1. In Python, what is the difference between a built-in function and a user-defined function? Provide an

example of each.

Ans- In Python, a built-in function is a function that is already defined and available for use in the Python interpreter. These functions are part of the Python programming language and are typically included in the standard library. They provide commonly used functionality and can be accessed without the need for additional code or imports.

On the other hand, a user-defined function is a function that is created by the user or programmer. These functions are defined using the def keyword followed by a function name, parameters (if any), and a block of code that defines the function's behavior. User-defined functions are created to perform specific tasks or encapsulate reusable blocks of code.

# Using the built-in function 'len()'

my\_list = [1, 2, 3, 4, 5]

length = len(my\_list)

print(length) # Output: 5

# Defining a user-defined function 'square'

def square(x):

return x\*x

#calling the user-define function

square(9)

2. How can you pass arguments to a function in Python? Explain the difference between positional

arguments and keyword arguments.

Ans- In Python, arguments can be passed to a function by placing them within the parentheses when calling the function. Positional arguments are passed based on their order in the function's parameter list, while keyword arguments are passed using the parameter names. Positional arguments rely on the order, while keyword arguments provide clarity by explicitly specifying the parameter they correspond to. Mixing positional and keyword arguments is allowed, with positional arguments coming first. By using these argument types, programmers can customize the behavior of functions and make their code more flexible and readable.

def greet1(name, age):

print(f"Hello {name}! You are {age} years old.")

# Calling the function with positional arguments

greet1("Alice", 25)

def greet2(name, age):

print(f"Hello {name}! You are {age} years old.")

# Calling the function with keyword arguments

greet2(age=30, name="Bob")

def greet3(name, age):

print(f"Hello {name}! You are {age} years old.")

# Calling the function with a mix of positional and keyword arguments

greet3("Alice", age=25)

3. What is the purpose of the return statement in a function? Can a function have multiple return

statements? Explain with an example.

Ans- The purpose of the return statement in a function is to specify the value or values that the function should output or "return" when it is called. It allows the function to pass data back to the caller and terminate its execution.

A function can have multiple return statements, but only one of them is executed. When a return statement is encountered, it immediately exits the function, and the specified value is returned to the caller. Subsequent return statements in the function are not executed.

```

def check\_even\_odd(num):

if num % 2 == 0:

return "Even"

else:

return "Odd"

result1 = check\_even\_odd(4)

print(result1) # Output: "Even"

result2 = check\_even\_odd(7)

print(result2) # Output: "Odd"

```

4. What are lambda functions in Python? How are they different from regular functions? Provide an

example where a lambda function can be useful.

Ans- Lambda functions, also known as anonymous functions, are small, one-line functions in Python that are defined without a name. They are typically used for simple and concise operations that don't require a full-fledged named function. Lambda functions are defined using the lambda keyword and can take any number of arguments but can only have a single expression. Lambda functions can be particularly useful in situations where a simple, one-time operation or expression is needed, such as in list comprehensions, sorting, filtering, or passing a function as an argument to another function (e.g., map(), filter(), etc.). They provide a concise and efficient way to define and use functions without the need for a complete function definition.

square= lambda x: x\*x

square(5)

products = [

{'name': 'Product1', 'price': 50},

{'name': 'Product2', 'price': 30},

{'name': 'Product3', 'price': 40},

{'name': 'Product4', 'price': 20},

]

sort\_dic=sorted(products,key=lambda x: x['price'])

sort\_dic

5. How does the concept of "scope" apply to functions in Python? Explain the difference between local

scope and global scope.

Ans- In Python, scope refers to the visibility and accessibility of variables. Functions in Python have local scope, meaning variables defined within a function are only accessible within that function. Global scope applies to variables defined outside of any function, making them accessible throughout the module. Local variables are created when a function is called and destroyed when the function execution completes. Global variables exist throughout the entire program execution. Local variables take precedence over global variables with the same name. Understanding scope is crucial for managing variable accessibility and avoiding naming conflicts between local and global variables.

x = 10 # Global variable

def my\_function():

x = 20 # Local variable

print(x) # Output: 20

my\_function()

print(x) # Output: 10

6. How can you use the "return" statement in a Python function to return multiple values?

Ans- In Python, the return statement in a function can be used to return multiple values by separating them with commas. The values are returned as a tuple, and the caller can assign each returned value to separate variables.

def get\_name\_and\_age():

name = "Alice"

age = 25

return name, age

result = get\_name\_and\_age()

name, age = result

print(name) # Output: "Alice"

print(age) # Output: 25

7. What is the difference between the "pass by value" and "pass by reference" concepts when it

comes to function arguments in Python?

Ans- The concepts of "pass by value" and "pass by reference" do not directly apply. Python uses a mechanism called "passing arguments by assignment" or "pass by object reference." When a function is called, the reference to an object is passed as the argument. Immutable objects, like numbers and strings, behave similarly to "pass by value" since changes made inside the function create a new object. For mutable objects, like lists and dictionaries, changes made inside the function can affect the original object, resembling "pass by reference."

# Pass by Value

def modify\_value(num):

num = num + 1

print(num) # Output: 6

value = 5

modify\_value(value)

print(value) # Output: 5

#

#Pass by Refrence

def modify\_list(my\_list):

my\_list.append(4)

print(my\_list) # Output: [1, 2, 3, 4]

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Output: [1, 2, 3, 4]

8. Create a function that can intake integer or decimal value and do following operations:

a. Logarithmic function (log x)

b. Exponential function (exp(x))

c. Power function with base 2 (2

x

)

d. Square root

Ans- import math

def perform\_operation():

x = float(input("Enter a value: "))

operation = input("Enter the operation (log, exp, power, sqrt): ")

result = None

if operation == "log":

result = math.log(x)

elif operation == "exp":

result = math.exp(x)

elif operation == "power":

result = 2 \*\* x

elif operation == "sqrt":

result = math.sqrt(x)

else:

print("Invalid operation")

if result is not None:

print("Result:", result)

perform\_operation()

9. Create a function that takes a full name as an argument and returns first name and last name.

Ans- def extract\_first\_last\_name(full\_name):

names = full\_name.split()

first\_name = names[0]

last\_name = names[-1]

return first\_name, last\_name

full\_name = "Pranjal Kumar Sharma"

first\_name, last\_name = extract\_first\_last\_name(full\_name)

print("First Name:", first\_name)

print("Last Name:", last\_name)