R-2020

PYTHON PROGRAMMING LAB MANUAL

COLLEGE OF ENGINEERING
(AUTONOMOUS)

Course code: 20IT11S1

Gayatri Vidya Parishad College of Engineering (Autonomous)

Table of Contents

WEEK 1: Input and output

OBJECTIVE:

What is a Variable?

Variables are nothing but reserved memory locations to store values.Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Declaring a Variable:

Python has no command for declaring a variable.

```
 x = 5
y = "John"          # intializing a single value
a = b = c = 1       # intializing a single value to multiple variables.

a,b,c = 1,2,"john" # Multi assignment
```

Reading a value From the Keyboard:

Programs often need to obtain data from the user, usually by way of input from the keyboard. The simplest way to accomplish this in Python is with input().

1. variable= input()

```
      s=input()
```

2. variable= input("message")

```
  y= input("What is your name? ")
>> What is your name?
```

Printing the value on to the Monitor:

1. print ("Message")   # to display a simple message

```
>> print("Hi CSE & IT")
>> Hi CSE & IT
```

2. print(value)   # to display value

```
>> print(9)
>> 9
```

3. print(value1,value2,...,value n)   # to display n-value

```
>> print(1,2,3,4,5)
>>1 2 3 4 5
```

4. print (Variable)   # to display value stored in a variable

```
  >>print (x)
  >> 5
```

5. print("message",variable)   # to display both message and value

```
    >>print("sum is:",sum)
```

```
>>sum is:5
```

1.(A) AIM: Python program to find the largest element among three Numbers.

OBJECTIVE:

Decision Making in Python (if , if..else, Nested if, if-elif)
In real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arises in programming also where we need to make some decisions and based on these decision we will execute the next block of code.
Decision making statements in programming languages decides the direction of flow of program execution. Decision making statements available in python are:

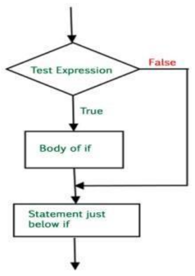
- if statement
- if..else statements
- nested if statements
- if-elif ladder

if statement:

if statement is the most simple decision making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

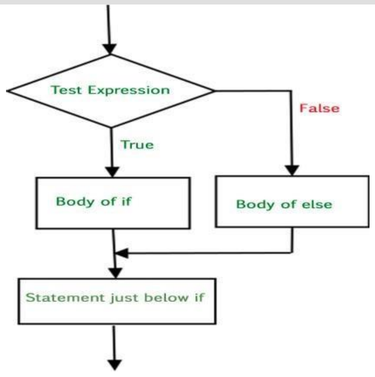
```
if condition:
   statement1
statement2

# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```



if..else statements
The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won’t. But what if we want to do something else if the condition is false. Here comes the *else* statement. We can use the *else* statement with *if* statement to execute ablock of code when the condition is false.

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

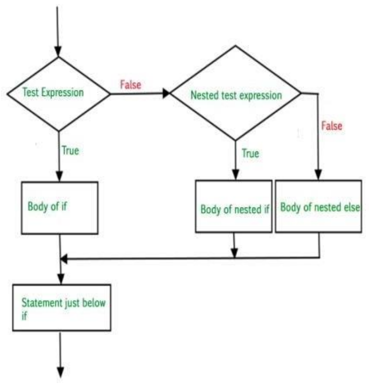


Nested if statements:

A nested if is an if statement that is the target of another if statement. Nested if statements means an if statement inside another if statement. Yes, Python allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

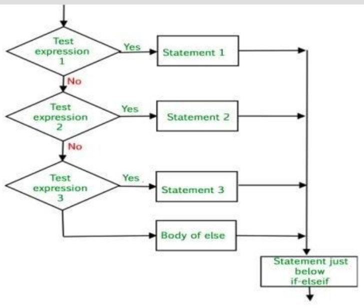


If-elif ladder:

Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:-

```
if (condition):
   statement
elif (condition):
   statement
.
.
else:
   statement
```



SOURCE CODE:

```
n1 = int(input("Enter 1st number: "))
n2 = int(input("Enter 2nd number: "))
n3 = int(input("Enter 3rd number: "))

if (n1 >= n2) and (n1 >= n3):
        lar = n1
elif (n2 >= n1) and (n2 >= n3):
        lar = n2
else:
        lar = n3

print("The largest number among",n1,",",n2,"and",n3,"is: ",lar)
```

OUTPUT:

```
Enter 1st number: 5
Enter 2nd number: 9
Enter 3rd number: 3
The largest number among 5 , 9 and 3 is:  9
```

Test cases:

S.No.	Input	Expected Output	Actual Output	Remarks
1.	1. Enter 1st number: 5 Enter 2nd number: 9 Enter 3rd number: 3 2.Enter 1st number: 8 Enter 2nd number: -9 Enter 3rd number: 9	The largest number among 5 , 9 and 3 is: 9 The largest number among 8 , -9 and 9 is: 9		

1.(B) AIM:Write a python program to print the sum of all the even number in the range 1 - 50 and print their sum.

OBJECTIVE:

While Loop:

```
while expression:
   statement(s)
```

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

For in Loop:
In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is similar to for each loop in other languages.
Syntax:

```
for iterator_var in sequence:
   statements(s)
```

It can be used to iterate over iterators and a range.

Python range() function
range() is a built-in function of Python. It is used when a user needs to perform an action for a specific number of times. range() in Python(3.x) is just a renamed version of a function called xrange in Python(2.x). The **range()** function is used to generate a sequence of numbers. range() is commonly used in for looping hence, knowledge of same is key aspect when dealing with any kind of Python code. Most common use of range() function in Python is to iterate sequence type (List, string etc..) with for and while loop.

Python **range()** Basics :
In simple terms, range() allows user to generate a series of numbers within a given range. Depending on how many arguments user is passing to the function, user can decide where that series of numbers will begin and end as well as how big the difference will be between one number and the next.range() takes mainly three arguments.

- **start**: integer starting from which the sequence of integers is to be returned
- **stop:** integer before which the sequence of integers is to be returned. The range of integers end at stop – 1.
- **step:** integer value which determines the increment between each integer in the sequence

SOURCE CODE:

```
start = int(input("Enter the start of range: "))
end = int(input("Enter the end of range: "))
sum=0
for num in range(start, end + 1):
    if num % 2 == 0:
        sum=sum+num
        print(num, end = " ")
print("\nsum is",sum)
```

OUTPUT:

```
Enter the start of range: 2
Enter the end of range: 10
2 4 6 8 10
sum is 30
```

Test Cases:

S.No.	**Input**	**Expected Output**	**Actual Output**	**Remarks**
1.	1. Enter the start of range: 2 Enter the end of range: 10 2. Enter the start of range: -2 Enter the end of range: 10	2 4 6 8 10 sum is 30 -2 0 2 4 6 8 10 sum is 28		

1.(C)AIM: Python Program to display all prime numbers within an interval of 20 and 50

OBJECTIVE:

Loop Control Statements
Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

SOURCE CODE:

```
start = 20
end = 50

print("Prime numbers between", start, "and", end , "are:")

for num in range(start, end + 1):

  if num > 1:
     for i in range(2, num):
        if (num % i) == 0:
           break
     else:
        print(num)
```

OUTPUT:

```
Prime numbers between 20 and 50 are:
23
29
31
37
41
43
47
```

Test Cases:

S.No.	Input	Expected Output	Actual Output	Remarks
1.	Prime numbers between 20 and 50 are:	23 29 31 37 41 43 47		

WEEK-2: Variables and Functions

Python Functions
Definition:A function is a block of code which only runs when it is called.

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()
```

Arguments
Information can be passed into functions as arguments.
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 + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Parameters or Arguments?
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.
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.

