Section-2: Design with Functions

Functions improve efficiency and reduce errors because of their,

1. Modularity - Break down a bigger problem into smaller functions (Top-Down Parsing)
2. Reusability - Python uses a **DRY** (Don't Repeat Yourself) principle. It means, **Write a function once**, and **Call the function any time, anywhere**.

Definition:

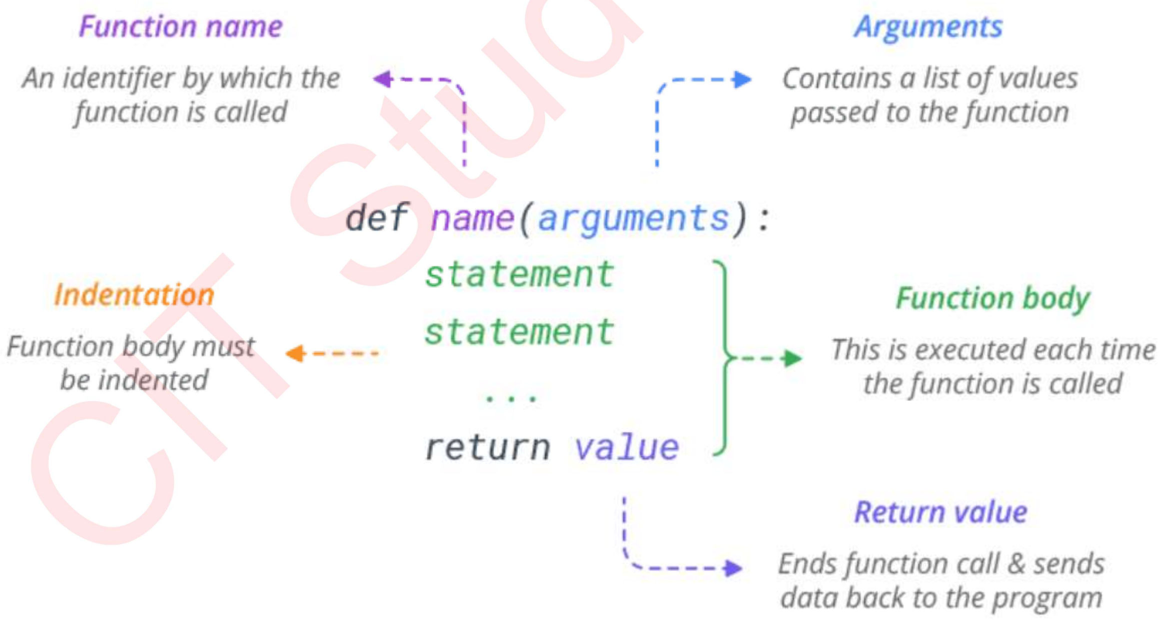
A function in Python is a block of statements that performs a specific task. Functions are useful when we must repeat the same task multiple times without rewriting the code.

- ➢ First, **'def'** keyword is used to **define a function**.
- ➢ Second, the **function must be called** to **run** it,
  - ○ We may pass arguments (values) to the function (optional).
- ➢ The function may return the result back to the calling area (optional).
- ➢ The function may return No value, Single value, or Multiple Values to the calling area (optional).

Syntax:

```
def function-name ( args ):
    statements
    return result
```

def - **keyword to define a function**
function-name - **the name of the function**
args - **list of values passed into the function (Optional)**
return - **will send the value back to calling area (Optional)**


Function name
An identifier by which the function is called
Arguments
Contains a list of values passed to the function
def name(arguments):
statement
statement
...
return value
Indentation
Function body must be indented
Function body
This is executed each time the function is called
Return value
Ends function call & sends data back to the program

Functions as Abstraction Mechanism:

Abstraction is used to hide the internal functionality of the function from the users. The users only interact with the basic implementation of the function, but the inner working is hidden. User is familiar with "what function does" but they don't know "how it does."
In Python, abstraction is used to hide irrelevant data in order to reduce the complexity. It also enhances the application efficiency.

Ex: Users just call a totalSalary() function to get a total salary but do not need to know how to do calculate it.

