Python Functions :
Python def keyword is used to define a function, it is placed before a function name that is provided by the user
to create a user-defined function. In Python, a function is a logical unit of code containing a sequence of statements
indented under a name given using the “def” keyword. In Python def keyword is the most used keyword.

In [1]:
```
# defining function
def func():
    print("Hello")

# calling function
func()
```

```
Hello
```

In [2]:
```
# Python3 code to demonstrate
# def keyword

# function for subtraction of 2 numbers.
def subNumbers(x, y):
    return (x-y)

# main code
a = 90
b = 50

# finding subtraction
res = subNumbers(a, b)

# print statement
print("subtraction of ", a, " and ", b, " is ", res)
```

```
subtraction of  90  and  50  is  40
```

In [3]:
```
# Python program to print first 10 prime numbers

# A function name prime is defined
# using def
def fun(n):
    x = 2
    count = 0
    while count < n:
        for d in range(2, int(x ** 0.5) + 1):  # check divisibility only up to
            if x % d == 0:
                break  # if divisible, it's not prime, so break the loop
        else:
            print(x)  # prime number
            count += 1
        x += 1

# Driver Code
n = 10

fun(n)
```

```
2
3
5
7
11
13
17
19
23
29
```

In [4]:
```
#Passing Function as an Argument
# A function that takes another function as an argument
def fun(func, arg):
    return func(arg)

def square(x):
    return x ** 2

# Calling fun and passing square function as an argument
res = fun(square, 5)
print(res)
```

```
25
```

In [6]:
```python
#Python def keyword example with *args
#In Python, *args is used to pass a variable number of arguments to a function
#The * allows a function to accept any number of positional arguments.
#This is useful when we are not sure how many arguments will be passed to the 

#Example:

def fun(*args):
    for arg in args:
        print(arg)

# Calling the function with multiple arguments
fun(1, 2, 3, 4, 5)



#Python def keyword example with **kwargs
#In Python, **kwargs is used to pass a variable number of keyword arguments to
#The ** syntax collects the keyword arguments into a dictionary, where the key
#and the values are the corresponding argument values. This allows the functio
#of named (keyword) arguments.


def fun(**kwargs):
    for k, val in kwargs.items():
        print(f"{k}: {val}")

# Calling the function with keyword arguments
fun(name="Alice", age=30, city="New York")
```

```
1
2
3
4
5
name: Alice
age: 30
city: New York
```

In [7]:
```python
class Person:
    # Constructor to initialize the person's name and age
    def __init__(self, name, age):
        self.name = name  # Set the name attribute
        self.age = age    # Set the age attribute

    # Method to print a greeting message
    def greet(self):
        print(f"Name - {self.name} and Age - {self.age}.")

# Create an instance of the Person class
p1 = Person("Alice", 30)

# Call the greet method to display the greeting message
p1.greet()


#def __init__(self) is the constructor method in Python, used in classes.
#It initializes the object's attributes when an instance of the class is creat
```

```
Name - Alice and Age - 30.
```

```
Benefits of Using pass in Functions
Helps Maintain Code Structure: We can set up the skeleton of your program
early on without having to worry about implementing every detail at once.
Keeps Syntax Valid: It prevents syntax errors when defining functions that
have no body yet. Without pass, Python would raise an indentation error if
the function body is left empty.
Facilitates Incremental Development: We can focus on high-level logic first
and implement specific functionality in small steps.
Improves Readability and Collaboration: When working in teams, using pass
clearly signals that a function is yet to be implemented, making the code
easier to read and understand.


Local Variable :
Declared inside a function and accessible only within that function.
Created when the function starts and destroyed when it ends.
Using a local variable outside its function results in an error.
If a variable with the same name exists globally, the local variable takes
precedence inside the function.

Global Variable :
Declared outside all functions and accessible throughout the program.
Can be accessed inside functions but not modified unless declared as global.
Changes inside a function will create a new local variable unless global is
used.
```

Comparision Basis	Global Variable	Local Variable
Definition	declared outside the functions	declared within the functions
Lifetime	They are created the execution of the program begins and are lost when the program is ended	They are created when the function starts its execution and are lost when the function ends
Data Sharing	Offers Data Sharing	It doesn't offers Data Sharing
Scope	Can be access throughout the code	Can access only inside the function
Parameters needed	parameter passing is not necessary	parameter passing is necessary
Storage	A fixed location selected by the compiler	They are kept on the stack
Value	Once the value changes it is reflected throughout the code	once changed the variable don't affect other functions of the program

Recursion involves a function calling itself directly or indirectly to solve a problem by breaking it down into simpler and more manageable parts. In Python, recursion is widely used for tasks that can be divided into identical subtasks.

In [8]:
```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))
```

```
120
```

Base Case and Recursive Case
Base Case: This is the condition under which the recursion stops. It is crucial to prevent infinite loops and to ensure that each recursive call reduces the problem in some manner. In the factorial example, the base case is n == 1.

Recursive Case: This is the part of the function that includes the call to itself. It must eventually lead to the base case. In the factorial example, the recursive case is return n * factorial(n-1).

