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

Ans1.) Lambda functions are useful for creating simple, short-lived functions on the fly without giving them a specific name. They are often used in situations where you need a quick function for a short period or as an argument for higher-order functions like map(), filter(), and reduce()

Differences between lambda functions and regular functions:

- a.) Syntax: Lambda functions are defined using the lambda keyword and have a more compact syntax compared to regular functions.
  - b.) Anonymous: Lambda functions are anonymous; they don't have a name. They are generally used for short and simple operations where defining a named function would be unnecessary.

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

Ans2.) Yes, a lambda function in Python can have multiple arguments. The syntax for defining a lambda function with multiple arguments is as follows:

```
lambda arg1, arg2, ..., argN: expression
```

You can specify any number of arguments separated by commas in the lambda function declaration. The expression following the colon will be evaluated and returned when the lambda function is called.

```
add = lambda x, y: x + y


result = add(3, 5)
    print(result)  # Output: 8
```

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

Ans3.) Lambda functions are typically used in Python for short and simple operations where defining a full-fledged named function would be unnecessary or less efficient. They are commonly used in situations where you need a quick function for a specific task or as arguments to higher-order functions like map(), filter(), and reduce(), which expect a function as an argument.

Here's an example use case to illustrate how lambda functions can be employed effectively:

Use Case: Sorting a list of tuples based on a specific element

Suppose you have a list of tuples, and you want to sort the list based on a specific element (e.g., the second element of each tuple). You can use the sorted() function with a lambda function as the key argument to achieve this without defining a separate named function

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

Ans 4.) Advantages of Lambda Functions:

- a.) Concise Syntax: Lambda functions allow you to define simple functions in a more compact and readable way, without the need to assign a specific name to the function.
- b.) Anonymous: Lambda functions are anonymous functions, which means you don't need to give them a name when defining them. This is useful for situations where you only need the function temporarily and don't want to clutter your code with unnecessary function names.
- c.) Simplified Operations: Lambda functions are particularly useful for quick and short-lived operations where creating a full-fledged regular function seems unnecessary and would result in additional overhead.
  - d.) Functional Programming: Lambda functions play a crucial role in functional programming in Python, as they can be easily used as arguments for higher-order functions like map(), filter(), and reduce(), which expect a function as input.

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

Ans5.)Yes, lambda functions in Python can access variables defined outside of their own scope. Lambda functions have access to variables from the containing scope where they are created. This behavior is similar to regular functions in Python.

Here's an example to demonstrate how lambda functions can access variables from the outer scope:

```
def outer_function(x):
   y = 10

   # Define a lambda function inside the outer function
   inner_lambda = lambda z: x + y + z

   return inner_lambda

# Call the outer function and get the lambda function as the result
result_lambda = outer_function(5)

# Call the lambda function with an argument
result = result_lambda(15)

    print(result)  # Output: 30
```

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

Ans6.) result = square (5)
    print(result) # Output: 25

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

Ans7.) numbers = [12, 45, 6, 78, 23, 56]
max_value = find_max(numbers)
    print(max_value) # Output: 78

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

Ans8.) Sure! You can use a lambda function with the filter () function to filter out all the even numbers from a list of integers. Here's the lambda function and an example of how to use it:

```
# Lambda function to filter even numbers
filter_even = lambda x: x % 2 == 0

# Example usage
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list (filter (filter_even, numbers))

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

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

    Ans9.) The lambda function lambda s: len(s) takes a string s as an argument and returns its length using the built-in len() function. The sorted() function uses this lambda function as the key argument to determine the sorting order based on the length of each string. As a result, the sorted_strings list contains the strings sorted in ascending order based on their lengths.

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

Ans10. #Lambda function to filter common elements between two lists

```
find_common_elements = lambda list1, list2: list(filter(lambda x: x in list2, list1))

# Example usage
list1 = [1, 2, 3, 4, 5]
list2 = [3, 4, 5, 6, 7]
common_elements = find_common_elements(list1, list2)

    print(common_elements)  #Output: [3, 4, 5]
```

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

Ans11.) Sure! A factorial of a positive integer n is the product of all positive integers from 1 to n. Recursive functions are functions that call themselves to solve a problem. Here's a recursive function to calculate the factorial of a given positive integer:

```
def factorial(n):
   # Base case: factorial of 0 and 1 is 1
   if n == 0 or n == 1:
      return 1
   # Recursive case: factorial of n is n times factorial of n-1
   else:
         return n * factorial
    (n - 1)
```

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

Ans12.) Sure! The Fibonacci sequence is a series of numbers in which each number (known as a Fibonacci number) is the sum of the two preceding ones, usually starting with 0 and 1. Here's a recursive function to compute the nth Fibonacci number:

```
def fibonacci(n):
   # Base case: For n = 0 or n = 1, return n as the Fibonacci number
   if n == 0:
      return 0
   elif n == 1:
      return 1
   # Recursive case: Compute the Fibonacci number using recursion
   else:
         return fibonacci(n - 1) + fibonacci(n - 2)
```

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

Ans13.) Sure! Here's a recursive function to find the sum of all the elements in a given list:

```
def recursive_sum(lst):
   # Base case: if the list is empty, return 0
   if not lst:
      return 0
   # Recursive case: return the first element + the sum of the rest of the elements
   else:
         return lst[0] + recursive_sum(lst[1:])
```

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

Ans14.) A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward (ignoring spaces, punctuation, and capitalization). To determine whether a given string is a palindrome, we can use a recursive approach. Here's a recursive function to check for palindromes:

```
def is_palindrome(s):
   # Base case: If the string is empty or contains only one character, it's a palindrome.
   if len(s) <= 1:
       return True

   # Recursive case: Compare the first and last characters of the string.
   # If they are equal, call the function recursively with the substring in between.
   # Otherwise, it's not a palindrome.
       return s[0] == s[-1] and is_palindrome(s[1:-1])
```

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

Ans15.) To find the Greatest Common Divisor (GCD) of two positive integers, we can use the Euclidean algorithm, which is well-suited for a recursive approach. The Euclidean algorithm states that the GCD of two numbers a and b is the same as the GCD of b and the remainder of a divided by b. Here's the recursive function to calculate the GCD:

```
def gcd(a, b):
    # Base case: If b is 0, the GCD is a.
    if b == 0:
        return a
    # Recursive case: Calculate the GCD using the remainder of a divided by b.
    return gcd(b, a % b)
```