FUNCTIONS

Function Introduction

A function is a subprogram that acts on data and often returns a value

OR

A function is a programming block of codes which is used

1. **to perform a single, related task.**
2. **It only runs when it is called.**
3. **We can pass data/value, known as parameters, into a function.**
4. **A function can return data as a result.**

Advantages of Using Functions:

1.Program development made easy and fast : Work can be divided among project members thus implementation can be completed fast.

2. Program testing becomes easy

3. Code sharing becomes possible

4.Code re-usability increases

5. Increases program readability

6.Function facilitates procedural abstraction : Once a function is written, it serves as a black box. All that a programmer would have to know to invoke a function would be to know its name, and the parameters that it expects

7.Functions facilitate the factoring of code : A function can be called in other function and so on…

Types of Functions

- Built –in functions (function using Libraries)
- Functions defined in the modules(function using Libraries)
- User defined functions

Built-in functions

Pre-defined functions that are always available for use.

e.g.
print() ,len() ,input() ,ord() ,hex() ,type(), int() etc

Functions using libraries(System defined function or Built- in Functions)

String functions:

Method	Description
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
partition()	Returns a tuple where the string is parted into three parts

Syntax:

stringname.functionname()

e.g.

a.isupper()

Functions using libraries(System defined function or Built- in Functions)

String functions:

Method	Description	Syntax
capitalize()	Converts the first character to upper case	str.capitalize()
center()	string padded with specified fillchar.	str.center(width,[fillchar]) #Width means length of the string
count()	Returns the number of times a specified value occurs in a string	str.count(substring,[start], [end])
endswith()	Returns true if the string ends with the specified value	str.endswith(substring,[start],[end])
format()	Formats specified values in a string	print("Hello {}, your rollno {}.".format("abc", 23)) print("Hello {name}, your rollno {rn}." .format(name="abc", rn=23))
find()	Searches the string for a specified value and returns the position(index) of where it was found	str.find(sub,[start],[end])
index()	Searches the string for a specified value and returns the position of where it was found	str.index(sub,[start],[end])

The only **difference** is that **find**() method returns -1 if the substring is not **found**, whereas **index**() throws an exception.

Functions using libraries(System defined function or Built- in Functions)

String functions:

Method	Description
replace()	Returns a string where a specified value is replaced with a specified value
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

Functions defined in modules

Pre-defined functions that are in particular modules and can only be used when corresponding module is imported.

e.g.
import math
sin(), cos(),abs(),floor() etc

Modules

Modules

- is a part of a program
- used for dividing up the program into smaller, more easily understood, reusable parts.

A module is a file containing

1. Python definitions and statements.
2. Module can define functions, classes and variables.
3. A module can also include runnable code.
4. Grouping related code into a module makes the code easier to understand and use.

Functions using libraries/ Functions defined in Modules

Mathematical functions:

Mathematical functions are available under math module.To use mathematical functions under this module, we have to import the module using import math.

For e.g.

To use sqrt() function we have to write statements like given below.

```
import math
r=math.sqrt(4)
print(r)
```

OUTPUT :

2.0

Functions using libraries(Functions defined in Modules)

Functions available in Python Math Module

Function	Description	Example
ceil(n)	It returns the smallest integer greater than or equal to n.	math.ceil(4.2) returns 5
factorial(n)	It returns the factorial of value n	math.factorial(4) returns 24
floor(n)	It returns the largest integer less than or equal to n	math.floor(4.2) returns 4
fmod(x, y)	It returns the remainder when n is divided by y	math.fmod(10.5,2) returns 0.5
exp(n)	It returns e**n	math.exp(1) return 2.718281828459045
log2(n)	It returns the base-2 logarithm of n	math.log2(4) return 2.0
log10(n)	It returns the base-10 logarithm of n	math.log10(4) returns 0.6020599913279624
pow(n, y)	It returns n raised to the power y	math.pow(2,3) returns 8.0
sqrt(n)	It returns the square root of n	math.sqrt(100) returns 10.0
cos(n)	It returns the cosine of n	math.cos(100) returns 0.8623188722876839
sin(n)	It returns the sine of n	math.sin(100) returns -0.5063656411097588
tan(n)	It returns the tangent of n	math.tan(100) returns -0.5872139151569291
pi	It is pi value (3.14159...)	It is (3.14159...)
e	It is mathematical constant e (2.71828...)	It is (2.71828...)

