
'Circus'] join =>
Monty+Python's+Flying+Circus replace
=> Monty Java's Flying Circus
find => 6 -1
count => 3
```

The in **Operator**

The word in is a boolean operator that takes two strings and returns True if the first appears as a substring in the second:

```
>>> 't' in
'python' True
>>> 'jan' in
'python' False
```

For example, the following function prints all the letters from word1 that also appear in

```
def in_both(word1, word2):
for letter in word1:
if letter in word2:
print(letter)
>>>
in_both('django','mongodb') d
n

g

o
```

List as Array

To store such data, in Python uses the data structure called list (in most programming languages the different term is used — “array”).

Arrays are sequence types and like lists, except that the type of objects stored in them is constrained.

A list (array) is a set of objects.

Individual objects can be accessed using ordered indexes that represent the position of each object within the list (array).

The list can be set manually by enumerating of the elements the list in squarebrackets,

like here: Primes = [2, 3, 5, 7, 11, 13]

Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo',

'Violet'] The list Primes has 6 elements, namely: Primes[0] == 2,

Primes[1] == 3, Primes[2] == 5, Primes[3] == 7,Primes[4] == 11,

Primes[5] == 13.

The list Rainbow has 7 elements, each of which is the string.

Like the characters in the string, the list elements can also have negative index, for example, Primes[-1] == 13,Primes[-6] == 2.

Several ways of creating and

First of all, we can create an empty list and can add items to the end of list using append.

Example

```
a = []                                          # start an empty list
n = int(input('Enter No of Elements'))          # read number of element in the
list for i in range(n):
new_element = int(input('Enter Element :'))     # read next
element a.append(new_element)                   # add it to the list
one:                                            # the last two lines could be replaced by
print(a)                                        # a.append(int(input('Enter Element :')))
```

Result


```
Enter No of
Elements5 Enter
Element :2
Enter Element :7
Enter Element :4
Enter Element :3
Enter Element :8
[2, 7, 4, 3, 8]
```