Functions Assignment

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

- lambda function, also known as an anonymous function, is a way to create small, one-line functions without explicitly defining them using the def keyword.
- Syntax: Lambda functions are defined in a single line using the **lambda** keyword, whereas regular functions require the **def** keyword and a block of code.
- Function Name: Lambda functions are anonymous, meaning they don't have a specific name. Instead, they are typically assigned to variables.
- Size and Simplicity: Lambda functions are usually concise and designed for simple operations. They are often used when a function is required as an argument to another function or when a small function is needed temporarily.
- Return Value: Lambda functions automatically return the result of the expression without needing an explicit **return** statement.

2. 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. Lambda functions are also known as anonymous functions, and they can take any number of arguments, separated by commas, just like a regular function. Here's an example of how you can define and use a lambda function with multiple arguments:

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

- Lambda functions in Python are commonly used in situations where you need a small, one-time function without the need for a formal function definition.

Ex:

```
names = ["Alice", "Bob", "Charlie", "David", "Eve"]

sorted_names = sorted(names, key=lambda x: len(x), reverse=True)

print(sorted_names)
```

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

Advantages of Lambda Functions:

Concise syntax: Lambda functions are defined in a single line of code, making them ideal for simple and short tasks. They eliminate the need for writing a full function definition with the def keyword.

Readability: Lambda functions can make the code more readable when used appropriately. For instance, they can be helpful when passing a function as an argument to another function, such as in sorting or filtering operations

Limitations of Lambda Functions:

Limited functionality: Lambda functions are restricted to a single expression. They cannot contain multiple statements or complex logic. This limitation makes them unsuitable for tasks that require control flow statements like loops or conditional statements.

Lack of documentation: As lambda functions are anonymous, they lack a formal name and docstring, which makes them less self-explanatory.

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

- lambda functions can access variables defined outside of their own scope. This is possible because lambda functions have access to the variables in the enclosing scope where they are defined.

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

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

number = 5

result = square(number)

print(result)
```

Output: 25

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

```
max_value = lambda lst: max(lst)
numbers = [5, 2, 9, 1, 7]
result = max_value(numbers)
print(result)  # Output: 9
```

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

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

filtered_numbers = list(filter(lambda x: x % 2 != 0, numbers))

print(filtered_numbers)

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

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

sorted_strings = sorted(strings, key=lambda x: len(x))

print(sorted_strings)

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

- common_elements = lambda list1, list2: [element for element in list1 if element in list2]

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

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

result = common_elements(list1, list2)

print(result)

11. 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)
```

12. 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)
n = 10
result = fibonacci(n)
print(f"The {n}th Fibonacci number is: {result}")
```

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

```
def recursive_sum(lst):
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + recursive_sum(lst[1:])
my_list = [1, 2, 3, 4, 5]
result = recursive_sum(my_list)
print(result)
```

Output: 15

14. 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])
my_string = "racecar"
result = is_palindrome(my_string)
print(result)
```

Output: True

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

```
def gcd_recursive(a, b):
    if b == 0:
        return a
    else:
        return gcd_recursive(b, a % b)
a = 36
b = 48
result = gcd_recursive(a, b)
print(f"The GCD of {a} and {b} is {result}")
```