Example
This function expects 2 arguments, and gets 2 arguments:

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

my_function("Emil", "Refsnes")
```

If you try to call the function with 1 or 3 arguments, you will get an error:

Example
This function expects 2 arguments, but gets only 1:

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

my_function("Emil")
```

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")
```

Keyword Arguments
You can also send arguments with the *key* = *value* syntax.
This way the order of the arguments does not matter.

Example

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

my_function(child1 = "Emil", child2 = "Tobias", child3 = "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
If the number of keyword arguments is unknown, add a double ** before the parameter name:

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

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

Return Values
To let a function return a value, use the return statement:

Example
```
def my_function(x):
   return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

The pass Statement
function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

Example
```
def myfunction():
  pass
```

Recursion
Python also accepts function recursion, which means a defined function can call itself.

Example
Recursion Example
```
deftri_recursion(k):
  if(k > 0):
    result = k + tri_recursion(k - 1)
    print(result)
  else:
    result = 0
  return result

print("\n\nRecursion Example Results")
tri_recursion(6)
```

Python - Global Variables

Variables that are created outside of a function are known as global variables.
Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

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. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.
To create a global variable inside a function, you can use the global keyword.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"
```

```
myfunc()

print("Python is " + x)
```

Also, use the global keyword if you want to change a global variable inside a function.

Example
To change the value of a global variable inside a function, refer to the variable by using the global keyword:

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Python Lambda
A lambda function is a small anonymous function.
A lambda function can take any number of arguments, but can only have one expression.

Syntax
lambda *arguments* : *expression*
The expression is executed and the result is returned:

Example
Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Lambda functions can take any number of arguments:

Example
Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Why Use Lambda Functions?
The power of lambda is better shown when you use them as an anonymous function inside another function.
Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

```
def myfunc(n):
  return lambda a : a * n
```

Use that function definition to make a function that always doubles the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
print(mydoubler(11))
```

Or, use the same function definition to make a function that always *triples* the number you send in:

Example

```
def myfunc(n):
  return lambda a : a * n
mytripler = myfunc(3)
print(mytripler(11))
```

Or, use the same function definition to make both functions, in the same program:

Example

```
def myfunc(n):
  return lambda a : a * n
mydoubler = myfunc(2)
mytripler = myfunc(3)
print(mydoubler(11))
print(mytripler(11))
```

2.(A) Write a program to swap two numbers without using a temporary variable.

SOURCE CODE:

```
a=int(input("enter first number"));
b=int(input("enter second number"));
print('Before swapping the values of a, b are',a,b);
a=a+b;
b=a-b;          (or)     (a,b)=(b,a);
a=a-b;
print('After swapping the values of a, b are',a,b);
```

S.No.	Input	Expected Output	Actual Output
1.	2 3	Before swapping the values of a, b are 2,3 After swapping the values of a, b are 3,2	
2.	x y	Before swapping the values of a, b are x,y After swapping the values of a, b are y,x	

2.(B) Write a program to define a function with multiple return values.

SOURCE CODE:

```
def fun():
   a=input("enter first value")
   b=input("enter second value")
return (a,b);
d=fun();
print(d);
```

S.No.	Input	Expected Output	Actual Output
1.	GVP college	("GVP","college")	
2.	10 20 30	(10,20,30)	

2.(C) Write a program to define a function using default arguments.

SOURCE CODE:

```
def greet(college,branch=”CSE"):
print("college name is ",+ college + ', branch name is ' + branch);
a=input(“enter first value”)
b=input(“enter second value”)
greet(a,b);
```

S.No.	**Input**	**Expected Output**	**Actual Output**
1.	GVPCOE IT	College name is GVPCOE , branch name is IT	
2.	Vignan EEE	College name is Vignan , branch name is CSE	

WEEK-3: Loops and conditionals

Variables
Variables are containers for storing data values

Creating Variables:
Python has no command for declaring a variable.
A variable is created the moment you first assign a value to it.

Example
```
x = 5
y = "John"
print(x)
print(y)
```
Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

Example
```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

Casting
If you want to specify the data type of a variable, this can be done with casting.

Example
```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Get the Type
You can get the data type of a variable with the type() function.

Example
```
x = 5
y = "John"
print(type(x))
print(type(y))
```
Single or Double Quotes?
String variables can be declared either by using single or double quotes:

Example
```
x = "John"
# is the same as
x = 'John'
```

Case-Sensitive
Variable names are case-sensitive.

Example
This will create two variables:
a = 4
A = "Sally"
#A will not overwrite a
Python - Variable Names

Variable Names
A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Example
Legal variable names:
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"

Example
Illegal variable names:
2myvar = "John"
my-var = "John"
my var = "John"

Python Conditions and If statements
Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

Example

If statement:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

Indentation

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

Example

If statement, without indentation (will raise an error):

```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Elif

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

Example

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

In this example a is equal to b, so the first condition is not true, but the elif condition is true, so we print to screen that "a and b are equal".

Else

The else keyword catches anything which isn't caught by the preceding conditions.

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
else:
  print("a is greater than b")
```

In this example a is greater than b, so the first condition is not true, also the elif condition is not true, so we go to the else condition and print to screen that "a is greater than b".

You can also have an else without the elif:

Example

```
a = 200
b = 33
if b > a:
  print("b is greater than a")
else:
  print("b is not greater than a")
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

You can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example
Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or
The or keyword is a logical operator, and is used to combine conditional statements:

Example
Test if a is greater than b, OR if a is greater than c:

```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Nested If
You can have if statements inside if statements, this is called *nested* if statements.

Example

```
x = 41
if x > 10:
  print("Above ten,")
  if x > 20:
    print("and also above 20!")
  else:
    print("but not above 20.")
```

The pass Statement
if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error.

Example

```
a = 33
b = 200

if b > a:
  pass
```

Python Loops
Python has two primitive loop commands:

- while loops
- for loops

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

Example
Print i as long as i is less than 6:

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

The while loop requires relevant variables to be ready, in this example we need to define an indexing variable, i, which we set to 1.

The break Statement
With the break statement we can stop the loop even if the while condition is true:

Example
Exit the loop when i is 3:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

The continue Statement
With the continue statement we can stop the current iteration, and continue with the next:

Example
Continue to the next iteration if i is 3:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

The else Statement
With the else statement we can run a block of code once when the condition no longer is true:

Example
Print a message once the condition is false:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Python For Loops

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

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

The range() Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

Example

Using the range() function:

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

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):

Example

Using the start parameter:

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

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**):

Example

Increment the sequence with 3 (default is 1):

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

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example

Print all numbers from 0 to 5, and print a message when the loop has ended:

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

```
else:
  print("Finally finished!")
```

Example

Break the loop when x is 3, and see what happens with the else block:

```
for x in range(6):
  if x == 3: break
  print(x)
else:
  print("Finally finished!")
```

Nested Loops

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

The 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.

3.(A) Write a program to print the following patterns using loop:

```
*
**
***
****
```

SOURCE CODE:

```
a=int(input("enter no.of rows"))
for i in range(1,a+1):
for j in range(1,i+1):
print("*",end="");
print("\r");
```

S.No.	Input	Expected Output	Actual Output
1	2	* **	
2	3	* ** ***	

3.(B) Write a program to print multiplication tables of a given number X1 to range X2.

SOURCE CODE:

```
  x 1=int(input("enter a
value"))
  x 2=int(input("enter
another value"))

for i in range(1,x2+1):
print(x1"x"+str(i)+"="+str(x1*i),end=",")
print("\r");
```

S.No.	Input	Expected Output	Actual Output
1.	8 2	8x1=8,8x2=16,	
2	15 3	15x1=15,15x2=30, 15x3=45,	

WEEK 4: Strings

Strings:
Strings in python are surrounded by either single quotation marks, or double quotation marks.

Example:print (“hello”)   (or)   print(‘hello’)
Assign String to a variable: Assigning a string to a variable is done with the variable name followed by an equal sign and the string.

Example: a=”hello”
print(a)

Multiline Strings: We can assign a multiline string to a variable by using either three double quotes or three single quotes.
Strings in python are treated as an array of characters. We can access the characters of the string with the help of index value, where the first character of the string has the index value is zero.

