1. **What is a lambda function in Python, and how does it differ from a regular function?**

In Python, a lambda function is an anonymous function that can be defined without a formal name. It is also known as a "lambda expression" or a "lambda form." Lambda functions are typically used for simple, one-line operations and are defined using the lambda keyword.

Here's the general syntax of a lambda function:

lambda arguments: expression

lambda: This is the keyword used to define a lambda function.

arguments: These are the input parameters of the function. You can have multiple arguments separated by commas, just like a regular function.

expression: This is the single expression or operation that the lambda function evaluates and returns as the result.

The main difference between a lambda function and a regular function defined using def is that lambda functions are anonymous and have no name associated with them. They are typically used when you need a function for a short, one-time use and don't want to define a named function separately.

Lambda functions are limited in their functionality compared to regular functions. They can only contain a single expression, and they cannot include statements or multiple lines of code. In contrast, regular functions can contain multiple statements, have a name, and can be reused throughout your program.

# Regular function

def multiply(x, y):

return x \* y

result1 = multiply(3, 4)

print(result1) # Output: 12

# Lambda function

multiply\_lambda = lambda x, y: x \* y

result2 = multiply\_lambda(3, 4)

print(result2) # Output: 12

1. **Can a lambda function in Python have multiple arguments? If yes, how can you define and use them?**

Yes, a lambda function in Python can have multiple arguments. You can define and use multiple arguments in a lambda function by separating them with commas, just like in a regular function.

Here's an example of a lambda function with multiple arguments:

multiply = lambda x, y: x \* y

result = multiply(3, 4)

print(result) # Output: 12

You can define lambda functions with any number of arguments by simply separating them with commas in the argument list. Here's an example with three arguments:

add = lambda a, b, c: a + b + c

result = add(1, 2, 3)

print(result) # Output: 6

In this case, the lambda function add takes three arguments, a, b, and c, and returns their sum.

Lambda functions allow you to define and use multiple arguments flexibly, enabling you to perform various operations and calculations within the lambda expression.

1. **How are lambda functions typically used in Python? Provide an example use case.**

In Python, a lambda function is an anonymous function that can be defined without a formal name. It is also known as a "lambda expression" or a "lambda form." Lambda functions are typically used for simple, one-line operations and are defined using the lambda keyword.

Here's the general syntax of a lambda function:

lambda arguments: expression

lambda: This is the keyword used to define a lambda function.

arguments: These are the input parameters of the function. You can have multiple arguments separated by commas, just like a regular function.

expression: This is the single expression or operation that the lambda function evaluates and returns as the result.

# Lambda function

multiply\_lambda = lambda x, y: x \* y

result2 = multiply\_lambda(3, 4)

print(result2) # Output: 12

1. **What are the advantages and limitations of lambda functions compared to regular functions in Python?**

Lambda functions in Python have several advantages and limitations compared to regular functions. Let's explore them:

Advantages of Lambda Functions:

Concise and Inline: Lambda functions allow you to define small, one-line functions directly in the expression where they are used. This eliminates the need for separate function definitions, making the code more compact and readable.

Anonymous: Lambda functions are anonymous, meaning they have no name associated with them. This is advantageous when you need a function for a specific task that doesn't require reuse or a separate definition.

Higher-Order Functions: Lambda functions are often used with higher-order functions like map(), filter(), and reduce(). These functions expect a function as an argument, and lambda functions provide a convenient way to define a simple function on the fly.

Limitations of Lambda Functions:

Single Expression: Lambda functions can only contain a single expression. They are not suitable for complex tasks that require multiple statements or extensive logic. Regular functions can handle such scenarios.

No Statements or Control Flow: Lambda functions cannot include statements like print or control flow structures like loops or conditionals. They are limited to evaluating a single expression and returning the result.

Reduced Readability: While lambda functions can make code more concise, excessive use of lambda functions or complex lambda expressions can reduce code readability. Regular functions with descriptive names can often be more understandable.

Limited Functionality: Lambda functions are designed for simple, immediate tasks. They are not intended to replace regular functions entirely. Regular functions provide greater flexibility, allow for code reuse, and can handle more complex operations.

