23/2/24

Unit-II: functions and Modules:-

Functions:

Functions contains a block of code to perform a particular task.

In Python, functions are defined using 'def' keyword.

→ Every function contains definition & declaration and calling.

syntax: def function_name(arg1, arg2.....):
    #statement
    #statement
    :
    return seq/value

Eg:
```
def add(a,b):
    print(a+b)
add(3,4)
```
o/p: 7.

Types of functions:

1. Pre-defined functions
2. User-defined functions.

1. Predefined functions:

functions already defined with compiler are called Pre-defined / built-in functions.

Eg: print(), input(), bin(), str(), Hex()....

2. User-defined functions:

The functions which are declared by the user are called as user-defined functions.

return statement: syntax: return exp/value/sequence

ex: return a+b; return a
    return 10 return 1,2,3,4,'IT'

→ 'Tuple' is the default (sequence) return type.

return 30,40,'IT',33.4 # default return values in the form of tuple.
(0 1 2 3)

* Types of user-defined function:

1. functions without Parameters and without return type.
2. functions without parameters and with return type.
3. functions with parameters and without return type.
4. functions with parameters and with return type.

→ The (need) outcome of the function depends on parameters.

Types of Parameters:

1. Actual / original Parameters:

The input data directly stored in the actual parameters. These Parameters are specified in function calling.

Parameter stores the input data

2. formal / dummy Parameters:

The parameters which receives copy from actual parameters. These Parameters are specified in function definition.

Definition parameters are called formal.

→ There is function overloadding in python.

1. functions without parameters and without return type.

```
def add():  #definition/declaration
    print ("Enter 3 values")
    a,b,c = int(input()), int(input()), int(input())
    print ("Addition is: ", a+b+c)
add()  #calling.
```

2. functions without parameters and with return type.

```
def add():  #definition.
    print ("enter 3 values")
    a,b,c = int(input()), int(input()), int(input())
    return a+b+c
print ("Addition is ", add())  #calling
```

3. functions with parameters and without return type.

```
def add(a,b,c):
    # print("enter 3 values")
    # a,b,c = int(input()), int(input()), int(input())
    print("Addition is: ", a+b+c)
print("enter 3 values")
x,y,z = int(input()), int(input()), int(input())
add(x,y,z)  # x,y,z are actual parameters.
```

4. functions with parameters and with return type.

```
def add(a,b,c):
    return a+b+c
print("enter 3 values")
x,y,z = int(input()), int(input()), int(input())
print("Addition is:", add(x,y,z))
```

LAB Program: 6

```
def fib(k):
    a,b = 0,1 ; s = 0
    while (k>0):
        if a%2 == 0:
            s = s+a
        a,b = b, a+b
        k = k-1
    return s
n = int(input("enter a value"))
print("Even values sum is:", fib(n))
```

* Types of Arguments:

1. Positional Arguments
2. Default arguments
3. keyword arguments
4. arbitary positional arguments
5. arbitary keyword arguments.

1. Positional Arguments:-

```
syntax: def function_name(arg1, arg2, arg3 ....):
            #statements
        function_name(arg1, arg2, arg3 ....)
```

In positional arguments, both formal and actual parameters count - same.

```
Ex: def add(a,b,c):
        return a+b+c
    print("enter 3 values:")
    x,y,z = int(input()), int(input()), int(input())
    print("addition is:", add(x,y,z))
```

2. Default arguments:

Syntax:

```
def function_name(arg1=val1, arg2=val2, arg3=val3.....)
    #statements
function_name(arg1, arg2, arg3...)
function_name(arg1, arg2)
function_name(arg1)
funct_name()
```

→ Default arguments always "initialised" from "right to left".

Ex:

```
def add(a=3, b=4, c=5):
    # return a+b+c
    print(a+b+c)
```

add() — o/p: 12.

add(1) — o/p: 10.

add(1,2) #8

add(1,2,3) #6.

) def add(a=3, b=4, c): #error.

) def add(a, b=4, c=5):
    print(a+b+c)

add() #error

add(1) #10

add(1,2) #8

add(1,2,3) #6.