Example: a=”hello”
print(a[1]) # prints the character **‘e’**

Looping through a string:
We can loop through the characters in a string, using for loop.

Example: for x in “gvpcoe”:
```
        print(x)
```

Example: a=”gvpcoe”
```
for x in a:
        print(x)
```
The above two examplesare giving the same output i.e. each character is printed on a new line.

Slicing Strings:
We can return a range of characters by using the slice syntax. Specify the start index and the end index, separated by a colon, to return a part of the string.
Example: a=”gvpcoe”
print(a[3:6]) # prints **“coe”**

Slice from the start: By leaving out the start index, the range will start at the first character.
Example: a=”gvpcoe”
print(a[:3]) # prints **“gvp”**

Slice to the end: By leaving out the end index, the range will go to the end.
Example: a=”gvpcoe”
print(a[3:]) # prints **“coe”**

Negative Indexing: Use negative indexes to start the slice from the end of the string.
Example: b = "Hello, World!"
print(b[-5:-2]) #prints **"orl"**

String Concatenation: To concatenate or combine two strings we can use the + operator.
Example: a=”gvp”
b=”coe”
print(a+b) # prints **"gvpcoe"**
Note: We cannot combine strings and numbers like the below example:
Example:a=36
txt=”the value is “+a
print(txt)
The above example will give an error i.e. TypeError
But we can combine strings and numbers by using the format() method.

Format Method: This method takes the passed arguments, formats them, and places them in the string where the placeholders {} are.
Example:a=36
txt=”the value is {}”
print(txt.format(a)) # prints **"the value is 36"**

1) This method takes unlimited number of arguments, and is placed into the respective placeholders.
2) We can use index numbers {0} to be sure the arguments are placed in the correct placeholders.

String Methods:

1) **len():**To get the length of a string, use the len() function.

   Example: a=”gvpcoe”
   print(len(a)) # prints the value **"6"**

2) **upper():**This method returns the string in uppercase.

   Example:a=”gvpcoe”
   print(a.upper()) # prints **"GVPCOE"**

3) **lower():**This method returns the string in lowercase.

   Example: a=”GvpCoe”
   print(a.lower()) #prints **"gvpcoe"**

4) **strip():**This method removes any whitespace from the beginning or the end.

5) **replace():** This method replaces a string with another string.

   Example:a=”gvpcoe”

print(a.replace(“g”,”G”)) # prints “**Gvpcoe”**

6) **split():**This method returns a list where the text between the specified separator becomes the list items.

   Example:a=”gvp,college,of,engineering”
    b=a.split(“,”)
   print(b) # prints **[“gvp”,”college”,”of”,”engineering”]**

7) **index():** This method finds the index of the first occurrence of the specified value. If the value is not found, it will raise an exception.

   Example:a=”hello”
   print(a.index(“l”)) #prints **“2”**

   Note:find() and index() method are almost same, but the only difference is that the find() returns -1 if the value is not found.

4.(A) Write a program to find the length of the string without using any library functions.

SOURCE CODE:

```
text=input() # enter any string
length=0
for i in text:
        length=length+1
print(length)
```

S.No.	**Input**	**Expected Output**	**Actual Output**
1.	CSE	The length of the given string is 3	
2.	GVP College	The length of the given string is 11	

4.(B)Write a program to check if two strings are anagrams or not.

SOURCE CODE:

```
def anagrams(s1,s2):
        l1=len(s1)
        l2=len(s2)

        if l1!=l2:
                return 0
        else:
                s1=sorted(s1)
                s2=sorted(s2)
for i in range(0,l1):
                if s1[i]!=s2[i]:
                        return 0
return 1
s1=input() #enter first string
s2=input() #enter second string
a=anagrams(s1,s2)
if a==0:
                print("The given strings are not anagrams");
else:
                print("The given strings are anagrams")
```

S.No.	Input	Expected Output	Actual Output
1	Listen silent	The given strings are anagrams	
2	Dad bad	The given strings are not anagrams	

4.(C) Write a program to check if a substring is present in a given string or not.(use regular expressions)

SOURCE CODE:

```
import re
main=input() #enter a main string
sub=input() #enter a sub string
ifre.search(sub,main):
        print("substring is present in a given string")
else:
        print("substring is not present in a given string")
```

S.No.	Input	Expected Output	Actual Output
1	gvpcollege college	substring is present in a given string	
2	Computer Science CSE	substring is not present in a given string	

WEEK 5: LISTS

Lists
Lists are used to store multiple items in a single variable.

Creation of list: Lists are created using square brackets.
Example:l1=[“cse”,”eee”,”ece”]
print(l1)
In the above example cse,ece,eee are called as items.
Note: List items are ordered, changeable, and allow duplicate values. List items are indexedand can be of any data type.

List length: To determine how many items a list has, use the len() function.
Example:a=[10,”ram”,88.50]
print(len(a)) # prints **3**

type(): Data type of a list is <class ‘list’>
Example:print(type(a))
list(): We can also create a new list by using list() constructor.
Example:a=list((10,”ram”,88.50))

Access List Items: List items are indexed and you can access them by referring to the index number.
Example: a=[“apple”,”banana”,”orange”]
print(a[1]) # prints **“banana”**

Negative Indexing: We can access the list items by using negative indexing which starts from the end.
Example:a=[“apple”,”banana”,”orange”]
print(a[-1])# print the last item of the list i.e. **“orange”**

Range of indexes:we can specify a range of indexes by specifying where to start and where to end the range.
Example:thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(thislist[2:5])
#This will return the items from position 2 to 5.
#Remember that the first item is position 0,
#and note that the item in position 5 is NOT included
Note: By leaving out the start value, the range will start at the first item. Similarly if we leave out the end value, the range will go on to the end of the list.

Change List Items:

Change item value: To change the value of a specific item, refer to the index number.

Example:thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"   #changing the second element in the list.
print(thislist)

Change a range of item values: To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.
Example:thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)# **["apple", "blackcurrant", "watermelon", "orange", "kiwi", "mango"]**

Note1: If you insert *more* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example:thislist = ["apple", "banana", "cherry"]

thislist[1:2] = ["blackcurrant", "watermelon"]

print(thislist)# ["apple", "blackcurrant", "watermelon","cherry"]

Note2: If you insert *less* items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly.

Example:thislist = ["apple", "banana", "cherry"]

thislist[1:3] = ["watermelon"]

print(thislist)# ["apple", "watermelon”]

Looping through Lists:

1) **Using for loop:**

   Example:thislist = ["apple", "banana", "cherry"]

   for x in thislist:print(x)

2) **Using index numbers:** Use the range() and len() functions to create a suitable iterable.

   Example:thislist = ["apple", "banana", "cherry"]
   for i in range(len(thislist)):
   print(thislist[i])

3) **Using a while loop:**

   Example:thislist = ["apple", "banana", "cherry"]

```
i = 0
while i <len(thislist):
print(thislist[i])
i = i + 1
```

Sort Lists: List objects have a sort method that will sort the list alphanumerically, ascending, by default.

Example:thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)

Note: Sort the list in desceding order with the help of keyword argument reverse=True

Example:thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort(reverse = True)
print(thislist)

By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters.

We can perform a case insensitive sort of the list using str.lower as a key function.

Example:thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.sort(key = str.lower)
print(thislist)

Copy Lists: Below are the two ways to make a copy.

1) **Using copy() method:**

   Example:thislist = ["apple", "banana", "cherry"]
   mylist = thislist.copy()
   print(mylist)

2) **Using list() method:**

   Example:thislist = ["apple", "banana", "cherry"]
   mylist = list(thislist)
   print(mylist)

Join Lists:

There are several ways to join or concatenate two or more lists in python.

1) **Using '+' operator:**

Example: list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3) # prints **[“a”,”b”,”c”,1,2,3]**

2) Using append() method:

Example:list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
for x in list2:
 list1.append(x)
print(list1) #prints **[“a”,”b”,”c”,1,2,3]**

3) Using extend() method:

Example:list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1) #prints **[“a”,”b”,”c”,1,2,3]**

List methods:

1) **append():**To add an item to the end of the list.
Example:thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)

2) **insert():**To insert a list item at a specified index.
Example:thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)

3) **remove():**Removes the specified item.
Example:thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)

4) **pop():**Removes the specified index.
Example:thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)

Note: If you do not specify the index, the pop() method removes the last item.

