Fruitful functions:

We write functions that return values, which we will call fruitful functions. Any function that returns a value is called Fruitful function. A Function that does not return a value is called a void function

The Keyword "return" is used to return back the value to the called function

```
# returns the area of a circle with the given radius:
def area(radius):
  temp = 3.14 * radius**2
  return temp
print(area(4))


def area(radius):
  return 3.14 * radius**2
print(area(2))

#Parameters:
#Parameters are passed during the definition of function while
Arguments are passed during the function call.



def add(a,b): #//function definition #here a and b are parameters
 return a+b

result=add(12,13)#function call #12 and 13 are arguments
print(result)
```

FUNCTION TYPES

Type1 : No parameters and no return type

Type 2: with param with out return type

Type 3: without param with return type

Type 4: with param with return type

```
#Type1 : No parameters and no return type
def Fun1() :
    print("No parameters and no return type")
Fun1()


#Type 2: with param with out return type
def fun2(a) :
    print("with param with out return type")
```

```
    print(a)
print()
fun2("hello")


#Type 3: without param with return type
def fun3():
    print("without param with return type")
    return "welcome to python"
print()
print(fun3())


#Type 4: with param with return type
def fun4(a):
    print("with param with return type")
    return a
print()
print(fun4("python is better then c"))
```

Local and Global scope:

Local Scope: A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing

Global Scope: A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file.

• The variable defined inside a function can also be made global by using the global statement. def function_name(args): ..... global x #declaring global variable inside a function .....

```
# create a global variable
x = "global"

def f():
    x=10
    print("x inside :", x)

f()
print("x outside:", x)

# create a local variable
def f1():
    y = "local"
    print(y)
f1()
```

```
# use local and global variables in same code
x = "global"

def f3():
    global x
    x=2
    y = "local"
    x=x*2
    print(x)
    print(y)

f3()
print(x)

x = 5

def f4():
    x= 10
    print("local x:", x)

f4()
print("global x:", x)

def myfunc():
  x = 300
  print(x)

myfunc()

#The local variable can be accessed from a function within the
function:

def myfunc():
  x = 300
  def myinnerfunc():
    print(x)
  myinnerfunc()

myfunc()
print(x)

#A variable created outside of a function is global and can be used by
anyone:

x = 300

def myfunc():
```

```
  print(x)

myfunc()

print(x)
```

```
#Naming Variables
#If you operate with the same variable name inside and outside of a
function, Python will treat them as two separate
#variables, one available in the global scope (outside the function)
and one available in the local scope (inside the function):

#Example
#The function will print the local x, and then the code will print the
global x:

x = 300

def myfunc():
  x = 200
  print(x)

myfunc()

print(x)
```

```
def myfunc():
  global xyz1
  xyz1 = 300

myfunc()

print(xyz1)
```

```
x = 300

def myfunc():
  global x
  x = 200

myfunc()

print(x)
```

Function composition

Function composition is the way of combining two or more functions in such a way that the output of one function becomes the input of the second function and so on.

```
# Function to add 2  to a number
def add(x):
```

```
    return x + 2

# Function to multiply  2 to a number
def multiply(x):
    return x * 2

# Printing the result of composition of add and multiply to add 2 to a
number and then multiply by 2
print("Adding 2 to 5 and multiplying the result with 2: ",
multiply(add(5)))
```

What is recursion

Recursion is the process of defining something in terms of itself.

```
def factorial(x):
    """This is a recursive function
    to find the factorial of an integer"""

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))


num = 3
print("The factorial of", num, "is", factorial(num))

factorial(3)          # 1st call with 3
3 * factorial(2)      # 2nd call with 2
3 * 2 * factorial(1)  # 3rd call with 1
3 * 2 * 1             # return from 3rd call as number=1
3 * 2                 # return from 2nd call
6                     # return from 1st call

Our recursion ends when the number reduces to 1. This is called the
base condition.

Every recursive function must have a base condition that stops the
recursion or else the function calls itself infinitely.
```

Advantages of Recursion

Recursive functions make the code look clean and elegant.

A complex task can be broken down into simpler sub-problems using recursion.

Sequence generation is easier with recursion than using some nested iteration.

Disadvantages of Recursion

Sometimes the logic behind recursion is hard to follow through.