User defined functions

These are defined by the programmer according to his/her requirements /needs.

Creating a Function (user defined)

A function is defined using the def keyword in python.

Syntax:

arguments/parameters-means values given to function

Name of the function

colon at the end means it requires a block

```
def <function_name>([arguments/parameters]):
    ["""" function's docstring """"]
        <statement>
         [statement]
          [statement]
              :
              :
```

def means function definition is starting

Function block/ definition/ working of the function

e.g.

```
def fun():
    print("Hello")
    print("world")
```

```
def cals(a):
    b=2*a
    print(b)
```

Few terms : Defining functions

- **Function header-** first line of the function that begins with the keyword def and end with the (:) colon. It specifies the name of the function and its parameters

- **Parameters**-values that are listed within the parentheses of a function header

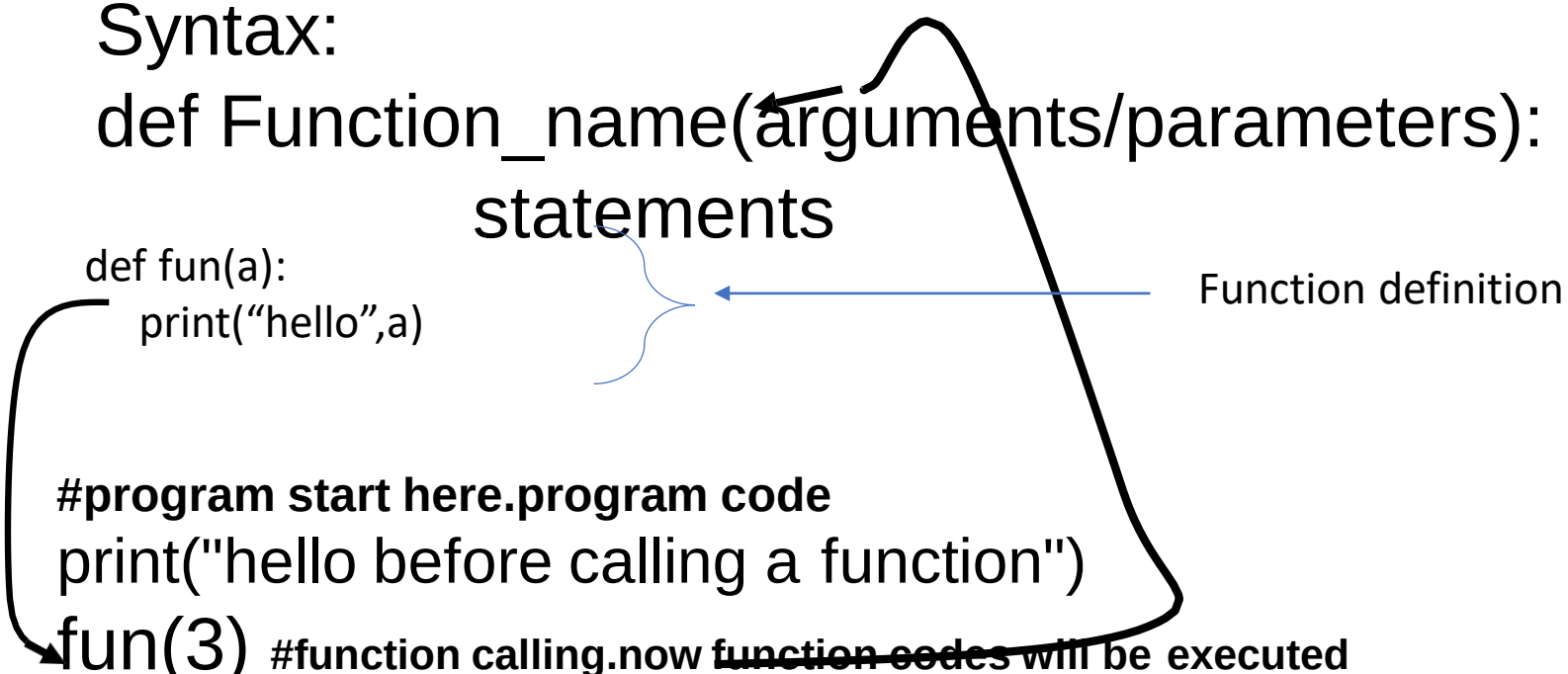
- **Indentation**- blank space in the beginning of a statement within a block. All statements within the block have same indentation.