3. Keyword arguments:

Syntax:

```
def function_name(arg1, arg2, arg3...):
    #statements
function_name(arg3=value1, arg1=val2, arg2=val3...)
```

arg3, arg1, arg2 are called keyword arguments.

Ex:

```
def add(a,b,c):
    print(a+b+c)
    print(a,b,c)
x,y,z = 5,6,7
add(c=x, a=z, b=y)
```

o/p: 18
7 6 5

4. Arbitary positional arguments:

→ These are represented with astrisk symbol('*')

→ these are also called "variable length arguments"

Syntax:

```
def function_name(*args):
    #statements
function_name(arg1,arg2,arg3...) # in the form of tuple.
```

Ex:

```
def add(*k):
    print(k)    # print(sum(k)) # 18.
x,y,z = 5,6,7
add(x,y,z)          add(x,y,z,1,2,3) # (5,6,7,1,2,3)
```

o/p: (5, 6, 7)

Recursion:- function calls itself is called recursion.

Ex: write a pp to display factorial of a number using default arguments.

```
def fact(k=5):
    if k==0:
        return 1
    else:
        return k*fact(k-1)
```

```
n=int(input("enter an int"))
print("factorial is:",fact(n))
print("factorial is:",fact())
```

o/p: enter an int 6

factorial is: 720

factorial is: 120.

LAB Program: (7)

```
s=0
def rev(n):
    global s
    if n>0:
        s=s*10+n%10
        n//=10
        rev(n)
    return s
n=int(input("enter an integer: "))
print("Reverse is:",rev(n))
#print("Reverse is:",rev())
```

Arbitary arguments in calling:

```
def add(a,b,c):
    print(a+b+c)
K=(3,4,5)
add(*K) #arbitary, tuple form.
```

Arbitary keyword arguments:

** symbol is used to represent arbitary keyword arguments, data is passed in the form of dictionary.

Syntax:

def function_name(**variable_name):

definition:

```
def display(**k):
        print(k)
display(a=3, b=6, c=9, d='it', e=2.3)
```

o/p: {'a':3, 'b':6, 'c':9, 'd':it, 'e':2.3}

Calling:

```
def display(a,b,c):
        print(a,b,c)
d={'a':3, 'b':6, 'c':9}
display(**d)
```

o/p: 3 6 9

```
def add(a:int, b:int)->int:        from 5.3 version.
        print(a+b)
add(3,4)
add(3.2,2.3)
add('a','b')
```

o/p: 7
5.5
ab

→ write a pp to perform arthmetic operations using functions.

```
def arithmetic(a,b):
        return a+b, a-b, a*b, a/b, a//b
x=int(input("enter first number"))
y=int(input("enter second number"))
print(arithmetic(x,y))
```

o/p: enter first number 3
enter second number 4

(7, -1, 12, 0, 75, 0)

```
def arithmetic(a,b):
        return a+b, a-b, a*b, a/b, a%b
x=int(input("enter first number"))
y=int(input("enter second number"))
z=arithmetic(x,y)
print("Addition of %d and %d is %d "%(x,y,z[0]))
print("subtraction of %d and %d is %d "%(x,y,z[1]))
print("Multiplication of %d and %d is %d "%(x,y,z[2]))
print("float division of %d and %d is %d"%(x,y,z[3]))
print("floor division of %d and %d is %d "%(x,y,z[4]))
```

24

Recursion in Python: ex (factorial, gcd, fibonacci)

The function calls itself is called recursion.

write a pp to display fibonacci series.

```
def fib(k):
    if k==1:
        return 0
    elif k==2:
        return 1
    else:
        return fib(k-2)+fib(k-1)
n=int(input("enter an integer"))
for i in range(1,n+1):
    print(fib(i))
```

→ The advantage of recursion is code reduction.

→ The disadvantage of recursion is more complexicity and more dangerous when compared to loops.

reverse
natural numbers
even series.

How to find gcd of given two numbers without using predefined function and using recursion.

```
def gcd(a,b):
        if b==0:
            return a
        else:
            return gcd(b,a%b)
a=int(input("enter first number"))
b=int(input("enter second number"))
print("GCD is:", GCD(a,b))
```

GCD

fun = GCD
fun(5,6)

Python-decorators

C-function pointers.