Recursive calls are expensive (inefficient) as they take up a lot of memory and time.

Recursive functions are hard to debug.

Python lambda (Anonymous Functions)

In Python, anonymous function means that a function is without a name. As we already know that "def" keyword is used to define the normal functions and the "lambda" keyword is used to create anonymous functions.

This function can have any number of arguments but only one expression, which is evaluated and returned.

lambda functions are syntactically restricted to a single expression.

```
X=lambda a : a
print(X(5))
```

```
#A lambda function that multiplies argument a with argument b and
print the result:

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

x = lambda a, b, c : a + b + c
print(x(5, 6, 2))

# Python code to illustrate cube of a number
# showing difference between def() and lambda().

def cube(y):
    return y*y*y

g = lambda x: x*x*x
print(g(5))

print(cube(5))
```

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:

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

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

mydoubler = myfunc(2)

print(mydoubler(11))
print(mydoubler(25))
```

map function

Thebuilt-in map() function is especially useful where it is required to process each item from one or more iterable sequences (string, list, tuple or dictionary). It subjects each element in the iterable to another function which may be either a built-in function, a lambda function or a user-defined function and returns the mapped object.

Syntax map(function, iterables)

function: The function to execute for each item

iterable: A sequence, collection or an iterator object. You can send as many iterables as you like, just make sure the function has one parameter for each iterable.

The returned value from map() (map object) can then be passed to functions like list() (to create a list), set() (to create a set) and so on.

```
#In following example we first define a function to compute factorial
of a number.

def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

# The map() function applies it to each element in the numbers[] list.

#The map object is converted to generate squares[] list.

numbers=[1,2,3,4,5]
factmap=map(factorial,numbers)
print(factmap)
print(list(factmap))
```

```
# instead of user-defined function, we now use factorial() function
from math module and
# use it as argument to map() function.

from math import factorial
numbers=[1,2,3,4,5]
factmap=map(factorial,numbers)
print (list(factmap))

# Next example uses a lambda function as argument to map() function.
# The lambda function itself takes two arguments taken from two lists
and returns first number raised to second.
# The resulting mapped object is then parsed to output list.

powersmap=map(lambda x,y: x**y, [10,20,30], [4,3,2])
print (set(powersmap))

def calculateSquare(n):
    return n*n

numbers = (1, 2, 3, 4)
result = map(calculateSquare, numbers)
print(list(result))

# converting map object to set
numbersSquare = list(result)
print(numbersSquare)

def myfunc(a, b):
  return a + b

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon',
'pineapple'))

print(x)

#convert the map into a list, for readability:
print(list(x))


# Add two lists using map and lambda

def myfun(a,b):
    return a+b

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]

x = map(myfun, numbers1,numbers2)
print(list(x))
```

```
result = map(lambda x, y: x + y, numbers1, numbers2)
print(list(result))
```

```
# Python program to demonstrate working of map.

# Return double of n
def addition(n):
    return n + n

# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

```
%reset
```

```
# Double all numbers using map and lambda

numbers = (1, 2, 3, 4)
result = map(lambda x: x + x, numbers)
print(list(result))

# List of strings
l = ['sat', 'bat', 'cat', 'mat']

# map() can listify the list of strings individually
test = list(map(list, l))
print(test)
```

filter() in python

The filter() method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.

syntax:filter(function, sequence)

function: function that tests if each element of a sequence true or not.

sequence: sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.

Returns: returns an iterator that is already filtered.

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
```

```
  else:
    return True

adults = filter(myFunc, ages)
print(type(adults))
print(type(ages))
print(adults)
for x in adults:
  print(x)

# It is normally used with Lambda functions to separate list, tuple,
or sets.

# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2 != 0, seq)
print(result)
print(list(result))

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
```

reduce()

Python's reduce() implements a mathematical technique commonly known as folding or reduction. You're doing a fold or reduction when you reduce a list of items to a single cumulative value. Python's reduce() operates on any iterable—not just lists—and performs the following steps:

Apply a function (or callable) to the first two items in an iterable and generate a partial result.

Use that partial result, together with the third item in the iterable, to generate another partial result.

Repeat the process until the iterable is exhausted and then return a single cumulative value.

In Python 3.x, if you need to use reduce(), then you first have to import the function into your current scope using an import statement in one of the following ways:

import functools and then use fully-qualified names like functools.reduce().

from functools import reduce and then call reduce() directly.

```
product = 1
list1 = [1, 2, 3, 4]
for num in list1:
```

```
    product = product * num
print(product)

from functools import reduce
product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
print(product)
```

```
24
```

```
# python code to demonstrate working of reduce()

# importing functools for reduce()
import functools

# initializing list
lis = [ 1 , 3, 5, 6, 2,4]

# using reduce to compute sum of list
print ("The sum of the list elements is : ",end="")
print (functools.reduce(lambda a,b : a+b,lis))

# using reduce to compute maximum element from list
print ("The maximum element of the list is : ",end="")
print (functools.reduce(lambda a,b : a if a > b else b,lis))

# python code to demonstrate working of reduce() using operator
functions

# importing functools for reduce()
import functools

# importing operator for operator functions
import operator

# initializing list
lis = [ 1 , 3, 5, 6, 2, 4]

# using reduce to compute sum of list using operator functions
print ("The sum of the list elements is : ",end="")
print (functools.reduce(operator.add,lis))

# using reduce to compute product
# using operator functions
print ("The product of list elements is : ",end="")
print (functools.reduce(operator.mul,lis))



# using reduce to concatenate string
print ("The concatenated product is : ",end="")
```

```
print (functools.reduce(operator.add,["mrcet ","engineering
","college"]))
```

```
from functools import reduce
lista=[]
for var in range(25):
    var=var+1
    lista.append(var)
print(lista)

evenlist=list(filter(lambda x:x%2==0,lista))
print(evenlist)

doubleevenlist=list(map(lambda x: x*2,evenlist))
print(doubleevenlist)

sumofdoubleevenlist=reduce(lambda x,y:x+y,doubleevenlist)
print(sumofdoubleevenlist)
```

Comprehensions in Python

Comprehensions in Python provide us with a short and concise way to construct new sequences (such as lists, set, dictionary etc.) using sequences which have been already defined.

Python supports the following 4 types of comprehensions:

List Comprehensions

Dictionary Comprehensions

Set Comprehensions

Generator Comprehensions

List Comprehensions:

List Comprehensions provide an elegant way to create new lists. The following is the basic structure of a list comprehension:

output_list =[output_exp for var in input_list if (var satisfies this condition)]

Note that list comprehension may or may not contain an if condition. List comprehensions can contain multiple for (nested list comprehensions).

```
# Constructing output list WITHOUT Using List comprehensions
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

output_list = []

# Using loop for constructing output list
for var in input_list:
```

```
    if var % 2 == 0:
        output_list.append(var)

print("Output List using for loop:", output_list)

# Using List comprehensions  for constructing output list

input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

list_using_comp = [var for var in input_list if var % 2 == 0]

print("Output List using list comprehensions:", list_using_comp)

# Constructing output list using for loop
output_list = []
for var in range(1, 10):
    output_list.append(var ** 2)

print("Output List using for loop:", output_list)


# Constructing output list using list comprehension
list_using_comp = [var**2 for var in range(1, 10)]

print("Output List using list comprehension:",list_using_comp)
```

Dictionary Comprehensions:

Extending the idea of list comprehensions, we can also create a dictionary using dictionary comprehensions. The basic structure of a dictionary comprehension looks like below.

output_dict = {key:value for (key, value) in iterable if (key, value satisfy this condition)}

```
input_list = [1, 2, 3, 4, 5, 6, 7]

output_dict = {}

# Using loop for constructing output dictionary
for var in input_list:
    if var % 2 != 0:
        output_dict[var] = var**3


print("Output Dictionary using for loop:", output_dict )



# Using Dictionary comprehensions for constructing output dictionary

input_list = [1,2,3,4,5,6,7]
```

```
dict_using_comp = {var:var ** 3 for var in input_list if var % 2 != 0}


print("Output Dictionary using dictionary comprehensions:",
dict_using_comp)
```

Set Comprehensions:

Set comprehensions are pretty similar to list comprehensions. The only difference between them is that set comprehensions use curly brackets { }. Let's look at the following example to understand set comprehensions.

```
input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

output_set = set()

# Using loop for constructing output set
for var in input_list:
    if var % 2 == 0:
        output_set.add(var)

print("Output Set using for loop:", output_set)



# Using Set comprehensions  for constructing output set

input_list = [1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 7]

set_using_comp = {var for var in input_list if var % 2 == 0}

print("Output Set using set comprehensions:", set_using_comp)
```

Generator Comprehensions:

Generator Comprehensions are very similar to list comprehensions. One difference between them is that generator comprehensions use circular brackets whereas list comprehensions use square brackets. The major difference between them is that generators don't allocate memory for the whole list. Instead, they generate each value one by one which is why they are memory efficient. Let's look at the following example to understand generator comprehension:

```
input_list = [1, 2, 3, 4, 4, 5, 6, 7, 7]

output_gen = (var+var for var in input_list)

print("Output values using generator comprehensions:", end = ' ')

for var in output_gen: print(var, end = ' ')
```

Python Arrays

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

Array can be handled in Python by a module named "array". They can be useful when we have to manipulate only a specific data type values. A user can treat lists as arrays. However, user cannot constraint the type of elements stored in a list. If you create arrays using the array module, all elements of the array must be of the same type.

Basic Operations

Traverse – print all the array elements one by one.

Insertion – Adds an element at the given index.

Deletion – Deletes an element at the given index.

Search – Searches an element using the given index or by the value.

Update – Updates an element at the given index.

Methods

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

```
# Python program to demonstrate  Creation of Array


import array as arr # importing "array" for array creations

a = arr.array('i', [1, 2, 3]) # creating an array with integer type
print(a)
```

```
print ("The new created array is : ", end =" ") # printing original
array
for i in range (0, 3):
    print (a[i], end =" ")
print()

b = arr.array('d', [2.5, 3.2, 3.3]) # creating an array with float
type

print ("The new created array is : ", end =" ")
for i in range (0, 3):
    print (b[i], end =" ")
```

Type Code	C Type	Python Type	Minimum Size In Bytes
'b'	signed cahar	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	int	2
'l'	signed long	int	4
'L'	unsigned long	int	4
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Adding Elements to a Array

Elements can be added to the Array by using built-in insert() function. Insert is used to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array. append() is also used to add the value mentioned in its arguments at the end of the array.

```
# Python program to demonstrate Adding Elements to a Array

import array as arr
from array import *

a = arr.array('i', [1, 2, 3])

print ("Array before insertion : ", end =" ")
for i in range (0, 3):
    print (a[i], end =" ")
```

```
print()

for i in (a):
    print(i)

a.insert(1, 4) # inserting array using  insert() function

print ("Array after insertion : ", end =" ")
for i in (a):
    print (i, end =" ")
print()

my_extnd_array = array('i', [7,8,9,10])
a.extend(my_extnd_array)
for i in a:
    print(i)

a.append(10)
print(a)

# Python program to demonstrate  accessing of element from list

import array as arr

a = arr.array('i', [1, 2, 3, 4, 5, 6])

# accessing element of array
print("Accessed element is: ", a[2])

print("Access element is: ", a[3])

b = arr.array('d', [2.5, 3.2, 3.3])

print("Access element is: ", b[1])

print("Access element is: ", b[2])
```

Removing Elements from the Array

Elements can be removed from the array by using built-in remove() function but an Error arises if element doesn't exist in the set. Remove() method only removes one element at a time, to remove range of elements, iterator is used. pop() function can also be used to remove and return an element from the array, but by default it removes only the last element of the array, to remove element from a specific position of the array, index of the element is passed as an argument to the pop() method.

Note – Remove method in List will only remove the first occurrence of the searched element.