Calling a Function(user defined)

Syntax:

def Function_name(arguments/parameters):

statements



4 Parts To Create User Defined Functions

- Function definition
- Arguments(those variables which are use in function calling)
- Parameters(those variables which are use in function definition)
- Function Calling

Flow of execution in function Calling

```
1   def fun(a,b):
2           c=a+b
3           print(c)
4   x=2
5   y=4
6   fun(x,y)
```

1-4-5-6-1-2-3

void functions- those functions which are not returning values to the calling function

Flow of execution in function Calling

```
1  def fun(a,b):
2          c=a+b
3          return c
4  x=int(input())
5  y= int(input())
6  z=fun(x,y)
7  print(z)
```

1-4-5-6-1-2-3-6-7

Non void functions- those functions which are returning values to the calling function

Flow of execution in function Calling

```
1.def fun(a,b):
2.      c=a+b
3.      print(c)
4.      return
5.x=2
6.y=4
7.z=fun(x,y)
8.print(z)
```

1-5-6-7-1-2-3-4-8

Void functions- those functions which are not returning values to the calling function.
We may use return but it will return none value to the function call

Void functions and Non void functions

```
def fun(a,b):
        c=a+b
        print(c)
        return
x=2
y=4
z=fun(x,y)
print(z)
```

Output
6
None

We may use return but it will return none value to the function call

```
def fun(a,b):
        return a+b, a, 5
x=2
y=4
z,w,q=fun(x,y)
print(z,w,q)
```

Output
6 2 5

Value return can be literal,variable , expression

Arguments/Parameters

Arguments- passed values in function call.
It can be of three types

1. Literals
2. Variables
3. Expressions

```
def fun(a,b):
        c=a+b
        print(c)

x=2
y=4
fun(x,y)          #variables
fun(5,6)          #literals
fun(x+3,y+6)      #expressions
```

Arguments

Arguments/Parameters

Parametes- received values in function definition.

It should be of variable types.

```
def fun(a,b):                    #parameters
        c=a+b
        print(c)


x=2
y=4
fun(x,y)
```

Returning Multiple Values from Functions

1. Received values as tuple

```
def fun(a,b):
        return a+b,a-b
x=2
y=4
z=fun(x,y)
print(z)
```

2. Unpack received values as tuple

```
def fun(a,b):
        return a+b,a-b
x=2
y=4
d,z=fun(x,y)
print(d,z)
```

```
def fun(x,y):   #Parameters- variables used in function definition - should be of variable type  here x and y are parameters
    z=x+y
    return z, z**3 ,5       #function can return values in the form of variable, expression, literal

#function calling
a=int(input("enter no1")) #1 a=4
b=int(input("enter no2")) #2    b=5function calling
q=fun(a,b)              #variable type arguments  (values which are used in function calling)
print(q)
t=fun(3,5)              #literal type arguments here 3 and 5 are literal type arguments
print(t)
z=fun(a+3,b+4)          #expression type arguments
print(z)

'''
Output
enter no1 5
enter no2 6
(11, 1331, 5)
(8, 512, 5)
(18, 5832, 5)
'''
```

Here output is in the form of tuples becoz function is returning multiple values and we r storing them in single variable

Returning Values from Functions

four possible combinations as functions

1. Void functions without arguments

```
def fun():
        a=2
        b=3
        print(a,b)
fun()
```

2. Void functions with arguments

```
def fun(a,b):
        print(a+b)
x=2
y=3
fun(x,y)
```

Returning Values from Functions

four possible combinations as functions

3. Non Void functions without arguments

```
def fun():
        a=2
        b=3
        return(a,b)
c=fun()
print(c)
```

4. Non Void functions with arguments

```
def fun(a,b):
        return(a+b)
x=2
y=3
c=fun(x,y)
print(c)
```

Types of Arguments

1. **Positional parameters(Required Arguments) -** these arguments must be provided for all parameters

```
def fun(a,b):
        c=a+b
        print(c)
x=10
y=3
fun(x,y)
fun(x,y,z)  #wrong no of arguments passed
```

Types of Arguments

2. Default Arguments- if right parameter have default value then left parameters can also have default value. this argument can be skipped at the time of function calling

```
def fun(a,b,c=3):
        d=a+b+c
        print(d)
x=10
y=3
fun(x,y) #here c parameter value is 3
z=5
fun(x,y,z)   #here it will be take parameter c value as 5
```

Types of Arguments

3. Keyword Arguments(Named Arguments)- We can write arguments in any order but we must give values according to their name

```
def fun(a,b,c):
        print(a,b,c)
a=1
b=3
c=5
fun(b,c,a)
```

Output
3 5 1

```
def fun(a,b,c):
        print(a,b,c)
fun(b=3,c=4,a=2)
```

Output
2 3 4

Scope of Variables

Scope means –to which extent a code or data would be known or accessed.