→ Anonymous or lambda functions:-

By using 'lambda' keyword python anonymous functions are defined.

These functions are mostly return in single line.

Syntax:

var/functionname = lambda arg1, arg2 : exp/statements.

Ex: p=lambda a,b:a+b

p(3,4)

7

write a pp to find out addition of given three numbers using lambda expressions.

```
add = lambda a,b,c:a+b+c
a=int(input("enter first number"))
b=int(input("enter second number"))
c=int(input("enter third number"))
print("Addition is:", add(a,b,c))
```

Colon:
slicing
functions
blocks
lambda function.

Parameter Passing techniques in python:

serialization -
deserialization -

parameter may be text type or video type & audio types.

Applications
client server communication

Types of Parameters:

1. Actual parameters
2. formal parameters

1. call by value:

Modification of formal parameters doesnot effect on actual parameters.

In python all immutable data types are followed call by value mechanism. (int, float, str, tuple ...etc)

2. call by reference:

Modification of formal parameters must effect on actual parameters.

In python all mutable data types are followed call by reference mechanism. (list, set and dictionary)

→ Every variable is called object in python.

Ex: call by value:

```
def display(a):   # a is formal parameter
    a=a+10
    print(a)  # a=30
x=20
print(x)  #x=20
display(x)   # x is actual parameter.
print(x)  #x=20
```

o/p: 20
30
20

2) swapping of given two numbers using call by value mechanism

```
def swap(a,b): # a,b are formal parameters
        a,b=b,a
        print("swapping in definition of function:",a,b)
x=int(input("enter first number"))
y=int(input("enter second number"))
print("Before function call:",x,y)
swap(x,y)  # x,y are actual parameters.
print("After function call:",x,y)
```

o/p: enter first number 4
enter second number 5
Before function call 4 5
swapping in definition of function 5 4
After function call 4 5

ex-3: student information.

```
def studentinfo(no,name,address,mail):
        no=1200                    formal parameters.
        name="John"
        address="us";
        mail='john123@gmail.com'
        print("In definition:")
        print("student no is:",no)
        print("student name is:",name)
        print("student address is:",address)
        print('student mail is:',mail)
p=int(Input("enter roll number"))
q=input("enter name:")
r=input("enter address:")
s=input("enter mail id:")
print("Before calling:")
```

In call by value, memory is seperately allocated for actual and formal parameters. so, they are not updated.

```
print("student no is:", p)
print("student name is:", q)
print("student address is:", r)
print("student mail is:", s)
studentinfo(p,q,r,s) #p,q,r,s actual parameters
print("After calling:")
print("student no is:", p)
print("student name is:", q)
print("student address is:", r)
print("student mail is:", s)
```

Call by reference:

ex-1: index, compound assignment operator

```
def display(a): # a is formal parameter
        a[0] = a[0] + 10  ---> a=[30]
        print(a) # a=30
x=[20]
print(x) # x=20
display(x) # x is actual parameter.
print(x) # x=30
```

o/p: [20]
[30]
[30]

o/p: [20]
[30]
[20]

```
def display(a):
        a+=[30]
        print(a)
x=[20]
print(x)
display(x)
print(x)
```

o/p:

2) swapping of two numbers

```
def swap(a,b):  #a,b is called formal parameter.
    a[0], a[1] = a[1], a[0]
    print("swapping in def of function:", a)
x=[int(input("enter first number:")), int(input("enter second number:"))]
print("Before function call:", x)
swap(x) # x is called actual parameter.
print("After function call:", x)
```

3) Student information.

```
def studentinfo(n):
    n[0]=1200
    n[1]="John"
    n[2]="US"
    n[3]="John123@gmail.com"
    print("In definition:")
    print("student no is:", n[0])
    print("student name is:", n[1])
    print("student address is:", n[2])
    print("student mail is:", n[3])
```

11/3/24

```
s=[int(input("enter roll number")), input("enter name:"),
   input("enter address"), input("enter mail")].
print("Before calling:")
print("student no is:", s[0])
print("student name is:", s[1])
print("student address is:", s[2])
print("student mail is:", s[3])
studentinfo(s)
Print("student After calling:")
print("student no is:", s[0])
print("student name is:", s[1])
print("student address is:", s[2])
print("student mail is:" s[3])
```

fruitful function in Python:

A function which contains return type statement is called as fruitful function.

