In [ ]:

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

A) In Python, the main difference between a built-in function and a user-defined function lies in their origin and availability:

Built-in Functions:

Built-in functions are pre-defined functions that are provided by Python as part of its standard library.
These functions are readily available and can be used without any additional steps.
Examples of built-in functions include print(), len(), sum(), max(), min(), etc.
Here's an example of using a built-in function:

```
# Example of using a built-in function
numbers = [1, 2, 3, 4, 5]
total = sum(numbers)
print("Sum of numbers:", total)
```

User-defined Functions:

User-defined functions are functions created by the users themselves to perform specific tasks as per their requirements.

These functions are defined using the def keyword followed by the function name, parameters (if any), and a block of code.

User-defined functions can be reused throughout the code, improving modularity and maintainability.

Here's an example of a user-defined function:

```
# Example of a user-defined function
def greet(name):
    print("Hello,", name)

# Calling the user-defined function
greet("John")
```

In [ ]:

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

A)
In Python, there are two ways to pass arguments to a function:

Positional Arguments:

Positional arguments are the most common way of passing arguments to a function.
When you call a function and provide arguments without specifying the parameter names, the arguments are assigned to the parameters based on their positions.
The order of the arguments is important, and it must match the order of the parameters in the function definition.
Example:

```
    def greet(name, age):
    print("Hello,", name)
    print("Your age is", age)
```

```
greet("John", 25)
```

In this example, the function greet() takes two positional arguments:
name and age. When calling the function, the first argument "John"
is assigned to the name parameter, and the second argument 25 is
assigned to the age parameter. The order of the arguments must
match the order of the parameters in the function definition.

Keyword Arguments:

Keyword arguments allow you to specify the parameter names explicitly
when calling a function.
Instead of relying on the position of the arguments, you provide
the parameter names followed by a colon (:) and the corresponding values.
Keyword arguments provide more flexibility and clarity in function calls,
especially when dealing with functions that have multiple parameters.
Example:

```
def greet(name, age):
    print("Hello,", name)
    print("Your age is", age)

greet(age=25, name="John")
```

In this example, the function greet() is called with keyword arguments.
The parameter names age and name are specified explicitly, followed by
the corresponding values. The order of the arguments does not matter as
long as the parameter names are provided.

The main difference between positional arguments and keyword arguments
lies in how the arguments are assigned to the parameters: positional
arguments rely on the order of the arguments, while keyword arguments
rely on the explicit specification of parameter names. Keyword arguments
provide more flexibility and readability, especially when dealing with
functions that have a large number of parameters or optional parameters.

In [ ]:

3. What is the purpose of the return statement in a function?
Can a function have multiple returnstatements? Explain with an example.

A) The return statement in a function is used to specify the value
that the function should return when it is called. It allows the
function to pass a result back to the caller, which can be used for
further computation, assignment, or any other purpose.

The return statement serves two main purposes:

Terminating the Function:

When a return statement is encountered in a function, it immediately
terminates the execution of the function and returns control back to
the caller.
Any code or statements after the return statement will not be executed.
The return statement can be used to handle specific conditions or
criteria that determine when the function should stop executing.

Returning a Value:

The return statement allows a function to provide a value as
the result of its execution.
This returned value can be assigned to a variable, used in an
expression, or further processed by the caller.
The value can be of any data type, including integers, strings, lists,

```
dictionaries, or even other objects.
Example:

def add_numbers(a, b):
    result = a + b
    return result

sum_result = add_numbers(5, 3)
print("The sum is:", sum_result)
In this example, the add_numbers() function takes two parameters a and b.
Inside the function, the variables a and b are added, and the result
is stored in the result variable. The return statement is then used to
return this result back to the caller.

When the function is called with arguments 5 and 3, the return
value of 8 is assigned to the variable sum_result. This value
is then printed as the sum of the two numbers.

Yes, a function can have multiple return statements.
However, only one return statement will be executed during
the function's execution. Once a return statement is encountered,
the function immediately exits and returns the specified value.
Any subsequent return statements in the function will not be executed.
```