5) **Clear():**Empties the list.The list still remains, but it has no content.
Example:thislist = ["apple", "banana", "cherry"]

thislist.clear()
print(thislist)

6) **count():**Returns the number of elements with the specified value.
Syntax:list.count(value)
Example: fruits = [1, 4, 2, 9, 7, 8, 9, 3, 1]
x = fruits.count(9)
print(x)

7) **index():**Returns the position at the first occurrence of the specified value.
Example: fruits = [4, 55, 64, 32, 16, 32]
x = fruits.index(32)
print(x)

List Comprehension:

List Comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Syntax:
newlist=[expression for item in iterable if condition==True]
The return value is a new list, leaving the old list unchanged.
Condition: It is like a filter that only accepts the item that valuates to true.
Iterable: It can be any iterable object like a list, tuple, set etc.
Expression: The expression is the current item in the iteration, which you can manipulate before it ends up like a list item in the new list.
Example: fruits=[“apple”,”banana”,”cherry”,”kiwi”,”mango”]
newlist=[x for x in fruits if “a” in x]
print(newlist)

Iterators in Python:
Iterators are objects that can be iterated upon.An object which will return data, one element at a time.
Iterator object must implement two methods: __iter__() and __next_()
The___iter_() function returns an iterator from them.
The __next_() method must return the next item in the sequence.On reaching the end, and in subsequent calls, it must raise stopIteration.
Syntax:next(obj) is same as obj.__next__()
Example:
l1=[1,2,3]
i=iter(l1)
print(next(i)) #prints **1**
print(i._next_()) #prints **2**

Generators:

Generator in python is special routine that can be used to control the iteration behaviour of a loop. A generator is similar to a function returning an array. A generator has parameter, whichwe can called and it generates a sequence of numbers. But unlike functions, which return a wholearray, a generator yields one value at a time which requires less memory.
Any python function with a keyword “yield” may be called as generator. In case of generator when it encounters a yield keyword the state of the function is frozen and all the variables are stored in memory until the generator is called again.

Generators with iterators:

```
defgenerator_thr_iter():
yield 'xyz'
yield 246
yield 40.50
for i in generator_thr_iter():
print(i)

Output:
‘xyz’
246
40.50
```

Generator using next:

```
defgenerator_thr_iter():
        yield 'xyz'
yield 246
        yield 40.50
g= generator_thr_iter()
g.__next__()

Output:
‘xyz’
```

Exception Handling:
Errors that occur at run time are called exceptions or logical errors.
For instance, they occur when we try to open a file (for reading) that does not exists (File not found error),try to divide a number by zero (Zero Division Error). Whenever these types of run time errors occur, python creates an exception object.

Catching exceptions in python:
Exceptions can be handled using a try statement.
The critical operation which can raise an exception is placed inside the try clause. The code that handles the exceptions is written in the except clause.

Example:

```
try:
print(x)
except:
print("An exception occurred")
```

Output: An exception occurred

Note: You can define as many exception blocks as you want.
Example:

```
try:
print(x)
exceptNameError:
print("Variable x is not defined")
except:
print("Something else went wrong")
```

Output: Variable x is not define
Else:
You can use the else keyword to define a block of code to be executed if no errors were raised.

Example:
```
try:
print("Hello")
except:
print("Something went wrong")
else:
print("Nothing went wrong")
```

Output:
```
Hello
Nothing went wrong
```

Finally: The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example:
```
try:
print(x)
except:
print("Something went wrong")
finally:
print("The 'try except' is finished")
```

Output:
```
Something went wrong
The ‘try except’ is finished
```

Raise:
The raise keyword is used to raise an exception.
You can define what kind of error to raise, and the text to print to the user.

5.(A) Write a program to perform the given operations on a list:

add ii. Insert iii. Slicing

SOURCE CODE:

```
l1=[]
l2=[]
n1=int(input()) #No.of elements you want to add in list1
n2=int(input()) #No.of elements you want to add in list2
#Enter the elements of the list1
for i in range(0,n1):
ele=input()
l1.append(ele)
#Enter the elements of the list2
for i in range(0,n2):
ele=input()
l2.append(ele)
l1=l1+l2
print("Adding list1 and list2 results in: "+str(l1))
pos=int(input())#Enter the position of element you want to insert in either list1 or list2
ele=(input()) #Enter the element you want to insert
l1.insert(pos,ele)
print("inserting "+ str(ele)+"at position "+str(pos)+"results in: "+str(l1))
start,end=input().split(" ")# "enter the start and end index for the slicing operation
start,end=int(start),int(end)
print("slicing the list from index "+str(start)+" till index "+str(end)+" exclusive "+str(l1[start:end]))
```

S.No.	Input	Expected Output	Actual Output
1	2 3 10 20 30 40 50 2 25 2 5	Adding list1 and list2 results in: ['10', '20', '30', '40', '50'] inserting 25at position 2results in: ['10', '20', '25', '30', '40', '50'] slicing the list from index 2 till index 5 exclusive ['25', '30', '40']	

5.(B) Write a program to perform any 5 built-in functions by taking any list.

SOURCE CODE:

```
list1=[]
n1=int(input())#No.of elements you want to add in list1
#Enter the elements of the list1
for i in range(0,n1):
ele=int(input())
list1.append(ele)
c=int(input())#Enter the element you want to count in the list
print("List is “+str(list1))
print("Length of list is “+ str(len(list1)))
print("Sum of list is “+str(sum(list1)))
print("Max of list is ”+str(max(list1)))
print("Min of list is “+str(min(list1)))
print("Count of “+str(c)+” in list is “+str(list1.count(c)))
```

OUTPUT:

S.No.	Input	Expected Output	Actual Output
1.	5 10 20 30 40 20 20	List is [10, 20, 30, 40, 20] Length of list is 5 Sum of list is 120 Max of list is 40 Min of list is 10 Count of 20 in list is 2	

5.(C) Write a program to get a list of even numbers from a given list of numbers.(use only comprehensions).

SOURCE CODE:

```
arr = list(map(int, input().split())) #Enter numbers separated by space
evenlist = [i for i in arr if i%2==0]
print(evenlist)
```

S.No.	Input	Expected Output	Actual Output
1.	1 2 3 4 5 6 7	[2,4,6]	

WEEK 6: TUPLES

Tuples:

Tuples are used to store multiple items in a single variable. A tuple is a collection which is ordered and unchangeable.

Ex:
```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

```
('apple', 'banana', 'cherry')
```

Tuple items are ordered, unchangeable, and allow duplicate values.
Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

```
('apple', 'banana', 'cherry', 'apple', 'cherry')
```

Tuple Length
To determine how many items a tuple has, use the len() function:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

```
3
```

Create Tuple With One Item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

```
thistuple = ("apple",)
print(type(thistuple))
```

```
<class 'tuple'>
```

```
#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

```
<class 'str'>
```

Tuple Items - Data Types
Tuple items can be of any data type

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
tuple1 = ("abc", 34, True, 40, "male")
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

```
<class 'tuple'>
```

The tuple() Constructor
It is also possible to use the tuple() constructor to make a tuple.

```
thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets
print(thistuple)
```

```
('apple', 'banana', 'cherry')
```

Access Tuple Items
You can access tuple items by referring to the index number, inside square brackets:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

```
banana
```

Negative Indexing
Negative indexing means start from the end.
-1 refers to the last item, -2 refers to the second last item etc

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

```
cherry
```

Range of Indexes
You can specify a range of indexes by specifying where to start and where to end the range.
When specifying a range, the return value will be a new tuple with the specified items.