1. **Are lambda functions in Python able to access variables defined outside of their own scope? Explain with an example.**

Yes, lambda functions in Python can access variables defined outside of their own scope. These variables are accessed through a concept called "lexical scoping" or "closure."

In lexical scoping, variables are resolved based on the environment in which the lambda function is defined. This means that lambda functions can access variables from the surrounding scope where they are defined, including global variables and variables from enclosing functions.

Ex: def outer\_function():

x = 10 # Variable defined in the outer function

inner\_lambda = lambda y: x + y

return inner\_lambda

closure = outer\_function()

result = closure(5)

print(result) # Output: 15

In this example, we have an outer function called outer\_function() that defines a variable x with a value of 10. Inside the outer function, we define a lambda function called inner\_lambda, which takes an argument y and adds it to the variable x. The lambda function is then returned from the outer function.

When we call outer\_function(), it returns the lambda function as a closure. We assign this closure to the variable closure. Later, we invoke the closure with an argument of 5, resulting in x (which is 10) being added to 5, giving us a result of 15.

The lambda function inner\_lambda accesses the variable x from its surrounding scope (the outer function's scope) even after the outer function has finished executing. This is possible due to the closure created when the lambda function is defined.

1. **Write a lambda function to calculate the square of a given number.**

square = lambda x: x\*\*2

In this lambda function, the argument x is raised to the power of 2 using the exponentiation operator (\*\*). The result is the square of the input number.

You can use this lambda function to calculate the square of any number by calling it with the desired number as an argument. Here's an example of how to use the square lambda function:

result = square(5)

print(result) # Output: 25

In this case, calling square(5) returns the square of 5, which is 25.

1. **Create a lambda function to find the maximum value in a list of integers.**

find\_max = lambda lst: max(lst)

In this lambda function, the max() function is used to find the maximum value in the input list lst. The lambda function takes a single argument, lst, representing the list of integers.

You can use this lambda function to find the maximum value in any list of integers by calling it with the desired list as an argument. Here's an example of how to use the find\_max lambda function:

numbers = [5, 9, 2, 1, 7, 3]

result = find\_max(numbers)

print(result) # Output: 9

1. **Implement a lambda function to filter out all the even numbers from a list of integers.**

filter\_even = lambda lst: list(filter(lambda x: x % 2 == 0, lst))

In this lambda function, the filter() function is used to create a new list that contains only the even numbers from the input list lst. The lambda function lambda x: x % 2 == 0 is used as the filtering criterion. It checks if each element x in the list is divisible by 2 (i.e., even) using the modulo operator %.

The lambda function filter\_even takes a single argument, lst, representing the list of integers.

You can use this lambda function to filter out even numbers from any list of integers by calling it with the desired list as an argument. Here's an example of how to use the filter\_even lambda function:

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

result = filter\_even(numbers)

print(result) # Output: [2, 4, 6, 8, 10]

1. **Write a lambda function to sort a list of strings in ascending order based on the length of each string.**

sort\_strings\_by\_length = lambda lst: sorted(lst, key=lambda x: len(x))

In this lambda function, the sorted() function is used to create a new list that contains the strings from the input list lst sorted in ascending order based on their lengths. The lambda function lambda x: len(x) is used as the sorting criterion. It returns the length of each string x, which is then used for comparison during the sorting process.

The lambda function sort\_strings\_by\_length takes a single argument, lst, representing the list of strings.

You can use this lambda function to sort a list of strings based on their lengths by calling it with the desired list as an argument. Here's an example of how to use the sort\_strings\_by\_length lambda function:

strings = ['apple', 'banana', 'orange', 'kiwi', 'pear']

result = sort\_strings\_by\_length(strings)

print(result) # Output: ['kiwi', 'pear', 'apple', 'banana', 'orange']

In this case, the sort\_strings\_by\_length(strings) call returns a new list that contains the strings from the strings list sorted in ascending order based on their lengths.

1. **Create a lambda function that takes two lists as input and returns a new list containing the common elements between the two lists.**

find\_common\_elements = lambda list1, list2: list(filter(lambda x: x in list2, list1))

In this lambda function, the filter() function is used to create a new list that contains the common elements between list1 and list2. The lambda function lambda x: x in list2 is used as the filtering criterion. It checks if each element x in list1 is present in list2.