Categories of functions with examples

The functions in Python are categorized into 4 categories. Categories are decided based on **Argument Passing** and **Return value.**

Category	Pass Arguments form Calling Area to Called Function	Return Value form Called Function to Calling Area	Example

**1. No Pass
No Return**
	No	No	**add( )
			print(10+5)**

**2. No Pass
Yes Return**
	No	Yes	**add( )
			return (10+5)**

**3. Yes Pass
No Return**
	Yes	No	**add(a, b)
			print(a+b)**

**4. Yes Pass
Yes Return**
	Yes	Yes	**add(a, b):
			return (a+b)**

1. No Pass, No Return

- No arguments are passed to function from calling area
- No return value sent from the function to calling area

Example:

```
# Function category: No Arguments, No Return value
def add():  # Function Definition
   a,b=10,5
   print("sum = ",a+b)
# main program
add()   # Function call
Output: sum =  15
```

2. No Pass, Yes Return

- No arguments are passed to function from calling area
- The called function returns result back to calling area

Example:

```
# Function category: No Arguments, With Return value
def add():  # Function Definition
    a,b=10,5
    return a+b    # with return value
# main program
sum = add()    # Function call
print("Total = ",sum)
```

Output: **Total = 15**

3. Yes Pass, No Return

- Arguments are passed to function from calling area
- No return value sent from function to calling area

Example:

```
# Function category: With Arguments, No Return value
def add(a,b):  # Function Definition
    print("Total = ",a+b)    # No return value
# main program
add(10,5)    # Function call
```

Output: Total = 15

4. Yes Pass, Yes Return

- Arguments are passed to function from calling area
- Return value is sent from function to calling area

Example:

```
# Function category: With Arguments, With Return value
def add(a,b):  # Function Definition
    return a+b    # with return value
# main program
sum = add(10,5)    # Function call
print("Total = ",sum)
```

Output: Total = 15

Compare a Function, a Fruitful Function, and an Anonymous Function with an example for each.

The functions are, mainly, divided into **2 categories**:
1. Built-in Functions or Standard Library Functions,
2. User-defined Functions.

The user-defined functions are further classified into
- **Functions (Non-fruitful functions or Void functions)**
- **Fruitful functions**
- **Anonymous or Lambda Functions,**
- **also, Recursion Functions.**

Functions or Non-fruitful Functions:

Definition:

A function that doesn't return any value is called a **non-fruitful function** or a void function.

Syntax:

```
def function-name ( args ):
    statements
```

def - keyword to define a function
function-name - the name of the function
args - list of values passed into the function

Example: **Non-Fruitful function**

```
def add(a, b):
  print(a+b)
```

Fruitful functions:

Definition:

A fruitful function in Python is a **function that returns a value** after performing some operation.

We use the **'def'** keyword to define a function. A function takes input arguments (or parameters), processes them, and returns a result. The return value can be of any data type, such as int, float, double, string, or a custom class.
Note: The result is returned to the calling area as a "**fruit**".

Syntax:

```
def function-name ( parameters ):
    statements
    return result
```

def - keyword to define a function
function-name - the name of the function
parameters - list of values received in the function
return - will send the result back to calling area

Example: Fruitful function
```
def add(a,b):
    return a+b
sum = add(50,30)
print(sum)    # 80
```

Anonymous functions or Lambda functions:

Definition:

Anonymous function is a function that has NO NAME when it is defined. It is also called **a lambda** function. The '**lambda' keyword** is used to create the lambda functions. **Lambda functions are restricted to a single code or expression.**

A **lambda** function can take **any number of arguments** but only have **one expression**.

Syntax:

lambda arguments : expression

Example:
```
# Lambda Function
sum = lambda a,b: a+b
print(sum(10,5))
print(type(sum))    #<class 'function'>
```

What is a lambda function? Describe its characteristics with an example.

A lambda function is a function that has **NO NAME** when it is defined. It is also called **an anonymous** function. We use lambda functions when we need a nameless function for a **short period of time**.

The '**lambda**' is a keyword in Python for defining the anonymous function.

A **lambda** function can take **any number of arguments** but only have **one expression**.

- Lambda functions are stored in a variable and created at run time.
- We can pass the lambda function as an argument to a higher-order function (a function that takes in other functions as arguments).
- Lambda functions can be used inside another function.

Syntax:

```
lambda arguments : expression
```

Example-1: Assigning Lambda to a variable

```
sum = lambda a,b: a+b
print(sum(10,5))
```

Or

Example-2: Not Assigning Lambda to a variable

```
print( (lambda a, b: a + b)(10,5) )
```

=> The advantage of lambda functions is best used when they are used inside another function.