In [9]:
```
#when defining methods within a class, the first parameter is always self.
#The parameter self is a convention not a keyword and it plays a key role in P

#Example:

class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Set instance attribute
        self.model = model  # Set instance attribute

    def display(self):
        return self.brand, self.model

# Create an instance of Car
car1 = Car("Toyota", "Corolla")

# Call the display_info method
print(car1.display())  # Output: This car is a Toyota Corolla
```

```
('Toyota', 'Corolla')
```

```
Explanation:

self in __init__: Used to assign values (brand and model) to the specific
instance (car1).
self in display_info: Refers to the same car1 instance to access its
attributes (brand and model).
Python automatically passes car1 as the first argument to display.
```

In [10]:
```
#Python Lambda Functions are anonymous functions means that the function is wi
#As we already know the def keyword is used to define a normal function in Pyt
#the lambda keyword is used to define an anonymous function in Python.

s1 = 'GeeksforGeeks'

s2 = lambda func: func.upper()
print(s2(s1))

#This code defines a lambda function named s2 that takes a string as its argum
#using the upper() method. It then applies this lambda function to the string
```

```
GEEKSFORGEEKS
```

```
Python Lambda Function Syntax
Syntax: lambda arguments : expression

lambda: The keyword to define the function.
arguments: A comma-separated list of input parameters (like in a regular
function).
expression: A single expression that is evaluated and returned.
```

```
Lambda functions provide a concise way to create small anonymous functions
without needing to define a formal function using def. They are particularly
useful in situations where a small function is needed temporarily or where
the function definition is straightforward and can be expressed in a single
line.
```

In [11]:
```python
# Example: Check if a number is positive, negative, or zero
n = lambda x: "Positive" if x > 0 else "Negative" if x < 0 else "Zero"

print(n(5))
print(n(-3))
print(n(0))
```

```
Positive
Negative
Zero
```

Feature	`lambda` Function	Regular Function (`def`)
Definition	Single expression with `lambda`.	Multiple lines of code.
Name	Anonymous (or named if assigned).	Must have a name.
Statements	Single expression only.	Can include multiple statements.
Documentation	Cannot have a docstring.	Can include docstrings.
Reusability	Best for short, temporary functions.	Better for reusable and complex logic.

In [12]:
```python
# Using lambda
sq = lambda x: x ** 2
print(sq(3))

# Using def
def sqdef(x):
    return x ** 2
print(sqdef(3))
```

```
9
9
```

In [13]:
```
#Combining lambda with list comprehensions enables us to apply transformations
#Example:

li = [lambda arg=x: arg * 10 for x in range(1, 5)]
for i in li:
    print(i())
```

```
10
20
30
40
```

In [14]:
```
#lambda with filter()
#The filter() function in Python takes in a function and a list as arguments.

#Example:

# Example: Filter even numbers from a list
n = [1, 2, 3, 4, 5, 6]
even = filter(lambda x: x % 2 == 0, n)
print(list(even))
```

```
[2, 4, 6]
```

In [15]:
```
#lambda with map()
#The map() function in Python takes in a function and a list as an argument.
#The function is called with a lambda function and a new list is returned whic
#the lambda-modified items returned by that function for each item.

#Example:


# Example: Double each number in a list
a = [1, 2, 3, 4]
b = map(lambda x: x * 2, a)
print(list(b))
```

```
[2, 4, 6, 8]
```

In [16]:
```
#lambda with reduce()
#The reduce() function in Python takes in a function and a list as an argument

#Example:

from functools import reduce

# Example: Find the product of all numbers in a list
a = [1, 2, 3, 4]
b = reduce(lambda x, y: x * y, a)
print(b)
```

```
24
```

In [1]:
```
#Python Inner Functions

#In Python, a function inside another function is called an inner function or
#Inner functions help in organizing code, improving readability and maintainin
#They can access variables from the outer function, making them useful for imp

#Example:


def fun1(msg): # outer function

    def fun2(): # inner function
        print(msg)
    fun2()

fun1("Hello")
```

```
Hello
```

```
Why Use Inner functions?
Inner functions provide several advantages:
Encapsulation: They help hide the inner logic from external access.
Code Organization: They make the code cleaner by grouping related
functionality.
Access to Enclosing Scope: Inner functions can access variables of the outer
function.
Closures: They allow functions to retain the state of their enclosing
function even after execution.
```

```
Decorators in Python

In Python, decorators are a powerful and flexible way to modify or extend
the behavior of functions or methods, without changing their actual code. A
decorator is essentially a function that takes another function as an
argument and returns a new function with enhanced functionality.
Decorators are often used in scenarios such as logging, authentication and
memorization, allowing us to add additional functionality to existing
functions or methods in a clean, reusable way.
```

```
Explanation:
decorator takes the greet function as an argument.
It returns a new function (wrapper) that first prints a message, calls
greet() and then prints another message.
The @decorator syntax is a shorthand for greet = decorator(greet).
```

In [3]:
```python
# A simple decorator function
def decorator(func):

    def wrapper():
        print("Before calling the function.")
        func()
        print("After calling the function.")
    return wrapper

# Applying the decorator to a function
@decorator

def greet():
    print("Hello, World!")

greet()
```

```
Before calling the function.
Hello, World!
After calling the function.
```