Syntax:

```
def fun-name(arg1,arg2 ...):
        #statement1
        #statement 2
        .....
        return var|exp|seq.
```

Need:
Data is processed and return back to the caller.

Justification

Ex:

```
def add(a,b,c):
        return a+b+c
a=int(input("enter first number"))
b=int(input("enter second number"))
c=int(input("enter third number"))
print("Addition is: ", add(a,b,c))
```

(x,y,z)

void functions in python:-

A function without return statement is called as void functions. (It doesn't mean that void keyword is required).

Nested functions:- Data abstraction some times encapsulation Readability To develop decorators, closures, factory functions.

↓

Adding extra functionality for function.
features

function pointers stores the reference of normal function

```
void (*add)();
void sub()
add=sub
```

Advantage: To hidden the original function name and substitute it to another variable.

Decorators:

1) def add()
2) return function name
3) print(10+3)
4) @

```
sub=add
c=add
c()   #13.
```

Definition:

A function contains definition of another function is called as nested function.

→ In python Data hiding and encapsulation possible with nested functions.

→ closures and factory functions are also decided by using nested functions.

Syntax:
```
def fun1(arg1, arg2 ...):
    #statements
    #statements
    def fun2(arg1, arg2 ...):
        #statements
        #statements
        .....
    fun2(arg1, arg2 ...) # calling.
```

→ Inner functions are always called inside the outer function at the end of definition.

→ It is not possible to call outside the outer function.

Ex:
```
def add(x,y):
    print(x+y)
    def sub(a,b):
        print(a-b)
    sub(30,40)
add(10,20)
```

O/P: 30
-10

Create and access a user defined package where the package contains a module named arithmetic demo which inturn contains a methods called sumtwo(), subtwo(), multwo(), divtwo() which takes two numbers as parameters.

Step1: create a folder with name Arithmetic

Step2: Type the following program in IDLE

```
def sumtwo(x,y):
        return x+y
def  subtwo(x,y):
         return x-y
def  multwo(x,y):
          return x*y
def  divtwo(x,y):
           return x/y
```

Step3: save the above program as ArithmeticDemo.Py

Step4: Now open another notepad IDLE and type the following code.

```
from Arithmetic.ArithmeticDemo import *
a,b = int(input('enter first number')), int(input('enter second no'))
print("Addition of %d and %d is %d"%(a,b,sumtwo(a,b)))
print("subtraction of %d and %d is %d"%(a,b,subtwo(a,b))
print("Multiplication of %d and %d is %d"%(a,b, multwo(a,b))
print("Division of %d and %d is %"%(a,b divtwo(a,b))
```

write a python program to compute LCD, GCM of Given two numbers.

```
from  math  import *
a=int(input('enter  first  number'))
b=int(input('enter  second  number'))
print("GCD is.", gcd(a,b))
print("LCM is", (a*b)/gcd(a,b))
```

Example: Lab Program - (10):

```
from itertools import *
print("count() function:")
for i in count(10,40):
        if i>200:
            break
        print(i)
print("cycle() function:")
k=0
for i in cycle("GEC"):
        if k==0:
            break
        print(i, end='')
        k=k+1
print("repeat() function:")
print(list(repeat('GEC', 10)))
#purpose of permutations.
print("permutations() function:")
for i in permutations("GEC"):
            print(''.join(i))
print("combinations() function:")
for i in combinations('srgec', 2):
            print(''.join(i))
print("product() function:")
for i in product("GEC", "IT"):
            print(''.join(i), end='')
print("chain() function:")
print(list(chain([3,4], [4,5,6], ['IT','CSE','EEE'])))
```

LAB program-11

```
st=input("enter a string")
s=''
for i in st:
    if i not in 'AEIOUaeiou':
        s=s+i
print("After removing vowels the string is",s).
```

LAB program-12:

```
def strlen(s):
    if s=='':
        return 0
    else:
        return 1+strlen(s[1:])
s=0
def palindrome(n):
    global s
    if n>0:
        s=s*10+n%10
        n=n//10
        palindrome(n)
    return s
st=input("enter string")
k=int(input('enter an integer'))
print("string length is:", strlen(st))
if(k==palindrome(k)):
    print(k,"is palindrome")
else:
    print(k," is not palindrome")
```

20/4/24

Types of variables in Python:

1) local
2) global
3) non-local.