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

```
('cherry', 'orange', 'kiwi')
```

Change Tuple Values
Once a tuple is created, you cannot change its values. Tuples are unchangeable, immutable.
But yYou can convert the tuple into a list, change the list,and convert the list back into a tuple.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
print(x)
```

Add Items
Once a tuple is created, you cannot add items to it.

```
thistuple = ("apple", "banana", "cherry")
thistuple.append("orange") # This will raise an error
print(thistuple)
```

Remove Items
You cannot remove items in a tuple.Tuples are unchangeable, so you cannot remove items from it,but you can use the same workaround as we used for changing and adding tuple items:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
print(thistuple)
('banana', 'cherry')
```

The del keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
delthistuple
```

6.(A) Write a program to create tuples (name, age, address, college) for at least two members and concatenate the tuples and print the concatenated tuples

SOURCE CODE:

```
person1 = ("Himanshu", 24, "234 LosAngels America", "LosAngelsCollege")
person2 = ("Rohith", 23, "34/56 BigStreetBanglore", "IIMB")
print(f"Tuple1 is {person1}")
print(f"Tuple2 is {person2}")
persons = person1 + person2
```

```
print(f"Concatenate Tuple is {persons}")
```

OUTPUT:

```
Tuple1 is ('Himanshu', 24, '234 LosAngels America', 'LosAngelsCollege')
Tuple2 is ('Rohith', 23, '34/56 BigStreetBanglore', 'IIMB')
Concatenate Tuple is ('Himanshu', 24, '234 LosAngels America', 'LosAngelsCollege', 'Rohith',
23, '34/56 BigStreetBanglore', 'IIMB')
```

6.(B) Write a program to return the top 'n' most frequently occurring chars and their respective counts.e.g. aaaaaabbbbcccc, 2 should return [(a 5) (b 4)]

SOURCE CODE:

```
import operator
defchar_frequency(str1):
dict = {}
keys = dict.keys()
for n in str1:
if n in keys:
dict[n] += 1
else:
dict[n] = 1
returndict

x=char_frequency(input('enter a string'))
d=sorted(x.items(),key=operator.itemgetter(1),reverse=True)
for i in range(int(input('enter n value to retrieve top n elements'))):
print(d[i])
```

OUTPUT:

```
enter a stringGayatriVidyaParishad
enter n value to retrieve top n elements3
('a', 5)
('i', 3)
(' ', 3)
```

S.No.	Input	Expected Output	Actual Output	Remarks
1	aaaaaabbbbcccc	('a', 6) ('b', 4)		
2	hello how are you	('o', 3) (' ', 3)		

WEEK 7: SETS

Sets:
Sets are used to store multiple items in a single variable.Set items are unordered, unchangeable, and do not allow duplicate values.

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

```
{'apple', 'banana', 'cherry'}
```

Set items can be of any data type
set1 = {"abc", 34, True, 40, "male"}
sets are defined as objects with the data type 'set'
print(type(set1))

```
<class 'set'>
```

set() Constructor
Set can also be created using the set() constructor

```
thisset = set(("apple", "banana", "cherry")) # Use double round-brackets
print(thisset)
```

```
{'apple', 'banana', 'cherry'}
```

Access Items
set items can be accessed using a for loop,it can be checked if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
print(x)
```

```
apple
banana
cherry
```

```
print("bananas" in thisset)
```

```
False
```

Change Items
Once a set is created, you cannot change its items, but you can add new items.
Add Items
To add one item to a set use the add() method.
To add items from another set into the current set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)

{'cherry', 'apple', 'orange', 'banana'}
```

Remove Item
To remove an item in a set, use the remove(), or the discard() method

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("orange")
```

If the item to remove does not exist, remove( ) will raise an error, but discard() will not raise an error.

```
thisset = {"apple", "banana", "cherry"}
thisset.discard("orange")
print(thisset)
```

pop() method to remove the last item. Since sets are unordered,you will not know what item that gets removed.

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)

apple
{'banana', 'cherry'}
```

clear() method empties the set

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)

set()
```

del keyword will delete the set completely

```
thisset = {"apple", "banana", "cherry"}
delthisset
```

7.(A) Write a program to count the number of vowels in a string (No control flow allowed).

SOURCE CODE:

```
vowels = "aeiouAEIOU"
s = input("Enter a string: ")
count = sum([1 for i in s if i in vowels])
print("The number of vowels in "+s+" is "+str(count))
```

OUTPUT:

```
Enter a string: gayatri college
The number of vowels in gayatri college is 6
```

S.No.	Input	Expected Output	Actual Output	Remarks
1	python programming	The number of vowels in python programming is4		
2	the city of destiny	The number of vowels in the city of destiny is 5		

7.(B) Write a program that displays which letters are present in both strings.

SOURCE CODE:

```
s1 = input("Enter string1: ")
s2 = input("Enter string2: ")
common =  set(s1).intersection(set(s2))
print(f"Common letters for above strings : {common}")
```

OUTPUT:

```
Enter string1: lemon
Enter string2: orange
Common letters for above strings : {'e', 'o', 'n'}
```

S.No.	Input	Expected Output	Actual Output	Remarks
1	Str1: engineering Str2: college	{'g', 'e'}		
2	Str1: MercedesBenz Str2: Royce Rolls	{'s', 'c', 'e'}		

7.(C) Write a program to sort a given list of strings in the order of their vowel counts.

SOURCE CODE:

```
vowels = "aeiouAEIOU"
defvowelcount(x):
    p = sum([1 for i in x if i in vowels])
return p

s = input("Enter multiple strings :").split()
print(sorted(s,key=vowelcount, reverse=True))
```

OUTPUT:

```
Enter multiple strings :this is best engineering college
['engineering', 'college', 'this', 'is', 'best']
```

S.No.	Input	Expected Output	Actual Output	Remarks
1	python is an easy programming language	['language', 'programming', 'easy', 'python', 'is', 'an']		
2	singapore is a beautiful country	['beautiful', 'singapore', 'country', 'is', 'a']		

WEEK 8: DICTIONARIES

OBJECTIVE:

What is a Dictionaries?
Dictionaries are used to store data values in key:value pairs.
A dictionary is a collection which is unordered, changeable and does not allow duplicates.
Dictionaries are written with curly brackets, and have keys and values:

Example:
Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary Items:
Dictionary items are unordered, changeable, and does not allow duplicates.
Dictionary items are presented in key:value pairs, and can be referred to by using the key name.
Unordered
When we say that dictionaries are unordered, it means that the items does not have a defined order, you cannot refer to an item by using an index.
Changeable
Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
Duplicates Not Allowed
Dictionaries cannot have two items with the same key:

Example:
Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Dictionary Length:
To determine how many items a dictionary has, use the len() function:
Example:
Print the number of items in the dictionary:
```
print(len(thisdict))
```
Dictionary Items - Data Types
The values in dictionary items can be of any data type:

Example:
String, int, boolean, and list data types:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

type()
From Python's perspective, dictionaries are defined as objects with the data type 'dict':

```
<class 'dict'>
```
Example
Print the data type of a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(type(thisdict))
```

Accessing Items:
You can access the items of a dictionary by referring to its key name, inside square brackets:
Example:
Get the value of the "model" key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
```

There is also a method called get() that will give you the same result:

Example:
Get the value of the "model" key:

```
x = thisdict.get("model")
```

Get Keys:
The keys() method will return a list of all the keys in the dictionary.
Example:
Get a list of the keys:

```
x = thisdict.keys()
```

The list of the keys is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example:
Add a new item to the original dictionary, and see that the value list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()
print(x) #before the change

car["color"] = "white"
print(x) #after the change
```

Get Values:
The values() method will return a list of all the values in the dictionary.
Example:

Get a list of the values:

```
x = thisdict.values()
```

The list of the values is a view of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.
Example:
Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}
```

```
x = car.values()
print(x) #before the change
car["year"] = 2020
print(x) #after the change
```

Get Items:

The items() method will return each item in a dictionary, as tuples in a list.