Example-3: * Real use of Lambda function is Inside Another Function *****

```
def power(n):
    return lambda x: x ** n
# SET power value
square = power(2)
cube = power(3)
# SEND base value
print(square(5))
print(cube(5))
```

Output:

```
25
125
```

The Characteristics of Lambda Functions:

The lambda function,

1. takes **many arguments** but has only **one expression**.
2. is restricted to **return a single expression**.
3. is used as an **anonymous function inside other functions**.
4. **does not need a return statement**, they always return a single expression.

Applications of Lambda Functions
also, **Applications of Higher Order Functions**

(**Note:** These examples can be written for both Lambda Functions and also for Higher Order Functions)

Lambda functions are used along with built-in functions like **filter(), map(), reduce()** etc.

➔ **Lambda with filter()**

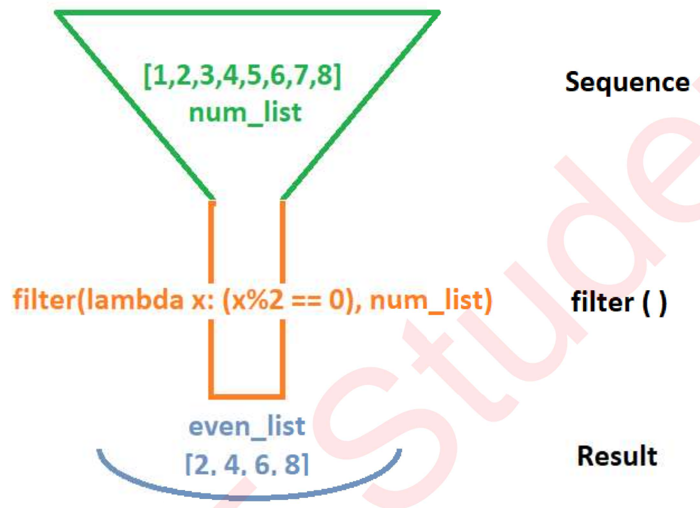
filter() function selects qualified elements **from** an iterable sequence based on the result of a function.

Syntax:

filter(function, iterable)
function - a function iterable - an iterable like sets, lists, tuples, etc.

Example:

```
num_list = [1,2,3,4,5,6,7,8]
even_list = list(filter(lambda x: (x%2 == 0) , num_list))
print(even_list) # [2, 4, 6, 8]
```



➔ **Lambda with map():**

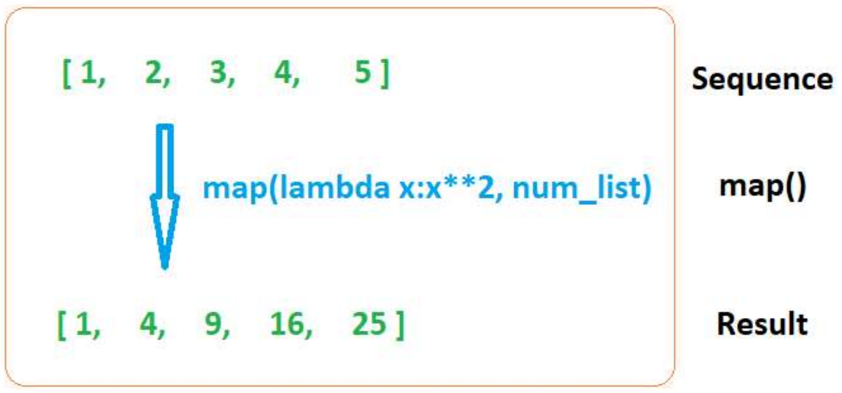
map() method applies a given function **to** each element of an iterable sequence (list, tuple etc.) and returns a sequence containing the results. We can pass more than one iterable sequence to the map() function.

Syntax:

map(function, iterable, ...)
function - a function iterable - an iterable like sets, lists, tuples, etc

Example:

```
num_list = [1,2,3,4,5]
squares_list=list(map(lambda x:x**2, num_list))
print(squares_list)      # [1, 4, 9, 16, 25]
```



➔ **Lambda with reduce():**

reduce() method applies a given function **to** all element of an iterable sequence (list, tuple etc.) and returns a single value. The reduce() method is similar to “for” loop in Python. The reduce() method is optimized and faster than “for” loop.

Note: The reduce() function in python is defined in “functools” module. We need to import “functools” before calling the reduce() function in our program.

