1Q. What is a lambda function in Python, and how does it

differ from a regular function?

A) In Python, a lambda function, also known as an anonymous function, is a small, one-line function that doesn't require a name. It is created using the lambda keyword and is typically used for simple, short-lived operations. The basic syntax of a lambda function is as follows:

lambda arguments: expression

Here, arguments refers to the parameters of the function, and expression is thecomputation that the function performs. The result of the expression is automatically returned when the lambda function is called.

Lambda functions have a few key differences compared to regular functions:

Syntax: Lambda functions are defined using a more compact syntax compared to regular functions. They are typically written as a single line of code.

Namelessness: Lambda functions are anonymous, meaning they don't have a name assigned to them. They are usually used where they are defined or passed directly as arguments to other functions.

Single Expression: Lambda functions are limited to a single expression, which is evaluated and returned automatically. They are not designed to contain complex statements or multiple lines of code.

Function Objects: Lambda functions create function objects, just like regular functions.

However, lambda functions are usually used as throwaway functions for specific purposes,

whereas regular functions are defined with a name and can be reused throughout a program.

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

```
# Regular function
def square(x):
    return x ** 2

print(square(5))  # Output: 25

# Lambda function
square_lambda = lambda x: x ** 2

print(square_lambda(5))  # Output: 25
```

In this example, both the regular function square and the lambda function square_lambda perform the same operation, which is squaring a given number. However, the lambda function is defined in a more concise manner without explicitly naming it.

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

A) Yes, a lambda function in Python can have multiple arguments. We can define and use

multiple arguments in a lambda function by separating them with commas in the argument

list. Here's an example:

```
multiply = lambda x, y: x * y
print(multiply(5, 3))  # Output: 15
```

In this example, the lambda function multiply takes two arguments, x and y, and returns their product. When the lambda function is called with multiply(5, 3), it multiplies 5 and 3 together and returns the result, which is 15.You can add as many arguments as needed in a lambda function by separating them with commas. For example, a lambda function with three arguments could be defined like this:

```
sum_three_numbers = lambda x, y, z: x + y + z

print(sum_three_numbers(2, 4, 6))  # Output: 12
```

In this case, the lambda function sum_three_numbers takes three arguments, x, y, and z, and returns their sum.lambda functions are designed for simple and concise operations, so it's generally recommended to use regular functions for more complex tasks that involve multiple statements or require extensive functionality.

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

A) Lambda functions in Python are often used in situations where a small,

one-time function is required, particularly as arguments to higher-order functions or for simple transformations. Here are a few example use cases where lambda functions are commonly employed:

Sorting: Lambda functions can be used as the key parameter in sorting operations to define custom sorting criteria. For instance, if you have a list of tuples representing people's names and ages, We can sort the list based on the age using a lambda function:

```
    people = [("Alice", 25), ("Bob", 20), ("Charlie", 30)]

sorted_people = sorted(people, key=lambda x: x[1])

print(sorted_people)

# Output: [("Bob", 20), ("Alice", 25), ("Charlie", 30)]
```

Here, the lambda function lambda x: x[1] extracts the second element (age)

from each tuple for sorting purposes.

Filtering: Lambda functions can be used with filtering functions like filter() to selectively include or exclude elements from a collection. For example, We can use a lambda function to filter out even numbers from a list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filtered_numbers = list(filter(lambda x: x % 2 != 0, numbers))
print(filtered_numbers)
# Output: [1, 3, 5, 7, 9]
```

The lambda function lambda x: x % 2 != 0 filters out numbers that are not divisible by 2, resulting in a list of odd numbers.

Mapping: Lambda functions can be used with mapping functions like map() to apply a transformation to each element of a collection. For instance, We can use a lambda function to square each number in a list:

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

The lambda function lambda x: x**2 squares each number in the list, resulting in a new list of squared numbers.

These are just a few examples of how lambda functions can be used in Python.

They provide a concise and convenient way to define simple functions on the fly, without the need for a formal function definition.

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

A) Lambda functions in Python offer several advantages over regular functions in certain scenarios, but they also have some limitations. Here are the advantages and limitations of lambda functions compared to regular functions:

Advantages of Lambda Functions:

Concise Syntax: Lambda functions allow you to express simple functions in a

compact, one-line syntax. This can make the code more readable and reduce the amount of boilerplate code required.

Anonymous Functions: Lambda functions are anonymous, meaning they don't require a specific name. This is useful when you need to create a small function on the fly for a specific purpose, such as passing it as an argument to a higher-order function.

Function Literals: Lambda functions can be treated as values and assigned to variables or used as elements in data structures like lists or dictionaries. This flexibility enables more dynamic and functional programming styles.

Higher-Order Functions: Lambda functions are often used as arguments to higher-order functions like map(), filter(), and sort(). They allow you to define simple transformations or custom sorting/filtering criteria without the need for a separate named function.

Limitations of Lambda Functions:

Single Expression: Lambda functions are limited to a single expression,

which means they cannot contain complex statements or multiple lines of code. This makes them unsuitable for more complex functions that require additional control flow or multiple statements.

Limited Readability: While lambda functions can make code more concise, they can also reduce readability if they become too complex or if the purpose of thefunction is not immediately clear. In such cases, it's generally better to usea regular function with a meaningful name.

Lack of Documentation: Lambda functions don't support docstrings, which are used to provide documentation and explanations for regular functions. This can make lambda functions less self-explanatory and harder to understand for other developers.

Reduced Reusability: Lambda functions are primarily designed for one-time or specific use cases. They lack the reusability of regular functions, which can be defined once and called multiple times throughout a program.

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

A)

Yes, lambda functions in Python can access variables defined outside of their own scope. Lambda functions have access to variables in the enclosing scope, which is known as lexical scoping or closure. This allows lambda functions to reference and use variables from the surrounding code block where they are defined.

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