The lambda function find\_common\_elements takes two arguments, list1 and list2, representing the two input lists.

You can use this lambda function to find the common elements between two lists by calling it with the desired lists as arguments. Here's an example of how to use the find\_common\_elements lambda function:

list1 = [1, 2, 3, 4, 5]

list2 = [4, 5, 6, 7, 8]

result = find\_common\_elements(list1, list2)

print(result) # Output: [4, 5]

1. **Write a recursive function to calculate the factorial of a given positive integer.**

def factorial(n):

if n == 0 or n == 1:

return 1

else:

return n \* factorial(n - 1)

In this recursive function, the base cases are defined for n = 0 and n = 1, where the factorial is known to be 1. For any other positive integer n, the factorial is calculated by recursively calling the factorial() function with n - 1 and multiplying it with n.

You can use this recursive function to calculate the factorial of a positive integer by calling it with the desired integer as an argument. Here's an example of how to use the factorial() function:

result = factorial(5)

print(result) # Output: 120

1. **Implement a recursive function to compute the nth Fibonacci number.**

def fibonacci(n):

if n <= 0:

raise ValueError("Input must be a positive integer.")

elif n == 1 or n == 2:

return 1

else:

return fibonacci(n - 1) + fibonacci(n - 2)

In this recursive function, the base cases are defined for n = 1 and n = 2, where the Fibonacci numbers are known to be 1. For any other positive integer n, the Fibonacci number is calculated by recursively calling the fibonacci() function with n - 1 and n - 2, and then adding the results together.

You can use this recursive function to compute the nth Fibonacci number by calling it with the desired value of n as an argument. Here's an example of how to use the fibonacci() function:

result = fibonacci(6)

print(result) # Output: 8

1. **Create a recursive function to find the sum of all the elements in a given list.**

def sum\_list\_elements(lst):

if len(lst) == 0:

return 0

else:

return lst[0] + sum\_list\_elements(lst[1:])

In this recursive function, the base case is defined for an empty list, where the sum is known to be 0. For a non-empty list, the function calculates the sum by adding the first element of the list to the sum of the remaining elements obtained by recursively calling the sum\_list\_elements() function with lst[1:], which represents the list excluding the first element.

You can use this recursive function to find the sum of all the elements in a list by calling it with the desired list as an argument. Here's an example of how to use the sum\_list\_elements() function:

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

result = sum\_list\_elements(my\_list)

print(result) # Output: 15

1. **Write a recursive function to determine whether a given string is a palindrome.**

def is\_palindrome(string):

if len(string) <= 1:

return True

elif string[0] != string[-1]:

return False

else:

return is\_palindrome(string[1:-1])

In this recursive function, the base cases are defined for strings of length 0 or 1, which are considered palindromes. If the first and last characters of the string are different, the function returns False, indicating that the string is not a palindrome. Otherwise, the function recursively calls itself with the substring obtained by excluding the first and last characters of the original string.

You can use this recursive function to determine whether a given string is a palindrome by calling it with the desired string as an argument. Here's an example of how to use the is\_palindrome() function:

result1 = is\_palindrome("racecar")

print(result1) # Output: True

result2 = is\_palindrome("hello")

print(result2) # Output: False

1. **Implement a recursive function to find the greatest common divisor (GCD) of two positive integers**

def gcd(a, b):

if b == 0:

return a

else:

return gcd(b, a % b)

In this recursive function, the base case is defined when b becomes 0, indicating that the GCD has been found. In this case, the function returns a, which is the GCD of the original input integers. For other cases, the function recursively calls itself with b as the new value of a and a % b as the new value of b. This step computes the GCD by repeatedly taking the remainder of division until the remainder becomes 0.

You can use this recursive function to find the GCD of two positive integers by calling it with the desired integers as arguments. Here's an example of how to use the gcd() function:

result = gcd(48, 60)

print(result) # Output: 12

In this case, calling gcd(48, 60) returns the GCD of 48 and 60, which is 12.