Syntax:

```
from functools import reduce

reduce(function, iterable)
```

Example:

```
from functools import reduce
quantity = [10, 20, 30, 40]
result = reduce(lambda x, y: x + y, quantity)
print(result)
```

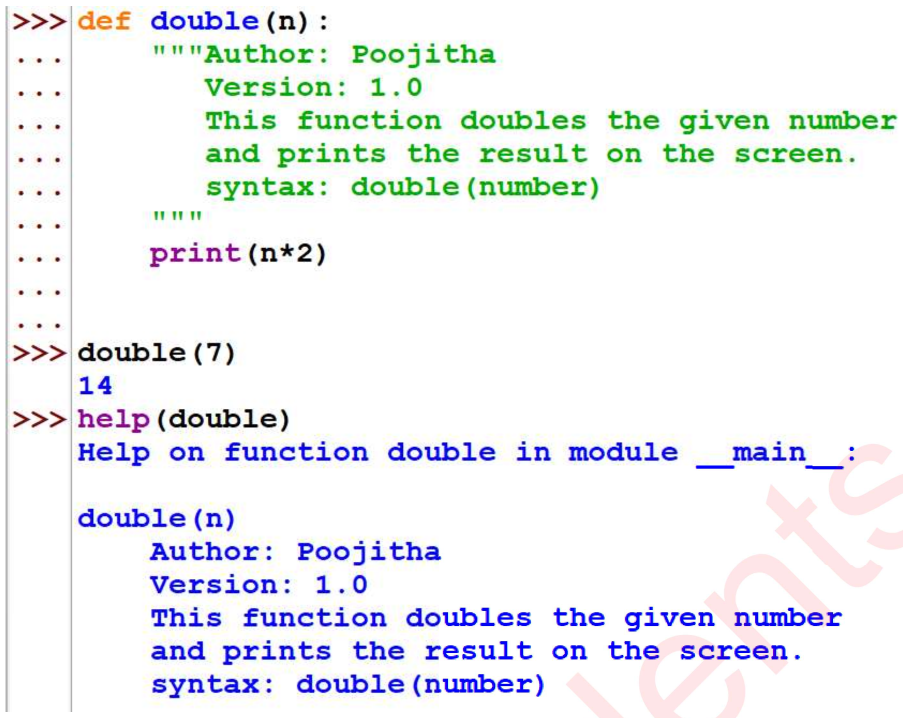
Output: 100

Explanation: This python program returns the sum of all values in the list as a single value. It uses the lambda function as (((10+20)+30)+40).

Docstring in Functions

Docstrings are enclosed in triple quotes for multi-line descriptions.

We can attach documentation to a function definition by including a string literal after the function header.

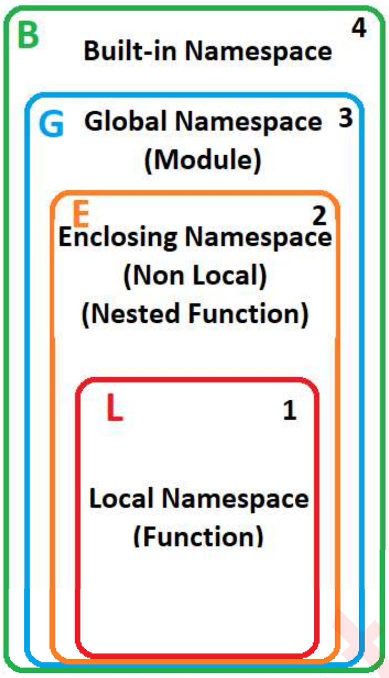


```
>>> def double(n):
...     """Author: Poojitha
...        Version: 1.0
...        This function doubles the given number
...        and prints the result on the screen.
...        syntax: double(number)
...     """
...     print(n*2)
...
...
>>> double(7)
14
>>> help(double)
Help on function double in module __main__:

double(n)
    Author: Poojitha
    Version: 1.0
    This function doubles the given number
    and prints the result on the screen.
    syntax: double(number)
```

Namespace, Scope and Lifetime of variables in Python

👉 A **Namespace** is a collection of defined names along with information about the object that each name refers to. Python has 4 types of namespaces.



Variable Lookup

👉The **Scope** is the region of the program where a variable is defined and can be accessed.

👉The **Lifetime** is the period of time during which a variable is available in the memory and can be accessed. The lifetime of a variable depends on its scope and how it was defined.

Note: A lifetime of a namespace depends upon the scope of objects, if the scope of an object ends, the lifetime of that namespace also ends.

LEGB Rule: When we read a variable, the Python interpreter will retrieve the variable in by looking up sequentially in the order of LEGB scope. That means, the first occurrence of this variable found in any of the scopes sequentially from, Local -> Enclosed -> Global -> BuiltIn, will be returned

Namespace	Scope (Accessible Area)	Lifetime (Accessible Period of Time)
Built-in Namespace	Python's built-in functions and variables (list, len, pow, round) can be accessed from anywhere in the program without using **import** statements. `>>> dir(__builtins__)`	Throughout the execution of the program. Usually determined by the Python Interpreter.