Example:

Get a list of the key:value pairs

```
x = thisdict.items()
```

The returned list is a view of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example:

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
 "brand": "Ford",
 "model": "Mustang",
 "year": 1964
 }

 x = car.items()
 print(x) #before the change
 car["year"] = 2020
 print(x) #after the change
```

Check if Key Exists:

To determine if a specified key is present in a dictionary use the in keyword:

Example:

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

Change Values:

You can change the value of a specific item by referring to its key name:

Example:

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

```
thisdict["year"] = 2018
```

Update Dictionary:
The update() method will update the dictionary with the items from the given argument.
The argument must be a dictionary, or an iterable object with key:value pairs.
Example:
Update the "year" of the car by using the update() method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})
```

Adding Items:
Adding an item to the dictionary is done by using a new index key and assigning a value to it:
Example:

```
thisdict = {
"brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["color"] = "red"
print(thisdict)
```

Update Dictionary:
The update() method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.
The argument must be a dictionary, or an iterable object with key:value pairs.
Example:
Add a color item to the dictionary by using the update() method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

Removing Items:
There are several methods to remove items from a dictionary:
Example:
The pop() method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

example:

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

Example:

The delkeyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
delthisdict["model"]
print(thisdict)
```

Example:

The del keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
delthisdict
print(thisdict) #this will cause an error because "thisdict" no longer exists.
```

Example

The clear() method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

8.(A) Write a program to check if a given key exists in a dictionary or not

SOURCE CODE:

```
d ={1:10,2:20,3:30,4:40,5:50,6:60}
defis_key_present(x):
if x in d:
        print('Key is present in the dictionary')
else:
        print('Key is not present in the dictionary')
is_key_present(5)
is_key_present(9)
```

OUTPUT:

```
Key is present in the dictionary
Key is not present in the dictionary
```

Test Cases:

S.No.	Input	Expected Output	Actual Output	Remarks
1.	1.is_key_present(5) 2. is_key_present(9)	Key is present in the dictionary Key is not present in the dictionary		

8.(B) Write a program to add a new key-value pair to an existing dictionary.

Assigning a new key as subscript
We add a new element to the dictionary by using a new key as a subscript and assigning it a value.

SOURCE CODE:

```
CountryCodeDict = {"India": 91, "UK" : 44 , "USA" : 1}
print(CountryCodeDict)
CountryCodeDict["Spain"]= 34
print "After adding"
print(CountryCodeDict)
```

OUTPUT:

```
{'Spain': 34, 'India': 91, 'USA': 1, 'UK': 44}
```

Using the update() method:

The update method directly takes a key-value pair and puts it into the existing dictionary. The key value pair is the argument to the update function. We can also supply multiple key values as shown below.

Example:

```
CountryCodeDict = {"India": 91, "UK" : 44 , "USA" : 1, "Spain" : 34}
print(CountryCodeDict)
CountryCodeDict.update( {'Germany' : 49} )
print(CountryCodeDict)

 # Adding multiple key value pairs
CountryCodeDict.update( [('Austria', 43),('Russia',7)] )
print(CountryCodeDict)
```

Output

Running the above code gives us the following result −

```
{'Spain': 34, 'India': 91, 'USA': 1, 'UK': 44}
 {'Germany': 49, 'Spain': 34, 'India': 91, 'USA': 1, 'UK': 44}
 {'USA': 1, 'India': 91, 'Austria': 43, 'Germany': 49, 'UK': 44, 'Russia': 7, 'Spain':
```

8.(C) Write a Python program to sum all the items in the given dictionary.

SOURCE CODE:

```
my_dict = {'data1':100,'data2':-54,'data3':247}
print(sum(my_dict.values()))
```

OUTPUT:

```
293
```

WEEK 9: FILES

What are the Errors and Exceptions in Python?
Python doesn’t like errors and exceptions and displays its dissatisfaction by terminating the program abruptly.
There are basically two types of errors in the Python language-

Types of errors in the Python language

1. Syntax Error 2. Errors occurring at run-time or Exceptions.

- Syntax Error.
- Errors occuring at run-time or *Exceptions.*

Syntax Errors
Syntax Errors, also known as parsing errors, occur when the parser identifies an incorrect statement. In simple words, syntax error occurs when the proper structure or syntax of the programming language is not followed.
An example of a syntax error:

```
>>>print(1 / 0 ))

File "", line 1
print( 1 / 0 ))
   ^
SyntaxError: invalid syntax
```

Exceptions
Exceptions occur during run-time. Python raises an exception when your code has a correct syntax but it encounters a run-time issue which it is not able to handle.
There are a number of defined built-in exceptions in Python which are used in specific situations.
Some of the built-in exceptions are:
There are another type of built-in exceptions called **warnings**. They are usually issued in situations where the user is alerted of some conditions. The condition does not raise an exception; rather it terminates the program.

Exception	Cause Of Error
ArithmeticError	Raised when numerical computation fails.
FloatingPointError	Raised when floating point calculation fails.
AssertionError	Raised in case of failure of the Assert statement.

ZeroDivisionError	Raised when division or modulo by zero takes place for all numerical values.
OverflowError	Raised when result of an arithmetic operation is very large to be represented.
IndexError	Raised when an index is not found in a sequence.
ImportError	Raised when the imported module is not found.
IndentationError	Raised when indentation is not specified properly.
KeyboardInterrupt	Raised when the user hits interrupt key.
RuntimeError	Raised when a generated error does not fall into any category.
SyntaxError	Raised when there is an error in Python syntax.
IOError	Raised when Python cannot access a file correctly on disk.
KeyError	Raised when a key is not found in a dictionary.
ValueError	Raised when an argument to a function is the right type but not in the right domain.
NameError	Raised when an identifier is not found in the local or global namespace.
TypeError	Raised when an argument to a function is not in the right type.

How to raise Custom Exceptions in Python?
Python comprises of a number of built-in exceptions which you can use in your program. However, when you're developing your own packages, you might need to create your own custom exceptions to increase the flexibility of your program.

You can create a custom Python exception using the pre-defined class **Exception**:

```
defsquare(x):
if x<=0 or y<=0:
raise Exception('x should be positive')
return x * x
```

Here, the function **square** calculates the square of a number. We raise an **Exception** if either the input number is negative or not.

Disadvantages of Exception Handling
Though exception handling is very useful in catching and handling exceptions in Python, it also has several disadvantages. Some of which are as follows—

- It can trap only run-time errors.
- When you use **try-except**, the program will lose some performance and slow down a bit.
- The size of the code increases when you use multiple **try**, **except**, **else** and **finally** blocks.
- The concept of **try-catch** might be a little difficult to understand for beginners.
- It is useful only in exceptional error cases.

Other than these disadvantages, understanding the concept of Exception Handling can ease your career as a programmer in the world of Python.

Files

Handling files in python
Data is often stored in text files, which is organized. There are many kinds of files. Text files, music files, videos, and various word processor and presentation documents are those we are familiar with.

Text files only contain characters whereas, all the other file formats include formatting information that is specific to that file format. Operations performed on the data in files include the read and write operations. To perform any operation the program must open the file. The syntax to open a file is given below:

```
with open(«filename», «mode») as «variable»:
«block»
```

Though there are several ways of opening a file I prefer this way because we need not specify the close statement at the end.

Reading a file:
There are several techniques for reading files. One way is reading the overall contents of the file into a string and we also have iterative techniques in which in each iteration one line of text is read. We, can also read each line of text and store them all in a list. The syntax for each technique is given below

```
#to read the entire contents of text into a single string
with open('file1.txt', 'r') as f:
contents = f.read()
#to read each line and store them as list
with open('file1.txt', 'r') as f:
lines = f.readlines()
#for iterative method of reading text in files
with open('planets.txt', 'r') as f:
for line in f:
print(len(line))
```

As our job is to just read the contents of the file and then finding the most frequent word in a text read from a file we have no space for the write operation. In case you want to learn it go through this link text file in Python

Now let’s get into our job of finding the most frequent words from a text read from a file.

