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

Lambda Function in Python:

* Lambda function is an anonymous, small, inline function.
* Created using the lambda keyword, followed by arguments and an expression.
* Typically used for simple operations within a single expression.

Difference from Regular Function:

* Lambda functions don't have a name.
* Limited to a single expression.
* Regular functions are defined using the def keyword and can be more versatile for complex operations.

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

Lambda Function with Multiple Arguments:

* Yes, a lambda function in Python can have multiple arguments.
* Define them within the lambda's argument list, separated by commas.
* Use the defined arguments in the expression to perform operations.

Example:

add = lambda x, y: x + y

result = add(3, 5) # Returns 8

In this example, the lambda function add takes two arguments, x and y, and returns their sum.

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

Typical Use of Lambda Functions:

Lambda functions are often used for small, one-time operations within function calls or higher-order functions like map, filter, and sorted.

Example Use Case:

# Using lambda with 'sorted' function to sort a list of tuples based on the second element

points = [(2, 5), (1, 8), (4, 3), (3, 6)]

points\_sorted = sorted(points, key=lambda x: x[1])

# Result: [(4, 3), (2, 5), (3, 6), (1, 8)]

In this example, a lambda function is used as the key argument in the sorted function to sort a list of tuples based on their second element. The lambda function specifies the operation to extract the sorting key.

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

Advantages of Lambda Functions:

* Concise syntax: Suitable for short operations without the need to define a named function.
* Useful for small tasks: Ideal for quick, one-time operations.
* Can be used inline: Inline use within other functions like map, filter, and sorted.

Limitations of Lambda Functions:

* Limited to single expressions: Cannot contain multiple statements or complex logic.
* Lack of name: Can make code less readable when used for complex operations.
* Reduced versatility: Not suitable for functions requiring documentation or reuse.

In summary, lambda functions excel at simplicity and conciseness for quick operations, while regular functions offer more flexibility and maintainability for more complex tasks.

**5. 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. This behavior is known as "closure."

Example:

x = 10

f = lambda y: x + y

result = f(5) # Returns 15

In this example, the lambda function f accesses the variable x from its enclosing scope. When the lambda is called with an argument of 5, it adds the value of x (which is 10) to the argument, resulting in 15. This demonstrates that lambda functions can capture variables from their containing scope, even after the scope has exited.

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

Here's a lambda function that calculates the square of a given number:

square = lambda x: x \*\* 2

You can use this lambda function by passing a number as an argument, like this:

result = square(4) # Returns 16

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

lambda function that finds the maximum value in a list of integers:

find\_max = lambda lst: max(lst)

You can use this lambda function by passing a list of integers as an argument, like this:

numbers = [5, 12, 8, 21, 15]

max\_value = find\_max(numbers) # Returns 21

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

lambda function that filters out all the even numbers from a list of integers:

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

You can use this lambda function by passing a list of integers as an argument, like this:

numbers = [2, 5, 8, 11, 14, 17]

even\_numbers = filter\_even(numbers) # Returns [2, 8, 14]

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

lambda function that sorts a list of strings in ascending order based on the length of each string:

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

You can use this lambda function by passing a list of strings as an argument, like this:

words = ["apple", "banana", "cherry", "date", "fig"]

sorted\_words = sort\_by\_length(words) # Returns ["date", "fig", "apple", "banana", "cherry"]

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

Lambda function that takes two lists as input and returns a new list containing the common elements between them:

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

You can use this lambda function by passing two lists as arguments, like this:

list1 = [2, 4, 6, 8, 10]

list2 = [6, 8, 12, 14]

common\_elements = find\_common(list1, list2) # Returns [6, 8]

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

recursive function that calculates the factorial of a given positive integer:

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

You can use this function by passing a positive integer as an argument, like this:

result = factorial(5) # Returns 120

In this example, factorial(5) returns 5 \* 4 \* 3 \* 2 \* 1, which is 120.

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

Certainly, here's a recursive function that computes the nth Fibonacci number:

def fibonacci(n):

if n <= 1:

return n

else:

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

You can use this function by passing a positive integer n as an argument, like this:

result = fibonacci(6) # Returns 8

In this example, fibonacci(6) returns the 6th Fibonacci number, which is 8 (0, 1, 1, 2, 3, 5, 8).

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

recursive function that finds the sum of all the elements in a given list:

def sum\_list\_recursive(lst):

if not lst:

return 0

else:

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

You can use this function by passing a list of numbers as an argument, like this:

numbers = [2, 4, 6, 8, 10]

total\_sum = sum\_list\_recursive(numbers) # Returns 30

In this example, sum\_list\_recursive(numbers) returns the sum of all the elements in the list, which is 30.

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

recursive function that determines whether a given string is a palindrome:

def is\_palindrome\_recursive(s):

if len(s) <= 1:

return True

else:

return s[0] == s[-1] and is\_palindrome\_recursive(s[1:-1])

You can use this function by passing a string as an argument, like this:

string1 = "radar"

string2 = "hello"

print(is\_palindrome\_recursive(string1)) # Returns True

print(is\_palindrome\_recursive(string2)) # Returns False

In this example, is\_palindrome\_recursive(string1) returns True because "radar" is a palindrome, while is\_palindrome\_recursive(string2) returns False because "hello" is not a palindrome.

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

recursive function that finds the greatest common divisor (GCD) of two positive integers using the Euclidean algorithm:

def gcd\_recursive(a, b):

if b == 0:

return a

else:

return gcd\_recursive(b, a % b)

You can use this function by passing two positive integers as arguments, like this:

a = 48

b = 18

result = gcd\_recursive(a, b) # Returns 6

In this example, gcd\_recursive(a, b) returns the GCD of 48 and 18, which is 6.