In [ ]:

```
4. What are lambda functions in Python? How are they different from regular
functions? Provide anexample where a lambda function can be useful.


A) In Python, a lambda function, also known as an anonymous function,
is a small, inline function that doesn't require a separate def statement.
It is created using the lambda keyword and can take any number of arguments
but can only have a single expression.

Here's the general syntax of a lambda function:


lambda arguments: expression
Lambda functions are different from regular functions in the following ways:


Anonymous: Lambda functions are anonymous, meaning they don't have a name.
    They are defined inline and are typically used for short,
    one-time operations without the need for a named function.

Single Expression: Lambda functions can only consist of a single
    expression. This expression is evaluated and returned as the
    result of the function.

Concise: Lambda functions provide a concise way to define simple
    functions without the need for a formal function definition using def.


Here's an example to demonstrate the usage of a lambda function:


# Regular function
def multiply(a, b):
    return a * b

result = multiply(3, 4)
print("Regular function result:", result)

# Lambda function
```

```
multiply_lambda = lambda a, b: a * b

lambda_result = multiply_lambda(3, 4)
print("Lambda function result:", lambda_result)
Output:


Regular function result: 12
Lambda function result: 12

In this example, we have a regular function called multiply() that
takes two arguments and returns their multiplication. We also have
a lambda function assigned to the variable multiply_lambda that performs
the same multiplication operation. Both functions are called with the
same arguments, and they return the same result.

Lambda functions are particularly useful in situations where a small,
one-time function is needed, such as in list comprehensions, filtering,
mapping, sorting, or as a parameter to other functions that expect a
function as an argument. They provide a concise and readable way to
define and use functions without the need for a separate function
definition.
```

In [ ]:

```
5. How does the concept of "scope" apply to functions in Python?
Explain the difference between localscope and global scope.

A )

In Python, the concept of "scope" refers to the visibility and accessibility
of variables within different parts of a program. Scopes determine which
variables can be accessed in a particular context and play a crucial role
in maintaining variable names' uniqueness and preventing naming conflicts.

When it comes to functions in Python, there are two primary scopes
to consider: local scope and global scope.

Local Scope:

Local scope refers to the scope within a function. It defines the visibility
of variables that are created inside the function.
Variables defined within a function are known as local variables and are
accessible only within that function.
Local variables have a limited lifespan and are created when the function
is called and destroyed when the function returns or completes execution.
Local variables take precedence over variables with the same name in outer
scopes (global or enclosing scopes).
Example:

    def my_function():
    x = 10  # Local variable
    print(x)

my_function()  # Output: 10
print(x)  # Raises NameError: name 'x' is not defined

In this example, the variable x is defined within the my_function()
function, making it a local variable. It is accessible only within the
function. When the function is called, the value of x is printed.
However, trying to access x outside the function scope raises a
NameError because x is not defined in the global scope.

Global Scope:
```

Global scope refers to the outermost scope of a program or module.
Variables defined outside any function or class, at the top level
of a module, have global scope.
Global variables are accessible from anywhere in the program, including
within functions.
Global variables have a longer lifespan compared to local variables
and persist throughout the program's execution.

```
x = 10  # Global variable

def my_function():
    print(x)

my_function()  # Output: 10
print(x)  # Output: 10
```

In this example, the variable x is defined outside any function,
making it a global variable. It can be accessed both within the
function my_function() and outside it.

It's important to note that modifying a global variable within a
function requires the use of the global keyword to indicate that
the variable being modified is the global one. Otherwise, a new
local variable with the same name will be created within the function.

Understanding and managing variable scopes is crucial for writing
maintainable and bug-free code. Scopes help in organizing and
encapsulating variables, preventing unintended side effects and
conflicts between variable names.