1) Local variables: Variables which are declared within a function are called as local variables.

The memory is represented in the form of local namespaces with dictionary format.

Syntax: def function_name:
    # variables

Ex-1:
```
def display(a): # a is local
    b=20  # b is local
    c=30  # c is local
    print(a+b+c)
display(50)
```

2) Global variables: Variables declared outside the function are called as Global variables.

Ex:
```
a=10  # global variable
def display(c): # local
    d=10  # local
    print(d+c)
display(30)
```

→ To access or update global variables within function, python contains 'global' keyword.

Ex:
```
a=50 print(a) #50
def display():
    global a  # it access global variable a.
    a=100 print(a) #100
display()
print(a) #100.
```

2)
```
a= 80
print(a) #80
def display():
    a=10
    print(a) #10
display()
print(a) #80.
```

3) Non-local variables: enclosing scopes.

→ 'nonlocal' is a keyword in python.

→ variables which are declared the scope of outer function and inner function are called Non-local variables. (enclosing area/scope)

→ 'nonlocal' is used to access enclosing variables inside nested function.

Ex:
1)
```
a=10 #global
def f1():
    a=20 #non-local  enclosing area
    print(a) #20
    def f2():
        a=40 #local
        print(a) #40
    f2()
    print(a) #20
f1()
```

2)
```
a=10 #global
print(a) #10
def f1():
    a=20 #non-local.
    print(a) #20
    def f2():
        non-local a
        a=40
        print(a) #40
    f2()
    print(a) #40
f1()
```

21/3/24

Modules:

dir( ) function:

It is a directory function to display module information in the form of a list, including functions and constants.

ex: >import math
> dir(math)

['__doc__', '__loader__', '__name__', '__package__', '__spec__', ... 'true']

2) >>import itertools
>dir(itertools)

['__doc__', '__loader__', '__name__', ... 'zip_longest']

3) >>import os
>>dir(os)

['__doc__', 'DirEntry' .....]

* how to invoke module into program:

1) import modulename

ex: import math
import os
import itertools
import tkinter
import primefy
import numpy
import collections

ex: math.gcd(10,20)

2) import module as aliasname

ex: import math as m
import itertools as y

ex: m.gcd(10,20)
ex: y.permutations('gec')

3) from keyword: By using, from keyword we import modules in python

syntax:

from modulename import *

Ex: from math import *

gcd(10,20)

4) from keyword:

syntax: from modulename import fun1, fun2.

from math import gcd, pow

User defined module:

Ex:

emp module

ename, eid, eaddress, email

salary

ta, da, hra, ai

empinfo.

module is a py file it contains set of python programming instructions.

emp module:

```
def getDetails(eid,ename,email,address):
def getDetails(address="us"):
    eid=int(input("enter employee id!"))
    ename=input("enter name of the emp")
    email = input("enter mail id!")
    return eid,ename,email,address.
```

Salary module:

```
def sal(t,d,h,a):
    return t/100*a+d/100*a+h/100*a+a
           (t/100*m+d/100*m+h/100*m+m)*12
```

employeeinfo module:

```
import emp
import salary
x=emp.getDetails()
ta=int(input("enter ta %:"))
```

```
da = float(input("enter da %:"))
hra = float(input("enter hra %:"))
month = float("input("enter salary:"))
s = salary.sal(ta, da, hra, month)
print("emp id is", k[0])
print("emp name is", k[1])
print("emp address mail is", k[2])
print("emp address is", k[3])
print("salary is:", s).
```

output: