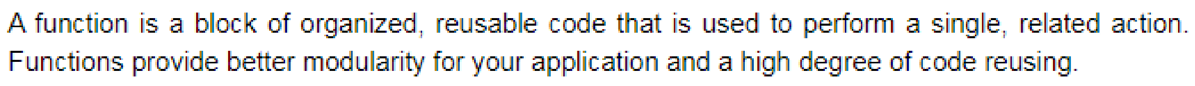
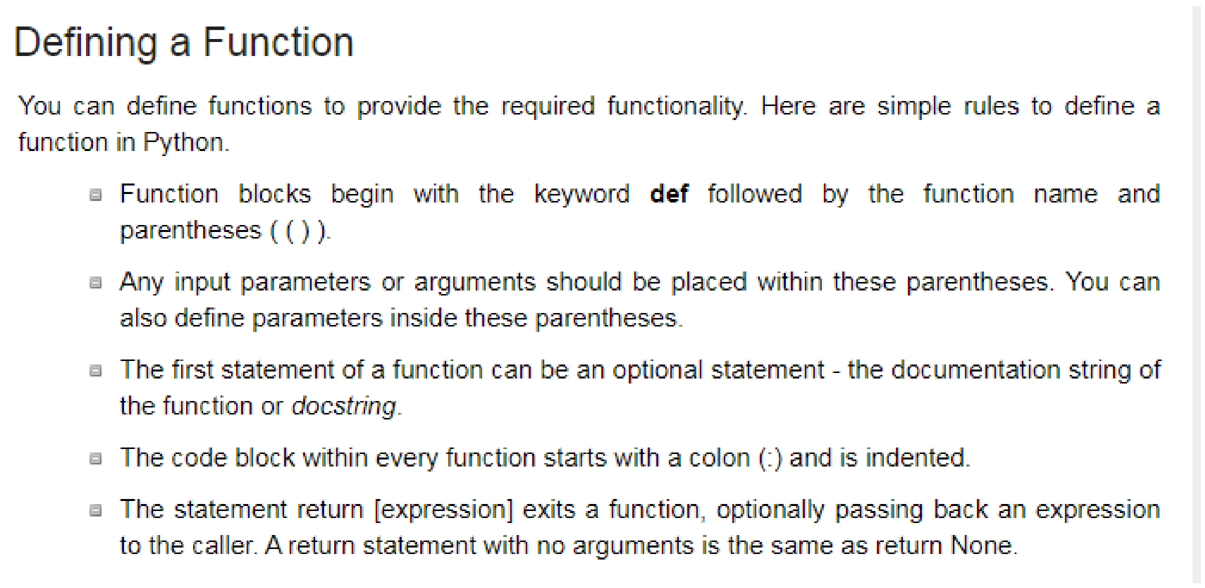
UNIT 3

Function

Q1. What do you mean by function? Give the syntax to define the function with example.

Ans:


A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.


Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Program for sum of digits from range x to y

```
def sum(x,y):   –> Master code
        s=0
        for  i in range(x,y+1):
                s-s+i
        print(“sum of integers from x to y :”,s)
sum(50,75)   –> Driver code
```

Output:

```
sum of integers from x to y : 1625
```

Function calling is called driver code and function definition with block of statements is called master code

Syntax

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

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

```
def printme( str ):
   "This prints a passed string into this function"
   print str
   return
```

Q2. Differentiate between argument and parameter.

Ans: Parameters are of two types:

1. Formal parameter
2. Actual Parameter

   Formal parameters are those written in function definition and actual parameters are those written in function calling.

   Formal parameters are called argument.

   Arguments are the values that are passed to function definition.

   Example:

   ```
   Def fun(n1):
     Print(n1)
   a=10
   fun(a)
   ```

   here n1 is argument and a is parameter.

Q3. Explain different types of arguments in python.

Ans:

You can call a function by using the following types of formal arguments −

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

```
def printme(str):

   print (str)

printme()# error missing 1 required positional argument

def printme( ):

   print(str)

printme('hello') #error, printme() takes 0 positional arguments
but 1 was given




def printme(str):

   print(str)

printme('abc')

Output:
abc
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

```
def printinfo( name, age ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

```
def printinfo( name, age = 35 ):
   "This prints a passed info into this function"
   print "Name: ", name
   print "Age ", age
   return;

# Now you can call printinfo function
printinfo( age=50, name="miki" )
printinfo( name="miki" )
```

When the above code is executed, it produces the following result −

```
Name:  miki
Age  50
Name:  miki
Age  35
```

```
def display(a='hello',b='world'): # you can have only default arguments also

   print(a,b)

display()
```

Output:

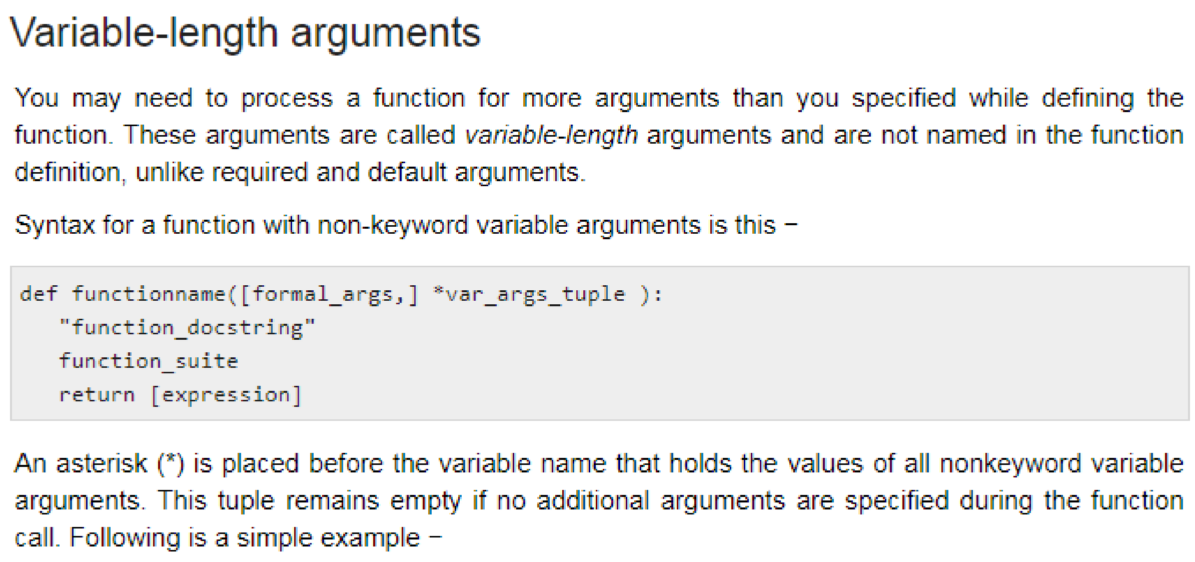
```
hello world
```

```
def display(a,c=10,b): # default argument will always come as last argument
```

```
   d=a+b+c
   print(d)
display(2,3)   # error, non default argument follows default argument
def display(a,b,c=10):
   d=a+b+c
   print(d)
display(2,3)
```

Output:

```
15
```



Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

```
def functionname([formal_args,] *var_args_tuple ):
   "function_docstring"
   function_suite
   return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

Variable length with first extra argument:

```
def printinfo( arg1, *vartuple ):
   "This prints a variable passed arguments"
   print "Output is: "
   print arg1
   for var in vartuple:
      print var
   return;

# Now you can call printinfo function
printinfo( 10 )
printinfo( 70, 60, 50 )
```

When the above code is executed, it produces the following result −

```
Output is:
10
Output is:
70
60
50
```

Variable length argument Example:

```
def display(*arg):
    for i in arg:
        print(i)
display('hello','world','python')
```

Output:

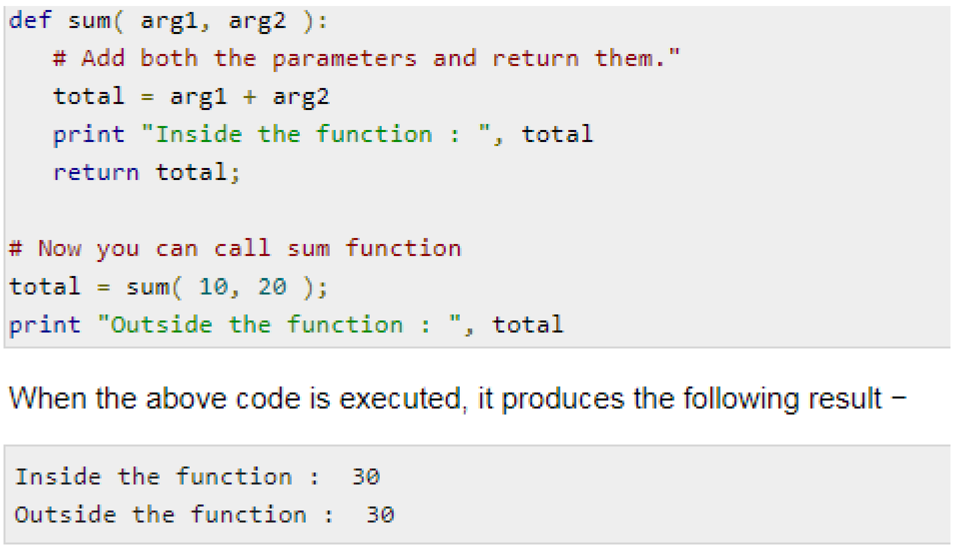
```
hello
world
python
```

Q5. Discuss return statement. Can we return more than one values with return statement?

Ans:

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −



```
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2
   print "Inside the function : ", total
   return total;

# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function :  30
Outside the function :  30
```

The Return Statement:

The return statement is used to return a value from the function to a calling function. It is also used to return from a function i.e. break out of the function.

Example:

Program to return the minimum of two numbers:

```
def minimum(a,b):
  if a<b:
     return a
  elif b<a:
    return b
  else:
    return 'both numbers are equal'
min=minimum(100,85)
print(min,'is minimum')
```

Output:

```
85 is minimum
```

Returning multiple values: It is possible in python to return multiple values

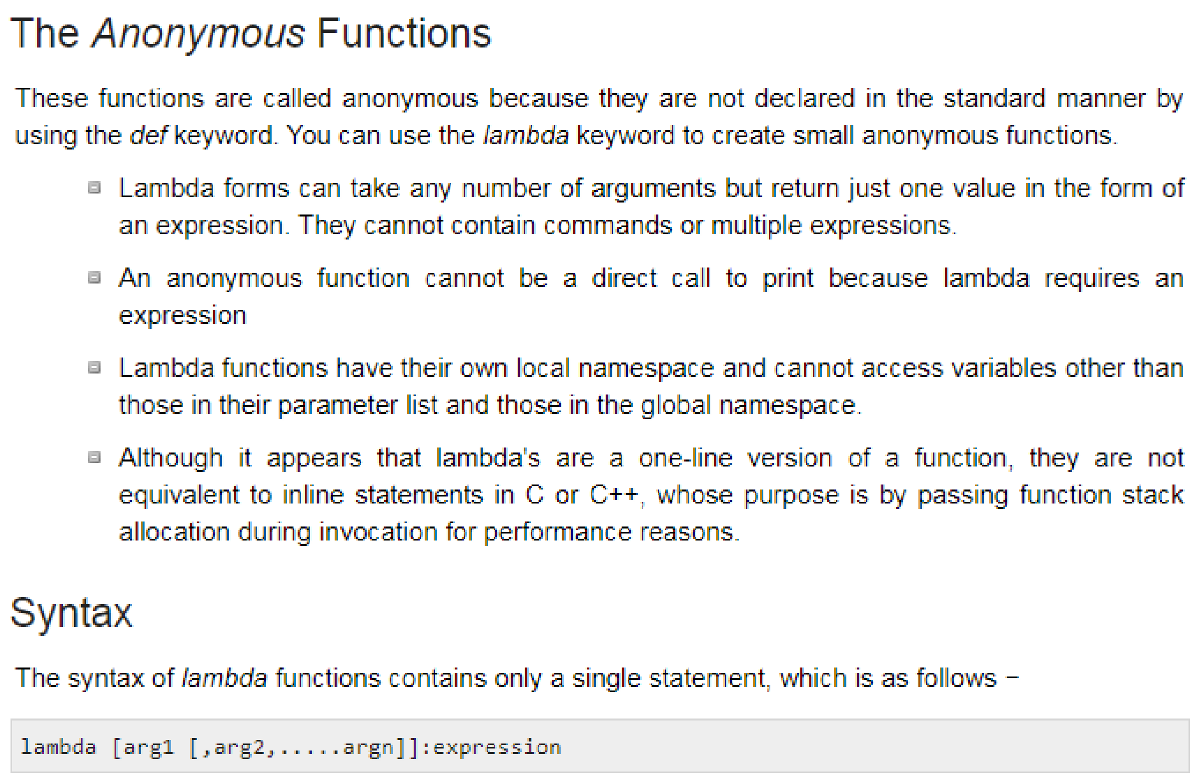
```
def compute(num):
        return num*num,num**3 # returning multiple values
```

```
square,cube=compute(2)

print(“square = “,square,”cube =”,cube)
```

Q4. What is anonymous function? How it is created?

Ans:



The *Anonymous* Functions

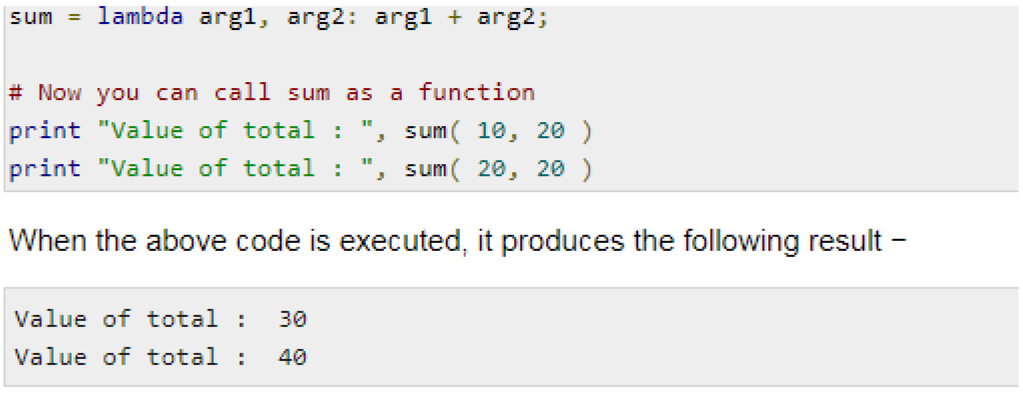
These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

```
lambda [arg1 [,arg2,.....argn]]:expression
```



```
sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

When the above code is executed, it produces the following result −

```
Value of total :  30
Value of total :  40
```

Program to find cube of a number using lambda function

cube=lambda x: x*x*x

print(cube(2))

Output:

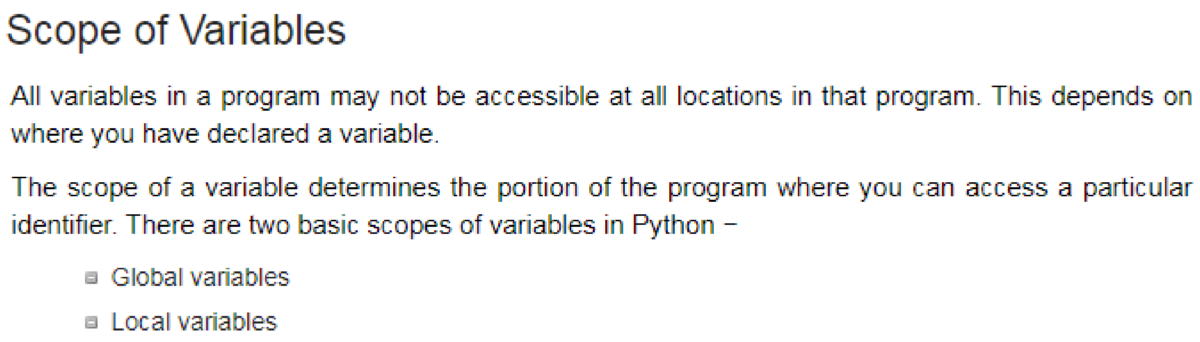
8

Note:

- The statement cube=lambda x: x*x*x creates a lambda function called cube, which takes a single argument and returns the cube of a number.
- Lambda function does not contain a return statements
- It contains a single expression as a body not a block of statements as a body

Q6. Define scope of a variable. Also differentiate local and global variable.

Ans:



Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

- Global variables
- Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example

```
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print "Inside the function local total : ", total
   return total;

# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result −

```
Inside the function local total :  30
Outside the function global total :  0
```

1.Program to access a local variable outside a functions

```
def demo():
   q=10
   print("the value of local variable is",q)
demo()
print("the value of local variable q is:",q)  #error, accessing a local variable outside the scope will cause an error
```

Output:

```
the value of local variable is 10
Traceback (most recent call last):
  File "C:/Users/Students/AppData/Local/Programs/Python/Python38-32/vbhg.py", line 5, in <module>
    print("the value of local variable q is:",q)
NameError: name 'q' is not defined
```

2. Program to read global variable from a local scope

```
def demo():
        print(s)
s='I love python'
demo()
```

Output:

```
I love python
```

3. Local and global variables with the same name

```
def demo():
        s='I love python
        print(s)
s='I love programming'
demo()  # first function is called, after that other satements
print(s)
```

Output:

```
I love python
I love programming
```

The Global Statement

Global statement is used to define a variable defined inside a function as a global variable. To make local variable as global variable, use global keyword

Example:

```
a=20
def display():
        global a
        a=30
        print('value of a is',a)
```

display()

print('the value of an outside function is',a)

Output:

value of a is 30

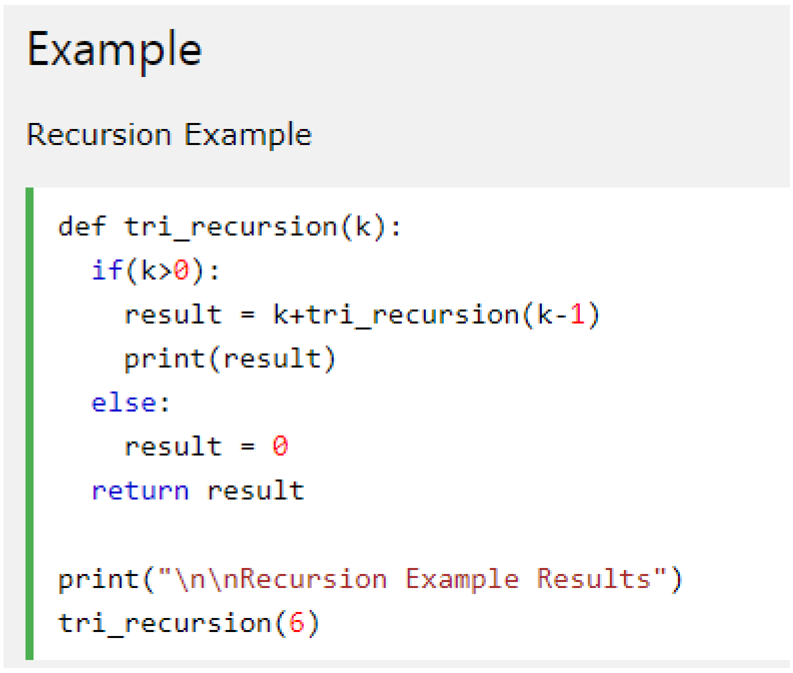
the value of an outside function is 30

Note:

Since the value of the global variable is changed within the function, the value of 'a' outside the function will be the most recent value of 'a'

Q 7. What is recursion? **Explain with example.Ans:**

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.



Example

Recursion Example

```
def tri_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)
```

Q Program to find the factorial of a number using recursion

```
def factorial(n):
        if n==0: # base condition
                return 1
        return n * factorial(n-1)
print(factorial(5))
```

Output:

120

Note: Recursion ends when the number n reduces to 1. This is called base condition. Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

Q8. **Program to find the fibonacci series using recursion.**

Ans:

```
def fibonacci(n):
   if(n <= 1):
      return n
   else:
      return(fibonacci(n-1) + fibonacci(n-2))
n = int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i in range(n):
   print fibonacci(i),
```

Q9. **Program to find the GCD of two numbers using recursion.**

Ans:

```
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:"))
GCD=gcd(a,b)
print("GCD is: ")
print(GCD)
```

Q10. **Program to find the sum of elements in a list recursively.**

Ans:

```
def sum_arr(arr,size):
   if (size == 0):
     return 0
   else:
     return arr[size-1] + sum_arr(arr,size-1)
n=int(input("Enter the number of elements for list:"))
a=[]
for i in range(0,n):
    element=int(input("Enter element:"))
    a.append(element)
print("The list is:")
print(a)
print("Sum of items in list:")
b=sum_arr(a,n)
print(b)
```

Q11. **Program to check whether a string is a palindrome or not using recursion.**

Ans:

```
def is_palindrome(s):
    if len(s) < 1:
        return True
    else:
        if s[0] == s[-1]:
            return is_palindrome(s[1:-1])
        else:
            return False
a=str(input("Enter string:"))
if(is_palindrome(a)==True):
    print("String is a palindrome!")
else:
    print("String isn't a palindrome!")
```