1. **Why are functions advantageous to have in your programs?**

Ans. **Code reuse**: Function allows us write a block of code once and use it multiple times throughout the program.

**Modularity**: Functions allows us to break our code in smaller, more mangeable pieces. Easy to read, understand, maintain, test and debug the code

**Encapsulation:** Functions allow you to encapsulate functionality and data within a single unit. This improves code organization and reduces the risk of unintended interactions between different parts of your program.

**Abstraction:** Functions allow you to abstract away implementation details and focus on the high-level logic of your program.

Example to understand the role of function in Encapsulation and Abstraction concept of the OOPs.

**class BankAccount:**

**def \_\_init\_\_(self, initial\_balance):**

**self.balance = initial\_balance**

**def deposit(self, amount):**

**self.balance += amount**

**def withdraw(self, amount):**

**if amount > self.balance:**

**raise ValueError("Insufficient funds")**

**self.balance -= amount**

**def get\_balance(self):**

**return self.balance**

account = BankAccount(1000) **##created object of BankAccount class**

**def make\_deposit(account, amount):**

**account.deposit(amount)**

**def make\_withdrawal(account, amount):**

**account.withdraw(amount)**

**def get\_account\_balance(account):**

**return account.get\_balance()**

make\_deposit(account, 500)

make\_withdrawal(account, 200)

balance = get\_account\_balance(account)

print(balance)# Output: 1300

**Encapsulation** refers to the practice of **hiding implementation details from the user** and **providing** a **public interface for interacting with an object**. In the **BankAccount** class, the balance is a private attribute that cannot be directly accessed or modified from outside the class. The public interface is provided by the **deposit**, **withdraw**, and **get\_balance** methods. This means that the user can interact with a **BankAccount** object without needing to know how it is implemented internally.

**Abstraction** refers to the **practice of presenting a simplified view of a complex system to the user**. In the above example, the **make\_deposit**, **make\_withdrawal**, and **get\_account\_balance** functions provide an **abstraction layer between the user and the BankAccount class**. They encapsulate the details of creating and interacting with a BankAccount object, and provide a simpler, more user-friendly interface for working with the account. This means that the user can work with a BankAccount object without needing to understand the details of how it is implemented or how it works internally.

In summary, encapsulation and abstraction **work together** to make the code more modular, maintainable, and flexible, by hiding implementation details and presenting a simplified, user-friendly interface.

1. **When does the code in a function run: when it's specified or when it's called?**

Ans. when it is called

1. **What statement creates a function?**

Ans. In Python, a function is created using the **def** keyword, followed by the function name, parentheses, and a colon.

Example:

**def add\_number(x,y):**

**return x+y**

1. **What is the difference between a function and a function call?**

Ans: A function is the definition of a particular set of actions to be performed, while a function call is the actual execution of those actions.

A function is a defined block of code that can be reused, while a function call is the act of actually invoking that code to perform a specific task.

1. **How many global scopes are there in a Python program? How many local scopes?**

**Ans:** In python, a scope is a region of the program where a particular name (variable, function, class etc) can be accessed without qualification. Python has two main types of scopes:

**Global scope:** outermost scope in a Python program, and it contains all the names that are defined at the top level of the program, outside of any functions or classes. Names in the global scope can be accessed from anywhere in the program.

**Local scope**: created whenever a function is called, and it contains all the names that are defined within that function. Names in a local scope can only be accessed within the function where they are defined, and they are destroyed when the function returns or completes execution.

an **example** to illustrate how scopes work in Python:

**# Global scope**

**x = 10**

**def my\_function():**

**# Local scope**

**y = 20**

**print("x inside function:", x) # Accessible from global scope**

**print("y inside function:", y) # Accessible from local scope**

**my\_function()**

**print("x outside function:", x) # Accessible from global scope**

**#print("y outside function:", y) # Not accessible from outside the function**

**When a name is referenced** in a Python program, Python **first looks** for that name in the **local scope**, **then** in **any enclosing local scopes** (for nested functions), and **finally** in the **global scope**. If the name is not found in any of these scopes, a **NameError** is raised.

1. **What happens to variables in a local scope when the function call returns?**

Ans: Once the function completes execution and returns, the local scope is **destroyed**, and any names defined within it are no longer accessible.

1. **What is the concept of a return value? Is it possible to have a return value in an expression?**

Ans: A return value is the value that a function or method returns when it is called. The return value is typically used to pass data or information from the function back to the calling code.

Example:

def add\_num(x,y):

return x+y

**yes**, we can use return values in expressions in Python.

result = **add\_num(3,4)\*2**

Print(result) ##output: 14

1. **If a function does not have a return statement, what is the return value of a call to that function?**

If a function does not have a return statement, the return value of a call to that function will be **None**. In Python, every function has to return some value, even if it doesn't explicitly use a **return** statement. If a function doesn't have a **return** statement, Python automatically returns **None** when the function completes.

Example:

def print\_msg():

Print(“hello”)

Now if we call the function and try to assign the return value to a variable, we’ll get ‘None’, like this:

Result = print\_msg()

Print(result) #output: None

**It's important to keep in mind that if we call a function that returns None, we won't be able to use the return value in any expressions or assignments**.

1. **How do you make a function variable refer to the global variable?**

**Ans:** Using **‘global ‘ keyword** to declare the variable as a global variable. This tells python to look for the variable in the global scale instead of the local scope.

Example**:**

x = 10 # global variable

def my\_function():

**global x** # declare x as global variable

x = 20 # modify the value of x

print("Inside function: x =", x)

my\_function()

print("Outside function: x =", x)

output:

Inside function: x = 20

Output function: x = 20

But if we simply refer the variable as local, then

x = 10 # global variable

def my\_function():

#global x # declare x as global variable

x = 20 # modify the value of x

print("Inside function: x =", x)

my\_function()

print("Outside function: x =", x)

output:

Inside function: x= 20

Output function: x = 10

Note: **Without** the **global** keyword, **Python** would **assume** that **x** is **a local variable** defined within the function, and any changes made to it would not affect the global variable. By **using** the **global** keyword, we tell **Python** to **use** the **global variable instead** of **creating** a **new local variable.**

1. **What is the data type of None?**

Ans: ‘None’ represents absence of a value. It is often used to indicate that a variable or expression does not have a value or has not been initialized.

Its datatype is ‘**NoneType’,** has only one possible value, which is ‘None’

1. **What does the sentence import areallyourpetsnamederic do?**

Ans: ‘import’ statement is **used to import modules and packages**{external filesor collection of files that contain python code}. When we import a module or package, we can access its functions, classes, and other objects from within our own python code.

1. **If you had a bacon() feature in a spam module, what would you call it after importing spam?**

Ans:

import spam

spam.bacon()

If we import the ‘spam’ module and then call its ‘bacon()’ function using the dot notation syntax. This tells python to look for bacon() function inside the ‘spam’ module.

1. **What can you do to save a programme from crashing if it encounters an error?**

Ans: **use try-except blocks**: to catch and handle exceptions that may occur during program execution

**Use logging:** to capture and record errors or unexpected events that occur during program execution. This can help you identify and fix issues before they cause your program to crash, and provide insight into what went wrong in case of a crash.

1. What is the purpose of the try clause? What is the purpose of the except clause?

Ans: they are used to catch and handle exceptions that may occur during program execution. By **wrapping potentially error-prone code in a try block** and **handling exceptions in an except block**, we can **prevent our program from crashing** and provide a more graceful **error handling** experience.