In [ ]:

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

A)In Python, the return statement in a function is used to specify
the value or values that the function should return when it is called.
While a function can only explicitly return a single value, there are
multiple ways to return multiple values from a function:

Returning a Tuple:

You can use a tuple to return multiple values from a function.
A tuple is an ordered collection of elements, and it can hold
multiple values.
By returning a tuple, you can effectively return multiple values
as a single object, which can be unpacked or accessed by the caller.
Example:

```
def calculate_statistics(numbers):
    total = sum(numbers)
    average = total / len(numbers)
    return total, average

numbers = [10, 20, 30, 40, 50]
total_result, average_result = calculate_statistics(numbers)
print("Total:", total_result)
print("Average:", average_result)
```

In this example, the calculate_statistics() function calculates the
total and average of a given list of numbers. Instead of returning two
separate values, it returns a tuple (total, average). The caller can
then unpack the returned tuple into two separate variables
(total_result and average_result) for further use.

```
Returning a List:

Similar to using a tuple, you can also use a list to return multiple
values from a function.
By returning a list, you can encapsulate multiple values and return t
hem as a single object.
Example:


def calculate_statistics(numbers):
    total = sum(numbers)
    average = total / len(numbers)
    return [total, average]

numbers = [10, 20, 30, 40, 50]
result = calculate_statistics(numbers)
print("Total:", result[0])
print("Average:", result[1])
In this example, the calculate_statistics() function
calculates the total and average of a given list of numbers
and returns them as a list [total, average]. The caller receives
the list and can access the individual values by their respective indices.

Using Namedtuples or Data Classes:

If you want to return multiple values with specific names or
attributes, you can use namedtuples or data classes.
Namedtuples and data classes provide a structured way to define
objects with named attributes, making it easier to work with and
understand the returned values.
Example using Namedtuple:


from collections import namedtuple

def get_person_details():
    Person = namedtuple("Person", ["name", "age", "city"])
    return Person("John Doe", 25, "New York")

person = get_person_details()
print("Name:", person.name)
print("Age:", person.age)
print("City:", person.city)
In this example, the get_person_details() function returns a
named tuple Person with three attributes: name, age, and city.
    The caller receives the named tuple and can access the values
    using the named attributes.

Returning multiple values from a function allows you to package
related information together and provide it to the caller in a
convenient manner. The choice of using tuples, lists, namedtuples,
or data classes depends on the specific requirements and the desired
structure of the returned values.
```

In [ ]:

```
7. What is the difference between the "pass by value" and
"pass by reference" concepts when itcomes to function arguments in Python?


A ) In Python, the concepts of "pass by value" and "pass by reference"
are often used to describe how function arguments are handled. However,
the reality is slightly different in Python, and it's more accurate to
say that Python uses a combination of both concepts. To understand this,
```

let's discuss the differences:

Pass by Value:

In pass by value, a copy of the value of a variable is passed to a function.
Any modifications made to the parameter inside the function do not affect the original variable.
The original variable remains unchanged.
Pass by Reference:

In pass by reference, the reference or memory address of a variable is passed to a function.
Any modifications made to the parameter inside the function affect the original variable.
The original variable may be modified.
In Python, the actual behavior depends on the type of the object being passed as an argument:

Immutable Objects (Pass by Value-like):

Immutable objects such as numbers, strings, and tuples are passed by value-like behavior.
When an immutable object is passed as an argument, a copy of the value is made and passed to the function.
Any modifications made to the parameter inside the function do not affect the original object.
The original object remains unchanged.
Mutable Objects (Pass by Reference-like):

Mutable objects such as lists, dictionaries, and user-defined objects are passed by reference-like behavior.
When a mutable object is passed as an argument, a reference to the object is passed to the function.
Any modifications made to the parameter inside the function affect the original object.
The original object may be modified.
Example:

```
def modify_immutable(value):
    value = 10  # Modifying the parameter
    print("Inside function - Value:", value)

def modify_mutable(data):
    data.append(4)  # Modifying the parameter
    print("Inside function - Data:", data)

# Immutable object (pass by value-like)
x = 5
modify_immutable(x)
print("Outside function - x:", x)

# Mutable object (pass by reference-like)
my_list = [1, 2, 3]
modify_mutable(my_list)
print("Outside function - my_list:", my_list)
```

Output:

```
Inside function - Value: 10
Outside function - x: 5
Inside function - Data: [1, 2, 3, 4]
Outside function - my_list: [1, 2, 3, 4]
```

In this example, the modify_immutable() function takes an immutable object value as an argument. When the function modifies the value parameter, it does not affect the original variable x outside the function. This behavior is similar to pass by value.