There are three types of variables with the view of scope.

1. Global Scope- Name declared in main program . It is usable inside the whole program.

```
def fun(a,b):
    s = a+b
    print(s)

X=int(input("enter no1"))
Y=int(input("enter no2"))
fun(X,Y)


#here X and Y are global variable
```

```
def fun(a):
    s = a+Y
    retrun(s)

X=10
Y=20
Z=fun(X,Y)
Print(Z)


#here X and Y are global variable
```

Scope of Variables

Scope means –to which extent a code or data would be known or accessed.

1. Local Scope- Name declared in function body. It is usable within the function.

```
def fun():
    A=10
    B=20
    return(A+B)

C=fun()
print(C)
print(A)


#here A and B are local  variables and C is
global varibale
```

Variable's Scope in function

1. Non local variable – accessible in nesting of functions,using nonlocal keyword.

```
def fun1():
    x = 100
    def fun2():
        nonlocal x #change it to global or remove this declaration
        x = 200
    print("Before calling fun2: " + str(x))
    print("Calling fun2 now:")
    fun2()
    print("After calling fun2: " + str(x))


x=50
fun1()
print("x in main: " + str(x))
```

OUTPUT:

Before calling fun2: 100
Calling fun2 now:
After calling fun2: 200
x in main: 50

Variable's Scope in function

There are three types of variables with the view of scope.

1. Local variable – accessible only inside the functional block where it is declared.
2. Global variable – variable which is accessible among whole program using global keyword.
3. Non local variable – accessible in nesting of functions,using nonlocal keyword.

Local variable program:

```
def fun():
    s = "I love India!" #local variable
    print(s)

s = "I love World!"
fun()
print(s)
```

Output:

I love India!
I love World!

Global variable program:

```
def fun():
    global s #accessing/making global variable for fun()
    print(s)
    s = "I love India!“ #changing global variable's value
    print(s)

s = "I love world!"
fun()
print(s)
```

Output:

I love world!
I love India!
I love India!

Name Resolution

.

Resolving Scope of a name: LEGB Rule

1. Local
2. Enclosed
3. Global
4. Built in

.

CASE 1

.

Variable in global scope not in local scope

```
def fun1(x,y):
        s=x+y
        print(num1)
        return s

num1=100
num2=200
sm=fun1(num1,num2)
print(sm)
```

Name Resolution

.

Resolving Scope of a name: LEGB Rule

1. Local
2. Enclosed
3. Global
4. Built in

.

CASE 2

.

Variable neither in in local scope nor in global scope

```
def fun():
        print("hello",n)

fun()

#
```

Output:

Name error: name 'n' is not defined

Name Resolution

.

Resolving Scope of a name: LEGB Rule

1. Local
2. Enclosed
3. Global
4. Built in

CASE 3

.

Variable name in local scope as well as in global scope

.

```
def fun():
        a=10
        print(a)

a=5
print(a)
fun()
print(a)
# output
5
10
5
```

Name Resolution

.

Resolving Scope of a name: LEGB Rule

1. Local
2. Enclosed
3. Global
4. Built in

.

```
def fun():
        global a
        a=10
        print(a)

a=5
print(a)
fun()
print(a)
# output
5
10
10
```

CASE 4

This case is discouraged in good programming

.

Using global variable inside local scope

Function

Parameters / Arguments

These are specified after the function name, inside the parentheses. Multiple parameters are separated by comma.The following example has a function with two parameters x and y. When the function is called, we pass two values, which is used inside the function to sum up the values and store in z and then return the result(z):

```
def sum(x,y):  #x, y are formal arguments
    z=x+y
    return z #return the result

x,y=4,5
r=sum(x,y) #x, y are actual arguments
print(r)
```

Note :- 1. Function Prototype is declaration of function with name ,argument and return type.

