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

Ans: In Python, a lambda function, also known as an anonymous function, is a small and anonymous function that doesn't have a name. It is defined using the lambda keyword and can take any number of arguments but can only have a single expression. Lambda functions are commonly used when you need a simple function for a short period and don't want to define a separate named function.

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

them?

Ans: Yes, a lambda function in Python can have multiple arguments. Lambda functions are also known as anonymous functions because they don't require a formal def statement to define them. Instead, you can use the lambda keyword followed by the arguments, a colon, and the expression to be evaluated.

Here's the general syntax of a lambda function with multiple arguments:

# Define a lambda function with two arguments

multiply = lambda x, y: x \* y

# Use the lambda function

result = multiply(3, 4)

print(result) # Output: 12

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

Ans: In Python, lambda functions, also known as anonymous functions, are small, inline functions that are defined without a name. They are typically used in situations where a small, one-time function is required and it doesn't make sense to define a full-fledged named function.

Here's an example to illustrate a use case for lambda functions in Python:

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

sorted\_numbers = sorted(numbers, key=lambda x: x\*\*2)

print(sorted\_numbers)

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

Python?

Ans: Lambda functions, also known as anonymous functions, are a feature in Python that allows you to create small, one-line functions without explicitly defining them using the def keyword. Here are the advantages and limitations of lambda functions compared to regular functions in Python:

Advantages of Lambda Functions:

Concise Syntax: Lambda functions have a compact syntax that allows you to define functions in a single line of code. This can be particularly useful for simple and short functions, eliminating the need for writing a separate function definition.

Limitations of Lambda Functions:

Single Expression Limitation: Lambda functions are limited to a single expression. This means you cannot include statements or multiple lines of code within a lambda function. They are best suited for simple computations or operations that can be expressed within a single line.

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

Explain with an example.

Ans:

Yes, lambda functions in Python can access variables defined outside of their own scope. This concept is known as "lexical scoping" or "closure." When a lambda function is defined, it retains access to variables in its surrounding scope, including global variables and variables from enclosing functions.

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

def outer\_function():

x = 10

# Define a lambda function that uses the variable x from the outer\_function's scope

inner\_lambda = lambda y: x + y

return inner\_lambda

# Create an instance of the inner\_lambda function

my\_lambda = outer\_function()

# Call the lambda function and pass an argument

result = my\_lambda(5)

print(result) # Output: 15

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

Ans: Certainly! Here's a Lambda function written in Python that calculates the square of a given number:

import json

def lambda\_handler(event, context):

# Parse the input from the event

input\_number = int(event['number'])

# Calculate the square

result = input\_number \*\* 2

# Prepare the response

response = {

'input': input\_number,

'result': result

}

# Return the response as JSON

return {

'statusCode': 200,

'body': json.dumps(response)

}

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

Ans: Certainly! Here's an example of a Python lambda function that finds the maximum value in a list of integers:

max\_value = lambda lst: max(lst)

numbers = [5, 10, 3, 8, 2]

result = max\_value(numbers)

print(result) # Output: 10

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

Ans:

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]

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

string.

Ans:

strings = ["apple", "banana", "cherry", "date", "elderberry"]

sorted\_strings = sorted(strings, key=lambda x: len(x))

print(sorted\_strings)

#Output:

['date', 'apple', 'banana', 'cherry', 'elderberry']

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

common elements between the two lists.

Ans:

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

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

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

result = common\_elements(list1, list2)

print(result)

#Output:

[4, 5]

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

Ans:

def factorial(n):

if n == 0 or n == 1:

return 1

else:

return n \* factorial(n - 1)

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

Ans:

def fibonacci(n):

if n <= 0:

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

# Base cases: Fibonacci numbers for n = 1 and n = 2 are 1

if n == 1 or n == 2:

return 1

# Recursive case: Fibonacci number is the sum of the previous two Fibonacci numbers

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

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

Ans:

def recursive\_sum(lst):

if len(lst) == 0:

return 0

else:

return lst[0] + recursive\_sum(lst[1:])

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

Ans:

def is\_palindrome(string):

if len(string) <= 1:

return True

else:

if string[0] == string[-1]:

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

else:

return False

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

Ans:

def gcd\_recursive(a, b):

if b == 0:

return a

else:

return gcd\_recursive(b, a % b)

#To use the function, you can simply call it with two positive integers as arguments:

num1 = 48

num2 = 18

result = gcd\_recursive(num1, num2)

print("GCD of", num1, "and", num2, "is:", result)

#Output

GCD of 48 and 18 is: 6