Write a program to find the most frequent words in a text.(read from a text file)
Most frequent words in a text file with Python
First, you have to create a text file and save the text file in the same directory where you will save your python program. Because once you specify the file name for opening it the interpreter searches the file in the same directory of the program. Make sure you have created and saved the file in proper directory.

The algorithm we are going to follow is quite simple first we open the file then we read the contents we will see how many times each word is repeated and store them in a variable called count. Then we check it with the maximum count which is initialized as zero in the beginning.If count is less than maximum count we ignore the word if it is equal we will place it in a list. Otherwise, if it is greater then we clear the list and place this word in the list.

9.(A) Write a program to sort words in a file and put them in another file. The output file should have only lower case words, so any upper case words from a source must be lowered. (Handle exceptions)

SOURCE CODE:

```
ipfile = open("test.txt",'r')
l = ipfile.read().split("\n")
lt = []
for i in l:
for j in i.split():
lt.append(j.lower())
lt.sort()
print(lt)
opfile = open("output.txt", "w")
for i in lt:
opfile.write(str(i)+"\n")
ipfile.close()
opfile.close()
```

OUTPUT:

```
['feel', 'great', 'hail', 'hello', 'i', 'is', 'is', 'lab', 'manual', 'python', 'python', 'python', 'this']
```

Test Cases:

S.No.	Input	Expected Output	Actual Output	Remarks
1	Hi this is lalitha. This day is more precious day	['day', 'day', 'hi', 'is', 'is', 'lalitha.', 'more', 'precious', 'this', 'this']		
2	python is simple and it is an easy language. Python is flexible language	['an', 'and', 'easy', 'flexible', 'is', 'is', 'is', 'it', 'language', 'language.', 'python', 'python', 'simple']		

9.(B) Write a program to find the most frequent words in a text.(read from a text file)

SOURCE CODE:

```
defword_freq(s):
dic ={}
keys = dic.keys()
for n in s:
if n in keys:
dic[n] += 1
else:
dic[n] = 1
returndic

ipfile = open("test.txt",'r')
l = ipfile.read().split("\n")
lt = []
for i in l:
for j in i.split():
lt.append(j.lower())
cf = word_freq(lt)
n = max(list(cf.values()))
print("Most frequently occuring words and it's frequency")
forkey,val in cf.items():
ifval>=n:
print(key,val)
```

OUTPUT:

```
Most frequently occurring words and it's frequency
python 16
```

Test Cases:

S.No.	Input	Expected Output	Actual Output	Remarks
1.	Hi this is lalitha. This day is more precious day	Most frequently occuring words and it's frequency this 2 is 2 day 2		
2.	python is simple and it is an easy language. Python is flexible language	Most frequently occuring words and it's frequency 3		

WEEK 10: CLASSES

OBJECTIVE:

What are classes and objects?

A class is a code template for creating objects. Objects have member variables and have behaviour associated with them. In python a class is created by the keyword class.
An object is created using the constructor of the class. This object will then be called the instance of the class. In Python we create instances in the following manner.

```
 class_instance = class_name(*arguments)
```

How to create classes ?

The simplest class can be created using the class keyword. For example, let's create a simple, empty class with no functionalities.

```
>>> class Snake:
>>>     pass
>>> snake = Snake()
>>> print(snake)
<<<__main__.Snake object at 0x7f315c573550>
```

Attributes and Methods in class: A class by itself is of no use unless there is some functionality associated with it. Functionalities are defined by setting attributes, which act as containers for data and functions related to those attributes. Those functions are called methods.

- **Attributes:** You can define the following class with the name Snake. This class will have an attribute name.

  ```
  >>> class Snake:
  ...     name = "python"
  ...     # set an attribute `name` of the class
  ```

  You can assign the class to a variable. This is called object instantiation. You will then be able to access the attributes that are present inside the class using the dot . operator. For example, in the Snake example, you can access the attribute name of the class Snake.

  ```
  >>> # instantiate the class Snake and assign it to variable snake
  >>> snake = Snake()
  >>> # access the class attribute name inside the class Snake.
  >>> print(snake.name)
  <<< python
  ```

- **Methods:** Once there are attributes that “belong” to the class, you can define functions that will access the class attribute. These functions are called methods. When you define methods, you will need to always provide the first argument to the method with a self keyword. For example, you can define a class Snake, which has one attribute name and one method change_name. The method change name will take in an argument new_name along with the keyword self.

```
>>> class Snake:
...     name = "python"
...     def change_name(self, new_name):
...         self.name = new_name
...     # access the class attribute with the self keyword
...     # Note: first argument of method is self
```

Now, you can instantiate this class Snake with a variable snake and then change the name with the method change_name.

```
>>> # instantiate the class
>>> snake = Snake()
>>> # print the current object name
>>> print(snake.name)
<<< python
>>> # change the name using the change_name method
>>> snake.change_name("anaconda")
>>> print(snake.name)
<<< anaconda
```

- **Instance attributes in python and the init method:** You can also provide the values for the attributes at runtime. This is done by defining the attributes inside the init method. The following example illustrates this.

```
class Snake:
    def__init__(self, name):
        self.name = name
    def change_name(self, new_name):
        self.name = new_name
```

Now you can directly define separate attribute values for separate objects. For example,

```
>>> # two variables are instantiated
>>> python = Snake("python")
>>> anaconda = Snake("anaconda")
>>> # print the names of the two variables
>>> print(python.name)
<<< python
>>> print(anaconda.name)
<<< anaconda
```

10.(A) Write a Python class named Person with attributes name, age, weight (kgs), height (ft) and takes them through the constructor and exposes a method “get_bmi_result()” which returns one of "underweight", "healthy", "obese" values.

SOURCE CODE:

```
class person(object):
    def___init_(self, name, age, height, weight):
        self.name = name
        self.age = age
        self.height = height
        self.weight = weight
    def get_bmi(self):
        bmi = float(self.weight / ((self.height * 0.3048) ** 2))
       return 'Underweight' if (bmi < 18.5) else 'Healthy' if (bmi >= 18.5 and bmi < 25) else 'Obese'
# Above line includes ternary operator in python

p1 = person('Person 1', 25, 6, 30)
print(p1.name, 'is', p1.get_bmi())
p2 = person('Person 2', 25, 6, 200)
print(p2.name, 'is', p2.get_bmi()
```

OUTPUT:

```
Person 1 is Underweight
Person 2 is Obese
```

10.(B) Write a Python class named Circle constructed by a radius and two methods which will compute the area and the perimeter of a circle.

SOURCE CODE :

```
class Circle():
    def___init_(self, r):
        self.radius = r

    def area(self):
        return 3.14 * (self.radius**2)

    def perimeter(self):
        return 2 * 3.14 * self.radius

radius = int(input('Enter radius of circle: '))
circle = Circle(radius)
print(f'Area of circle is {circle.area()}')
print(f'Perimeter of circle is {circle.perimeter()}')
```

OUTPUT :

```
Enter radius of circle: 7
Area of circle is 153.86
Perimeter of circle is 43.96
```

WEEK 11: ARRAYS

OBJECTIVE:

Python Arrays :
Python does not have built-in support for Arrays, but Python Lists can be used instead.

Arrays:
Note: We use LISTS as ARRAYS, however, to work with arrays in Python you will have to import a library, like the NumPy library. Arrays are used to store multiple values in one single variable:

```
# Create an array containing car names:
cars = ["Ford", "Volvo", "BMW"]
```

What is an Array?
An array is a special variable, which can hold more than one value at a time.
If you have a list of items (a list of car names, for example), storing the cars in single variables could look like this.

```
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
# using array
cars = ["Ford", "Volvo", "BMW"]
```

However, what if you want to loop through the cars and find a specific one? And what if you had not 3 cars, but 300? The solution is an array, An Array can hold many values under a single name, and you can access the values by referring to an index number.