On the other hand, the modify_mutable() function takes a mutable object data as an argument. When the function appends an element to the data parameter, it affects the original list my_list outside the function. This behavior is similar to pass by reference.

So, while the terms "pass by value" and "pass by reference" are commonly used, it's important to understand that Python behaves differently based on the object's mutability. Immutable objects e xhibit pass by value-like behavior, and mutable objects exhibit pass by reference-like behavior.

In [ ]:
```
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

A)
```

In [1]:
```
import math

def perform_operations(value):
    # Logarithmic function (log x)
    logarithmic_result = math.log(value)

    # Exponential function (exp(x))
    exponential_result = math.exp(value)

    # Power function with base 2 (2^x)
    power_result = math.pow(2, value)

    # Square root
    square_root_result = math.sqrt(value)

    # Return the results as a dictionary
    results = {
        'logarithmic': logarithmic_result,
        'exponential': exponential_result,
        'power': power_result,
        'square_root': square_root_result
    }

    return results

# Test the function with different input values
value1 = 2
value2 = 3.5

results1 = perform_operations(value1)
results2 = perform_operations(value2)

# Print the results
print("For value =", value1)
print("Logarithmic result:", results1['logarithmic'])
print("Exponential result:", results1['exponential'])
print("Power result:", results1['power'])
print("Square root result:", results1['square_root'])
```

```
print("\nFor value =", value2)
print("Logarithmic result:", results2['logarithmic'])
print("Exponential result:", results2['exponential'])
print("Power result:", results2['power'])
print("Square root result:", results2['square_root'])
```

```
For value = 2
Logarithmic result: 0.6931471805599453
Exponential result: 7.38905609893065
Power result: 4.0
Square root result: 1.4142135623730951

For value = 3.5
Logarithmic result: 1.252762968495368
Exponential result: 33.11545195869231
Power result: 11.313708498984761
Square root result: 1.8708286933869707
```

In [ ]:
```
9. Create a function that takes a full name as an argument and
returns first nameand last name.
```

In [2]:
```
def extract_name(full_name):
    # Split the full name into a list of words
    name_parts = full_name.split()

    # Extract the first name (first element in the list)
    first_name = name_parts[0]

    # Extract the last name (last element in the list)
    last_name = name_parts[-1]

    # Return the first name and last name as a tuple
    return first_name, last_name

# Test the function
full_name1 = "John Doe"
full_name2 = "Alice Smith"

first_name1, last_name1 = extract_name(full_name1)
first_name2, last_name2 = extract_name(full_name2)

# Print the results
print("Full Name:", full_name1)
print("First Name:", first_name1)
print("Last Name:", last_name1)
print()

print("Full Name:", full_name2)
print("First Name:", first_name2)
print("Last Name:", last_name2)
```

```
Full Name: John Doe
First Name: John
Last Name: Doe

Full Name: Alice Smith
First Name: Alice
Last Name: Smith
```

In [ ]:
```
In this example, the extract_name() function takes a full name as
an argument. It splits the full name into a list of words using the
split() method, which separates the words based on whitespace.
The first name is extracted from the first element of the list
```

```
(name_parts[0]), and the last name is extracted from the last
element of the list (name_parts[-1]). Finally, the function returns
the first name and last name as a tuple.

The function is tested with two different full names
(full_name1 and full_name2), and the first name and last name
are printed accordingly.
```