```
# Python program to demonstrate Removal of elements in a Array

# importing "array" for array operations
import array

arr = array.array('i', [1, 2, 3, 1,10, 5])

print ("The new created array is : ", end ="")
for i in range(len(arr)):
    print(arr[i], end =" ")
print ()

# using pop() to remove element at 2nd position
print ("The popped element is : ", arr.pop())
print ("The popped element is : ", arr.pop(2))


# printing array after popping
print ("The array after popping is : ", end ="")
for i in range (0, 4):
    print (arr[i], end =" ")

print()

# using remove() to remove 1st occurrence of 1
arr.remove(10)

# printing array after removing
print ("The array after removing is : ", end ="")
for i in range (0, 3):
    print (arr[i], end =" ")
```

Slicing of a Array

In Python array, there are multiple ways to print the whole array with all the elements, but to print a specific range of elements from the array, we use Slice operation. Slice operation is performed on array with the use of colon(:).

To print elements from beginning to a range use [:Index],

to print elements from end use [:-Index],

to print elements from specific Index till the end use [Index:],

to print elements within a range, use [Start Index:End Index] and

to print whole List with the use of slicing operation, use [:].

Further, to print whole array in reverse order, use [::-1].

```
# Python program to demonstrate silicing of elements in a Array
```

```
import array as arr

# creating a list
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(type(l))

a = arr.array('i', l)
print(type(a))

print("Intial Array: ")
for i in (a):
    print(i, end =" ")

# Print elements of a range using Slice operation
print("\nSlicing elements in a range 3-8: ", a[3:8])

# Print elements from a  pre-defined point to end
print("\nElements sliced from 5th element till the end: ",a[5:] )

# Printing elements from beginning till end
print("\nPrinting all elements using slice operation: ",a[:])
```

Searching element in a Array

In order to search an element in the array we use a python in-built index() method. This function returns the index of the first occurrence of value mentioned in arguments.

```
# searching an element in array

import array

arr = array.array('i', [1, 2, 3, 1, 2, 5])

# printing original array
print ("The new created array is : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print ()


# using index() to print index of 1st occurrenece of 2
print ("The index of 1st occurrence of 2 is : ", arr.index(2))

# using index() to print index of 1st occurrenece of 1
print ("The index of 1st occurrence of 1 is : ", arr.index(1))
print ("The index of 1st occurrence of 1 is : ", arr.index(7))
```

```
from array import *
arr1=array('i',[]) #empty array
n=int(input("how many elements:"))
```

```
for i in range(n):
    x=int(input("enter value"))
    arr1.append(x)
print(arr1)

val=int(input("enter value to be searched:"))
counter=0
for i in arr1:
    if(i==val):
        break
    counter=counter+1

print("value found at index",counter)

print(arr1.index(val))
```

Updating Elements in a Array

In order to update an element in the array we simply reassign a new value to the desired index we want to update.

```
# importing array module
import array

arr = array.array('i', [1, 2, 3, 1, 2, 5])

print ("Array before updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")

print ()

# updating a element in a array
arr[2] = 6
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")
print()

# updating a element in a array
arr[4] = 8
print("Array after updation : ", end ="")
for i in range (0, 6):
    print (arr[i], end =" ")

import sys
!conda install --yes --prefix {sys.prefix} numpy

import numpy as np

import numpy as np
```

```
print (np.__version__)
```

```
from array import *
arr = array('i',[1,2,3,4,5])
arr1 = array(arr.typecode,(i+5 for i in arr))
print(arr1)
```

```
for i in range(len(arr)):
    arr1.append(arr[i]+5)
print(arr1)
```

```
%reset
```

2

A) Write a python program to add two numbers.

B) Write a python program to print a number is positive/negative using if-else.

C) Write a python program to find largest number among three numbers.

D) Write a python Program to read a number and display corresponding day using if_elif_else?

3

A) Write a program to create a menu with the following options

1. TO PERFORM ADDITITON 2. TO PERFORM SUBTRACTION 3. TO PERFORM MULTIPICATION 4. TO PERFORM DIVISION Accepts users input and perform the operation accordingly. Use functions with arguments.

B) Write a python program to check whether the given string is palindrome or not.

C) Write a python program to find factorial of a given number using functions

D) Write a Python function that takes two lists and returns True if they are equal otherwise false

```
NumList = []

Number = int(input("Please enter the Total Number of List Elements:
"))
for i in range(1, Number + 1):
    value = int(input("Please enter the Value of %d Element : " %i))
    NumList.append(value)

for i in range (Number):
    for j in range(i + 1, Number):
```

```
        if(NumList[i] > NumList[j]):
            temp = NumList[i]
            NumList[i] = NumList[j]
            NumList[j] = temp

print("Element After Sorting List in Ascending Order is : ", NumList)
```