```
def multiplier(n):
   return lambda x: x * n
```

```
double = multiplier(2)
triple = multiplier(3)

print(double(5))  # Output: 10
print(triple(5))  # Output: 15
```

In this example, the multiplier function returns a lambda function
that multiplies its argument x with the value of n. When multiplier(2)
is called, it returns a lambda function that multiplies a given number by 2.
Similarly, multiplier(3) returns a lambda function that multiplies a given number by 3.

The lambda functions double and triple are assigned the returned lambda
functions from multiplier(2) and multiplier(3) respectively. These lambda
functions can access the n variable from the enclosing scope of multiplier.
When double(5) is called, it invokes the lambda function with x=5 and uses
the n value from the enclosing scope, which is 2. Therefore, the output is
10 (5 * 2). Similarly, triple(5) multiplies 5 by the n value of 3, resulting
in an output of 15.
This example demonstrates how lambda functions can capture and access variables from their enclosing scope, allowing for more flexible and dynamic behavior.

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

```
square = lambda x: x**2
```

In this lambda function, x is the parameter representing the input number, and x**2 is the expression that computes the square of x. You can use this lambda function to calculate the square of any number by calling it with the desired input value.

Here's an example of using the lambda function to calculate the square of a number:

```
square = lambda x: x**2


result = square(5)
print(result)  # Output: 25
```

In this example, the lambda function square is called with an argument of 5, which computes the square of 5 and assigns the result to the variable result. The output is 25, which is the square of 5.

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

A) find_max = lambda lst: max(lst)
In this lambda function, lst is the parameter representing
the list of integers, and max(lst) is the expression that calculates
the maximum value from the list using the built-in max() function.

Here's an example of using the lambda function to find the maximum value in a list:

```
find_max = lambda lst: max(lst)

numbers = [5, 8, 2, 10, 3]
result = find_max(numbers)
print(result)  # Output: 10
```

In this example, the lambda function find_max is called with the numbers list as the argument. The lambda function internally uses the max() function to determine the maximum value from the list. The result is 10, which is the maximum value in the list.

8Q. 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, lst is the parameter representing the list of integers. The lambda function uses the filter() function along with a lambda function as the filtering condition. The lambda function lambda x: x % 2 == 0 checks if a number x is divisible by 2 (i.e., an even number).

Here's an example of using the lambda function to filter out even numbers from a list:

```
filter_even = lambda lst: list(filter(lambda x: x % 2 == 0, lst))

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
result = filter_even(numbers)
print(result)  # Output: [2, 4, 6, 8, 10]
```

In this example, the lambda function filter_even is called with the numbers list as the argument. The lambda function internally uses the filter() function with the lambda function lambda x: x % 2 == 0 to filter out the even numbers from the list. The result is [2, 4, 6, 8, 10], which are the even numbers present in the list.

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

A)

```
sort_by_length = lambda lst: sorted(lst, key=lambda x: len(x))
```

In this lambda function, lst is the parameter representing the list of strings. The lambda function uses the sorted() function along with a lambda function as the key parameter. The lambda function lambda x: len(x) retrieves the length of each string x and uses it as the sorting criterion.

Here's an example of using the lambda function to sort a list of strings based on their length:

```
sort_by_length = lambda lst: sorted(lst, key=lambda x: len(x))
```

```
strings = ["apple", "banana", "cherry", "date", "elderberry"]
result = sort_by_length(strings)
print(result)  # Output: ['date', 'apple', 'banana', 'cherry', 'elderberry']
```

In this example, the lambda function sort_by_length is called with the strings list as the argument. The lambda function internally uses the sorted() function with the lambda function lambda x: len(x) to sort the strings in ascending order based on their length.
The result is ['date', 'apple', 'banana', 'cherry', 'elderberry'], which is the sorted list of strings.

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

A)