2. A formal parameter, i.e. a parameter, is in the function definition. An actual parameter, i.e. an argument, is in a function call.

Function Arguments

Functions can be called using following types of formal arguments −

- Required arguments - arguments passed to a function in correct positional order
- Keyword arguments - the caller identifies the arguments by the parameter name
- Default arguments - that assumes a default value if a value is not provided to argu.

```
#Required arguments/Positional arguments
def square(x):
    z=x*x
    return z

print(square(3)
#In above function square() we have to  definitely need to pass some value
```

```
#Keyword arguments
def fun( name, age ):
   print (name,age)
   return;

fun( age=15, name="mohak" )
# value 15 and mohak is being passed  to relevant argument based on keyword  used for them.
```

```
#Default arguments
def sum(x=3,y=4):
    z=x+y
    return z

r=sum() #default value of x and y is being used  when it is not passed
print(r)
r=sum(x=4)
print(r)
r=sum(y=45)
print(r)
```

Mutable/Immutable properties of data w/r function

Everything in Python is an object,and every objects in Python can be either **mutable** or **immutable**.

Since everything in Python is an Object, every variable holds an object instance. When an object is initiated, it is assigned a unique object id. Its type is defined at runtime and once set can never change, however its state can be changed if it is mutable.

Means a **mutable** object can be changed after it is created, and an **immutable** object can't.

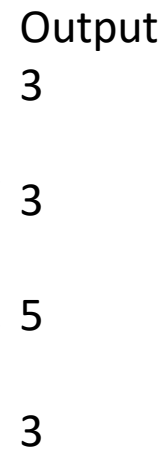
Mutable objects: list, dict, set, byte array
Immutable objects: int, float, complex, string, tuple, frozen set ,bytes

Passing Immutable type values to a function

e.g.

```
def fun(a):
    print(a)
    a=a+2
    print a

n=3
print(n)
fun(n)
print(n)
```



Passed value of n remains unchanged because it is immutable

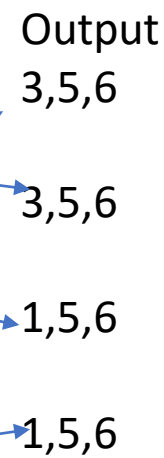
Passing mutable type value to a function

CASE-1

e.g.

```
def fun(a):
    print(a)
    a[0]=1
    print (a)

n=[3,5,6]
print(n)
fun(n)
print(n)
```



Value changed in function change the value in main program(because list change values at same address and only single value is changing)

Passing mutable type value to a function

CASE-2

e.g.

```
def fun(a):
    print(a)
    f=[7,8]
    a=f
    print (a)

n=[3,5,6]
print(n)
fun(n)
print(n)
```

Output

3,5,6

3,5,6

7,8

3,5,6

Value changed in function did not change for main program(because a is assigned a whole new list not a change of single values for whole program

Pass Mutable type values to a function

Arrays are popular in most programming languages like: Java, C/C++, JavaScript and so on. However, in Python, they are not that common. When people talk about Python arrays, more often than not, they are talking about Python lists. Array of numeric values are supported in Python by the array module.

e.g.

```
def dosomething( thelist ):
    for element in thelist:
        print (element)

dosomething( ['1','2','3'] )
alist = ['red','green','blue']
dosomething( alist )
```

OUTPUT:

```
1
2
3
red
green
Blue
```

Note:- List is mutable datatype that's why it treat as pass by reference.It is already explained in topic Mutable/immutable properties of data objects w/r function

Mutable/immutable properties of data objects w/r function

How objects are passed to Functions

#Pass by reference

```
def updateList(list1):
    print(id(list1))
    list1 += [10]
    print(id(list1))
n = [50, 60]
print(id(n))
updateList(n)
print(n)
print(id(n))
```

OUTPUT

```
34122928
34122928
34122928
[50, 60, 10]
34122928
```

#In above function list1 an object is being passed and its contents are changing because it is mutable that's why it is behaving like pass by reference

#Pass by value

```
def updateNumber(n):
    print(id(n))
    n += 10
    print(id(n))
b = 5
print(id(b))
updateNumber(b)
print(b)
print(id(b))
```

OUTPUT

```
1691040064
1691040064
1691040224
5
1691040064
```

#In above function value of variable b is not being changed because it is immutable that's why it is behaving like pass by value