Global Namespace	Variables defined **outside of any function** have global scope. Accessed from anywhere in the program.	Available in memory during the execution of the whole program. They are only destroyed when the program terminates.
Enclosing Namespace	Variables defined in the outer function of a **nested function** have an enclosing scope. These variables can be accessed by the nested function.	Available during the nested function is being executed. Once the nested function finishes, the variables are destroyed and their memory is freed.
Local Namespace	Variables defined **within a function** have local scope. These variables can only be accessed within that function.	Available only during the execution of a function where they are defined. Once the function finishes, the variables are destroyed and their memory is freed.

The scope and lifetime of variables in Python are used to avoid naming conflicts, better use of memory, and write efficient code.

Example: Local scope variable

```
def rank():
  x = 100   # Local, x can be used only in rank() function
  print(x)
rank()
```

Output: 100

Example: Global scope variable

```
def rank():
    print(x)
    global y    # global keyword makes the variable global
    y=200       # y is global, can be used in all functions
x=100   # x is global, can be used in all functions
rank()
print(y)
```

Output: 100
200

Example: **Built-in scope variable**

```
from math import pi
def pival():
   print('Local scope: ', pi)
print('Global scope: ', round(pi))
pival()
```

Output:

```
Global scope:  3
Local scope:  3.141592653589793
```

Example: **Enclosed scope variable (also called Non-Local scope)**

```
# Enclosed/Non-Local scope in child()
def parent():
   a = 10
   def child():
       print('child ', a) #10, a has Enclosed or Non-Local scope in child()
   child()
parent()
```

Output:

```
Child  10
Parent  10
```

# Note: a in child() is neither Global nor Local. Hence, a is Enclosed in child()

```
# Local scope in child()
def parent():
   a = 10
   def child():
       a = 20
       print('Child ', a) #20, a has Local scope in child() as per LEGB rule
   child()
   print('Parent ',a)
parent()
```

Output:

```
Child  20
Parent  10
```

What are the different types of arguments (or parameter passing) in Python functions? Justify with suitable examples

The Function Arguments in Python are also called Formal arguments. We can call a function by using the following 4 types of formal arguments.

1. **Required arguments (Positional arguments)**
2. **Keyword arguments (Named arguments)**
3. **Default arguments**
4. **Variable-length arguments (or Arbitrary arguments)**

1. Required Arguments or Positional Arguments:

Explanation:
Required or Positional arguments are values assigned to the arguments by their position when the function is called. Ex: 1st value to 1st argument, 2nd value to 2nd argument, and so on.

👉Values must be required for all arguments according to their position. We must pass values in the same sequence defined in a function definition.

➔ By default, Python functions are called by using the positional arguments.

Application:
```
# Required arguments
def student(name, marks):
    print('Details:', name, marks)

# Function call
student('Sumanth', 15)
```

Output:
```
Details: Sumanth 15
```

2. Keyword Arguments:

Explanation:
The **Keyword Argument** is also called a **Named Argument**. We can change the sequence of keyword arguments by **using their name in function calls**. When we call functions in this way, the **order** (position) of the arguments **can be changed**.

Application:
```
# Keyword arguments
def student(name, marks):
    print('Details:', name, marks)
```

```
# Function Call: both Keyword arguments
student(name='Varma', marks=14)

# Function Call: 1 positional and 1 keyword
student('Venu', marks=12)

# Function Call: both Keyword arguments in different order
student(marks=15, name='Varshitha')
```

Output:

```
Details: Varma 14
Details: Venu 12
Details: Varshitha 15
```

3. **Default Arguments:**

Explanation:

The function **arguments can have default values**. We can assign default values to the arguments using the '=' (assignment) operator when defining a function. We can set a default value to any number of arguments.

- The default value will be used if we do not pass a value to that argument.
- If we pass a value, then the passed value will override the default value.

Application:

```
def student(name, marks, college="CIT"):
    print('Details:', name, marks, college)

# Passed only the required arguments
student('Vasanthi', 95)
```

Output:

```
Details: Vasanthi 95 CIT
```

4. Variable-length Arguments or Arbitrary Arguments:

Explanation:
We use **variable-length arguments** if we do not know the number of arguments to pass into a function. We can pass multiple arguments into a function. Internally all these values are represented in the form of a **tuple**.

Python has 2 types of Variable-length arguments as follows:

Type	**Variable-length Positional Arguments**	**Variable-length Keyword Arguments**
Declaration	**(*args)** * followed by 1 argument name	**(**kwargs)** ** followed by 1 argument name
Syntax	**def f-name(*args):** **statements**	**def f-name(**kwargs):** **statements**
Explanation	● Asterisk operator(*) is used ● We can pass multiple positional arguments to a function	● Unpacking operator(**) is used ● We can pass multiple keyword arguments to a function ● The kwargs are accessed using key-value pair (same as accessing a Dictionary).

Examples:

Application: Variable-length **Positional Arguments (*args)**

```
def add(*scores):
    sum = 0
    for i in scores:
        sum += i
    print("Total= ", sum)

# main program
# Function called with variable arguments
sum = add(60,50)    # 110
sum = add(60,50,70)      # 180
```

Output:
```
Total=  110
Total=  180
```

Application: **Variable-length Keyword Arguments (**kwargs)**

```
def totalmarks(**sub_marks):
    total = 0
    for i in sub_marks:
        # get subject name
        sub = i
        # get subject value
        marks = sub_marks[i]
        total = total+marks
        print(sub, "=", marks)
    print("Total (Variable KW Args)=",total)
# pass multiple keyword arguments
totalmarks(math=60, chem=50, python=70)
totalmarks(chem=50, math=60, python=70)
```

Output:

```
math = 60
chem = 50
python = 70
Total= 180
```

How to pass a list into a function? Explain with an example program.

We can pass any data type (list, dict, str, number, etc) as an argument into a function.

A list is a sequence or collection of many elements of the same or different data types.
When we pass a list into a function as an argument.
The passed list is still treated as a list inside the function.

Example:

```
def semester(subjects):
  for s in subjects:
    print(s)

# Passing list into a function
subjects = ["Chem", "Math", "Python"]   # List defined
semester(subjects)  # Passed list into the function
```

Output:

```
Chem
Math
Python
```

What is recursion in Python? Write a program to find the factorial of a given number using recursion.

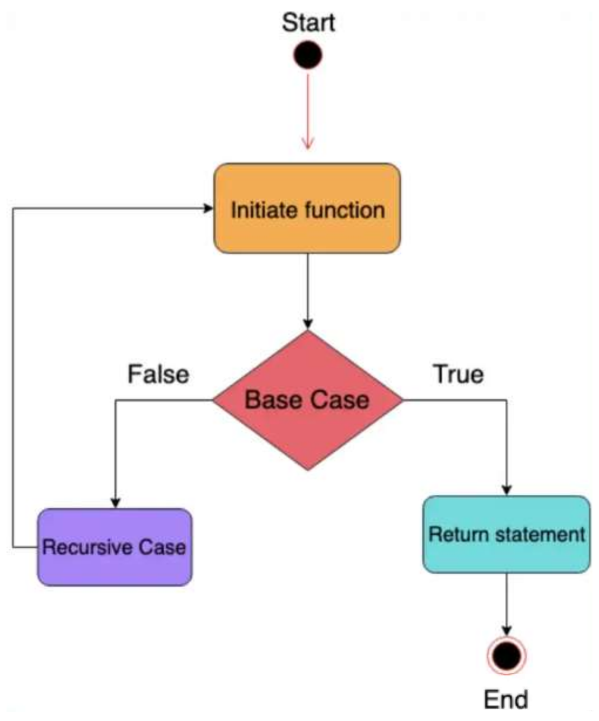
Definition:

A function called by itself repetitively is called a recursive function. The function call is termed a recursive call.

The recursive function will call itself multiple times until a condition is satisfied. The recursive functions should be used very carefully because, when a function is called by itself it enters into the infinite loop. And when a function enters into the infinite loop, the function execution never gets completed.

👉 We MUST define the condition to exit from the function call so that the recursive function gets terminated.

Flowchart of Recursive Function:



The recursive function has two main parts in its body,

1. **the base case** (condition) and
2. **the recursive case (recursive function call).**

The flow of execution of Recursive Function:

- first, the program checks the base case condition.
- If it is TRUE, the function returns and quits;
- otherwise, the recursive case is executed by calling the function recursively.