Access the Elements of an Array
You refer to an array element by referring to the *index number*.

```
# Get the value of the first array item:
x = cars[0]
# Modify the value of the first array item:
cars[0] = "Toyota"
```

The Length of an Array
Use the len() method to return the length of an array. (No. of elements in an array)

```
x = len(cars) # number of elements in the cars array
```

Note: The length of an array is always one more than the highest array index.

Looping Array Elements
You can use the for in loop to loop through all the elements of an array.

```
for x in cars:
    print(x)
```

Adding Array Elements

You can use the append() method to add an element to an array.

```
# Add one more element to the cars array:
cars.append("Honda")
```

Removing Array Elements

You can use the pop() method to remove an element from the array.

```
#Delete the second element of the cars array:
cars.pop(1)
# You can also use the remove() method
# to remove an element from the array.
#Delete the element that has the value "Volvo":
cars.remove("Volvo")
```

Note: The list's remove() method only removes the first occurrence of the specified value.

Array Methods

Python has a set of built-in methods that you can use on lists/arrays.

Method	**Description**
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

11.(A) Write a program to create, display, append, insert and reverse the order of the items in the array.

SOURCE CODE :

```
# Create array
array = [1, 2, 3, 4, 5]
# display array
print(f'Created Array : {array}')
# Append 6,7 to array
array.append(6)
array.append(7)
print(f'Array after appending is {array}')
# Insert -1,0 at 0,1 places in array
array.insert(0, -1)
array.insert(1, 0)
print(f'Array after inserting is {array}')
# reverse the array
array.reverse()
print(f'Array after reversing is {array}')
```

OUTPUT :

```
        Created Array : [1, 2, 3, 4, 5]
        Array after appending is [1, 2, 3, 4, 5, 6, 7]
        Array after inserting is [-1, 0, 1, 2, 3, 4, 5, 6, 7]
        Array after reversing is [7, 6, 5, 4, 3, 2, 1, 0, -1]
```

11.(B) AIM : Write a program to add, transpose and multiply two matrices.

SOURCE CODE :

```
# input first matrix
mat_a = []
r1 = int(input("Enter rows in matrix a    : "))
c1 = int(input("Enter columns in matrix a : "))
for i in range(0, r1):
    row = list(map(int,input(f"For matrix a, input row no.{i+1}: ").split()))
    mat_a.append(row)

# input second matrix
mat_b = []
r2 = int(input("Enter rows in matrix b    : "))
c2 = int(input("Enter columns in matrix b : "))
for i in range(0, r2):
    row = list(map(int, input(f"For matrix b, input row no.{i+1}: ").split()))
    mat_b.append(row)


# Transpose for matrix-a
transpose_a = [[0 for _ in range(r1)] for _ in range(c1)]
for i in range(r1):
    for j in range(c1):
        transpose_a[j][i] = mat_a[i][j]
# display transpose of matrix-a
print("Transpose of matrix a is :")
for t_a_row in transpose_a:
    print(t_a_row)


# Transpose for matrix-b
transpose_b = [[0 for _ in range(r2)] for _ in range(c2)]
for i in range(r2):
    for j in range(c2):
        transpose_b[j][i] = mat_b[i][j]
# display transpose of matrix-b
print("Transpose of matrix b is :")
for t_b_row in transpose_b:
    print(t_b_row)

# Addition is possible only when dimensions are same
if r1 == r2 and c1 == c2:
    add_result = []
    for i in range(0, r1):
        add_row = []
        for j in range(0, c1):
            add_row.append(mat_a[i][j]+mat_b[i][j])
        add_result.append(add_row)

    print("Addition of given two matrices is: ")
    for result_row in add_result:
        print(result_row)
else:
    print("Addition is not possible for given matrices")


# Multiply matrix
if c1 == r2:
    multiplication_result = [[0 for _ in range(c2)] for _ in range(r1)]
    for i in range(r1):
        for j in range(c2):
            for k in range(r2):
                multiplication_result[i][j] += mat_a[i][k] * mat_b[k][j]
    # Display matrix multiplication result
    print("Multiplication of given two matrices is :")
    for result_row in multiplication_result:
        print(result_row)
else:
    print("Matrix multiplication is not possible")
```

OUTPUT :

```
Enter rows in matrix a    : 3
Enter columns in matrix a : 4
For matrix a, input row no.1: 1 2 3 4
For matrix a, input row no.2: 5 6 7 8
For matrix a, input row no.3: 9 10 11 12
Enter rows in matrix b    : 4
Enter columns in matrix b : 3
For matrix b, input row no.1: 1 2 3
For matrix b, input row no.2: 4 5 6
For matrix b, input row no.3: 7 8 9
For matrix b, input row no.4: 10 11 12
Transpose of matrix a is :
        [1, 5, 9]
        [2, 6, 10]
        [3, 7, 11]
        [4, 8, 12]
Transpose of matrix b is :
        [1, 4, 7, 10]
        [2, 5, 8, 11]
        [3, 6, 9, 12]
Addition is not possible for given matrices
Multiplication of given two matrices is :
        [70, 80, 90]
        [158, 184, 210]
        [246, 288, 330]
```

WEEK 12: PYTHON MAPS ,FILTERS &GENERATORS

MAP

The `map()` function applies a given function to each item of an iterable (list, tuple etc.) and returns an iterator.

map() Syntax

Its syntax is:

map(function , iterable ,……)

map() Parameter

The `map()` function takes two parameters:

- **function** - a function that perform some action to each element of an iterable
- **iterable** - an iterable like sets, lists, tuples, etc

You can pass more than one `iterable` to the `map()` function.

map() Return Value

The `map()` function returns an object of map class. The returned value can be passed to functions like

- list() - to convert to list
- set() - to convert to a set, and so on.

Since `map()` expects a function to be passed in, lambda functions are commonly used while working with `map()` functions.
A lambda function is a short function without a name.

Generator

There is a lot of work in building an iterator in Python. We have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, and raise `StopIteration` when there are no values to be returned.

This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above

are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can

iterate over (one value at a time).

Here is how a generator function differs from a normal function.

- Generator function contains one or more `yield` statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the

  caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Similar to the lambda functions which create anonymous functions, generator expressions create anonymous generator functions.

The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

The major difference between a list comprehension and a generator expression is that a list comprehension produces the entire list while the generator expression produces one item at a time.

They have lazy execution (producing items only when asked for). For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

12.(A) AIM : Accept two lists ,one list represents temperatures in Fahrenheit and another list represents temperatures in Celsius. Perform map operations Fahrenheit-Celsius and Celsius-Fahrenheit using lambda.

```
  def fahrenheit(T):
              return ((float(9)/5)*T + 32)
  def celsius(T):
              return (float(5)/9)*(T-32)
              t = (36.5, 37, 37.5, 38, 39)
f=list(map(fahrenheit, t))
c=list(map(celsius,f))
print(f)
print(c)
d=list(map(lambda x: (float(9)/5)*x + 32, t))
e=list(map(lambda x: (float(5)/9)*(x - 32), f))
print(d)
print(e)
```

Sample output:

```
[97.7 ,98.60000000000001 , 99.5, 100.4, 102.2]
[36.5, 37.0000000000001, 37.5, 38.0000000000001, 39.0]
```

12.(B) AIM : Create a Fibonacci sequence that contains ‘N’ terms and filter only even Terms using lambda.

```
from functools import reduce
fib_series = lambda n: reduce(lambda x, _: x+[x[-1]+x[-2]],
                              range(n-2), [0, 1])
print(fib_series(6))
```

12.(C) AIM: Write a program to find the numbers of rows in a text file using Genrator and yield.

```
def csv_reader(file_name):
        file = open(file_name)
        result = file.read().split("\n")
        return result
csv_gen = csv_reader("sample.txt")
row_count = 0

for row in csv_gen:
        row_count += 1

print(f"Row count is {row_count}")
```

12.(D) AIM: Find the sum of squares of 1 to n numbers using Generator Expresssions.

```
my_list = [1, 3, 6, 10]
list= [x**2 for x in my_list]
print(list)
print(sum(list))
```

Sample output:

```
[1 ,9 , 36, 100]
146
```