```
find_common_elements = lambda lst1, lst2: list(filter(lambda x: x in lst2, lst1))
```

In this lambda function, lst1 and lst2 are the parameters representing the two lists. The lambda function uses the filter() function along with a lambda function as the filtering condition. The lambda function lambda x: x in lst2 checks if an element x from lst1 is present in lst2,filtering out the elements that are not common.

Here's an example of using the lambda function to find the common elements between two lists:

```
find_common_elements = lambda lst1, lst2: list(filter(lambda x: x in lst2, lst1))

list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
result = find_common_elements(list1, list2)
print(result)  # Output: [4, 5]
```

In this example, the lambda function find_common_elements is called with list1 and list2 as the arguments. The lambda function internally uses the filter() function with the lambda function lambda x: x in lst2 to filter out the common elements between the two lists. The result is [4, 5],
which are the common elements present in both lists.

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

A)

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

In this recursive function, the base case is defined when
n is equal to 0 or 1. In these cases, the factorial is 1.
Otherwise, the function calls itself with n decremented by 1,
and multiplies the result with n.

Here's an example of using the factorial function to calculate the factorial of a positive integer:

```
    result = factorial(5)
print(result)  # Output: 120
```

In this example, the factorial function is called with an argument of 5. The function recursively calculates the factorial as 5 * 4 * 3 * 2 * 1,

which equals 120. The result is printed as 120.

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

A)

```
def fibonacci(n):
    if n <= 0:
        raise ValueError("Fibonacci sequence is only defined for positive integers.")
    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 when n is equal to 1 or 2. In these cases, the Fibonacci number is 1. Otherwise, the function calls itself recursively with n decremented by 1 and n decremented by 2, and returns the sum of the two previous Fibonacci numbers.

Here's an example of using the Fibonacci function to compute the nth Fibonacci number:

```
result = fibonacci(6)
print(result)  # Output: 8
```

In this example, the Fibonacci function is called with an argument of 6. The function recursively calculates the 6th Fibonacci number, which is the sum of the 4th and 5th Fibonacci numbers: 2 + 3 = 5. The result is printed as 8.

Please note that the Fibonacci sequence starts with 1, 1, 2, 3, 5, 8, 13, and so on. So, the index starts from 1, and the function will raise a ValueError if a non-positive value is passed as an argument.

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

A)

```
def sum_list_elements(lst):
    if not lst:
        return 0
    else:
        return lst[0] + sum_list_elements(lst[1:])
```

In this recursive function, the base case is defined when the list lst is empty. In this case, the sum is 0. Otherwise, the function recursively adds the first element of the list (lst[0]) with the sum of the remaining elements (lst[1:]) by calling itself.

Here's an example of using the sum_list_elements function to find the sum of elements in a list:

```
my_list = [1, 2, 3, 4, 5]
result = sum_list_elements(my_list)
print(result)  # Output: 15
```

In this example, the sum_list_elements function is called with the my_list list as the argument. The function recursively calculates the sum of all the elements in the list as 1 + 2 + 3 + 4 + 5, which equals 15. The result is printed as 15.

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

A)

```
def is_palindrome(string):
    if len(string) <= 1:
        return True
    elif string[0] == string[-1]:
        return is_palindrome(string[1:-1])
    else:
        return False
```

In this recursive function, the base cases are defined when the length of the string is less than or equal to 1. In these cases, the string is considered a palindrome. The function checks if the first and last characters of the string are equal. If they are, it calls itself recursively with the string excluding the first and last characters (string[1:-1]). If the first and last characters are not equal, the string is not a palindrome.

Here's an example of using the is_palindrome function to determine whether a given string is a palindrome:

```
result1 = is_palindrome("radar")
print(result1)  # Output: True

result2 = is_palindrome("hello")
print(result2)  # Output: False
```

In the first example, the is_palindrome function is called with the string "radar". The function recursively checks if the first and last characters ("r" and "r") are equal, and then calls itself with the remaining substring "ada". It continues this process until the base case is reached, and since all characters match, the result is True.

In the second example, the is_palindrome function is called with the string "hello". The function recursively checks if the first and last characters ("h" and "o") are equal, which they are not. Therefore, the result is False.

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

A)

```
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 the second integer b becomes zero. In this case, the GCD is the value of the first integer a. Otherwise, the function recursively calls itself with the arguments b and a % b, which computes the remainder when a is divided by b.

Here's an example of using the gcd function to find the GCD of two positive integers:

```
    result = gcd(48, 18)
print(result)  # Output: 6
```

In this example, the gcd function is called with the integers 48 and 18. The function recursively computes the GCD using the Euclidean algorithm. The process continues until b becomes zero. The GCD of 48 and 18 is 6, which is the result printed.