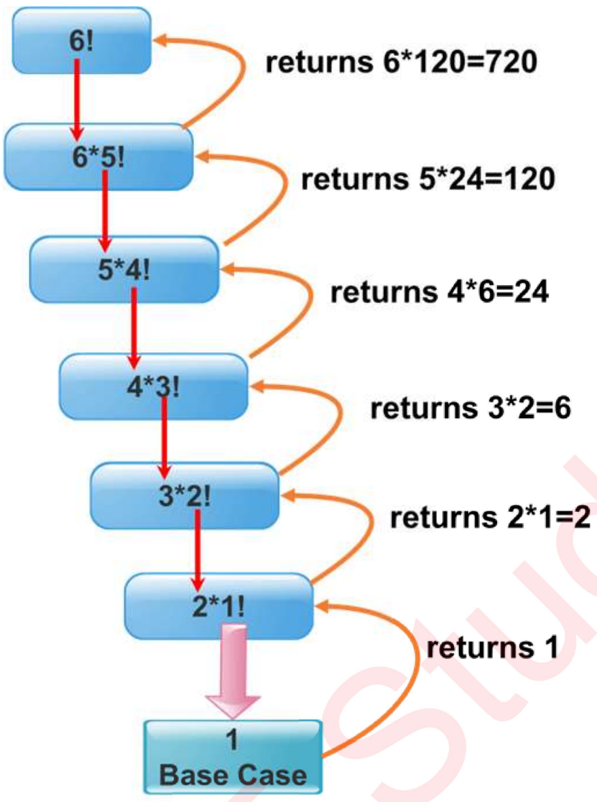
Syntax: Recursion in a Python

```
def recursive_function(argument)
{   # base case condition
    if base_case == True:
       return result
    # recursive case
    else:
       return recursive_function(argument) #recursive call
}
```

Application: Python Program to Find Factorial of a given integer number

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)  # Recursive call
print("Factorial is: ", fact(6))    # First Function call
```

Output: **Factorial is: 720**



Ex: The recursive function call execution **to find factorial of 6**.

Flow of recursion:
fact(6)
6 * fact(5)
6 * 5 * fact(4)
6 * 5 * 4 * fact(3)
6 * 5 * 4 * 3 * fact(2)
6 * 5 * 4 * 3 * 2 * fact(1)
6 * 5 * 4 * 3 * 2 * 1 = **720**

Application: **Find Fibonacci Series of a given number of terms using Recursion Function**

```
def fib(i):
    if (i == 0):
        return 0
    if (i == 1):
        return 1
    return fib(i - 1) + fib(i - 2)

n=int(input("Enter terms for Fibonacci series: "))
for i in range (n):
      print(fib(i),end=" ")
```

Output:
```
Enter terms for Fibonacci series: 7
0 1 1 2 3 5 8
```

Advantages of Recursive Functions:

1. We can Reduce the length of the code,
2. We can Improve the Readability of code,
3. We can Solve complex problems.

Disadvantages of Recursive Functions:

1. Need more memory and time for execution,
2. Debugging is difficult.

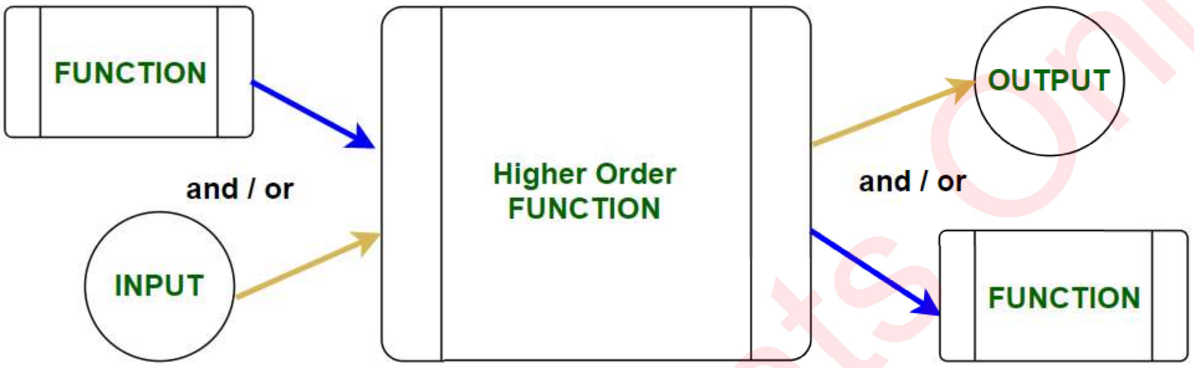
What are Higher Order Functions? Explain them with an example program.

Definition:

In Python, a higher-order function is a function that takes one or more functions as arguments or returns a function as its result.

Hence, the functions that operate with another function are known as Higher-order Functions.

Note: The higher-order functions can manipulate other functions by treating them like any other variable.



Characteristics of higher-order functions:

- A function is an instance of the Object type.
- We can store the function in a variable.
- We can pass the function as a parameter to another function.
- We can return the function from a function.
- We can store them in data structures such as hash tables, and lists, etc.

Function as an object:

In Python, a function can be assigned to a variable. This assignment does not call the function, instead, a reference to that function is created.

Example:

```
# Function as an object variable
def caps(name):
    return name.upper()
# Assigning function to a variable
upper_name = caps
print("Function object upper_name = ", upper_name('Dhanush'))
```

Output:

```
Function object upper_name =  DHANUSH
```

Explanation:
In this example, the caps() function is assigned to a variable called upper_name. The upper_name is a function object that points to the function caps().

Passing Function as an argument to other function:
Functions are like objects in Python. Therefore, the functions can be passed as arguments to other functions. In the following example, we have created a function speak() that takes another function as an argument.

Example:
```
# Function passed as an argument to other functions
def caps(name):
    return name.upper()
def smalls(name):
    return name.lower()
def convert(farg):
    # storing the function in a variable
    result = farg("Function passed as an argument.")
    print(result)
convert(caps)
convert(smalls)
```

Output:
```
FUNCTION PASSED AS AN ARGUMENT.
function passed as an argument.
```

Returning function:
Since the functions are objects, we can return a function from another function.
Example:
```
# Functions that return another function
def add_salary(s):      # s=1000
    def add_bonus(b):   # b=100
        return s + b
    return add_bonus
sal = add_salary(1000) # sal = add_bonus
print("Total Salary & Bonus is ",sal(100))
```

Output:
```
Total Salary & Bonus is  1100
```

Functions Return Multiple Values: A function in Python can also return **Multiple values.**
[Note: Multiple values cannot be returned from a function in C, C++, or Java. **]**

1. **Return value as Tuple** - A Tuple is a sequence of items separated by a comma with or without (). Tuples are Ordered, Immutable, Indexed, Allows Duplicates.

```
def detailsTuple():
    semester = "CIT Sem-2 2023"
    students = 600
    #return semester, students # Return a tuple without ()
    return (semester, students)  # Return a tuple with ()
# Returns as Tuple (Method Call)
sem, std = detailsTuple() # Assigning returned tuple
print(sem, std)
```
Output: CIT Sem-2 2023 600

2. **Return value as List** - List is a sequence/collection of values of different data types enclosed in [ ]. Tuples are Ordered, Mutable, Indexed, Allows Duplicates.

```
def detailsList():
    semester = "CIT Sem-2 2023"
    students = 600
    return [semester, students] # Returns a list
# Returns as List   (Method Call)
slist = detailsList()
print(slist)
```
Output: ['CIT Sem-2 2023', 600]

3. **Return value as Dict** - A Dictionary is a sequence/collection of Key-Value pairs of different data types enclosed in { }. Dict is Ordered, Mutable, Not-Indexed, No Duplicate Keys but Duplicate Values are allowed.

```
def detailsDict():
    d = dict()
    d["semester"] = "CIT Sem-2 2023"
    d["students"] = 600
    return d # Returns the dictionary
# Returns as Dictionary   (Method Call)
sdict = detailsDict() # Assigning returned dictionary
print(sdict)
```
Output: {'semester': 'CIT Sem-2 2023', 'students': 600}

4. **Return value as Object** - We can create a class (similar to a struct in C) to hold multiple values and return an object of the class.

```
# Function returns multiple values using Class-Object
class FirstYear:
    def __init__(self):
        self.semester = "CIT Sem-2 2023"
        self.students = 600

# Method returns multiple values using Class-Object
def details():
    return FirstYear()  # Returns Class-Object

s = details()
print(s.semester)
print(s.students)

Output:   CIT Sem-2 2023
          600
```

5. **Return value using yield** - The yield keyword generates a sequence of values, one value at a time. To return multiple values from a generator function, you can use the **yield** keyword to yield each value in each turn and continues until the generator function completes execution or encounters a return statement.

```
# Function returns multiple values using yield
def get_data():
    yield 522007
    yield 'CIT'
    yield [60,80,100]

# Method call
data = get_data()
print(next(data))  # Prints 522007
print(next(data))  # Prints 'CIT'
print(next(data))  # Prints [60,80,100]
Output:   522007
          CIT
          [60, 80, 100]
```