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

Ans:Lambda function also called anonymous function.It is used where we do not define repeative function, we used this for simple question for short period of time and donot want to define a separate named function.

Lambda vs regular function:
Syntax: Lambda functions have a more concise syntax compared to regular functions. They are typically written in a single line and don't require a return statement. In Python, lambda functions are defined using the lambda keyword, followed by the parameter list and the expression. Regular functions, on the other hand, are defined with the def keyword, followed by the function name, parameter list, and an indented block of code.

Name: Lambda functions are anonymous, meaning they don't have a specific name. They are usually used when you need a small, one-time function and don't want to define a named function. Regular functions, on the other hand, have a name that can be used to call them from other parts of the code.

Usage: Lambda functions are often used as arguments to higher-order functions, such as map(), filter(), or sort(), where you need a simple function for a specific operation. Regular functions, on the other hand, are more suitable when you want to define reusable blocks of code that can be called from multiple places in your program.

Scope: Lambda functions are limited to the expression they contain and have access to variables in the enclosing scope. They can't contain statements or multiple expressions. Regular functions, on the other hand, can contain multiple statements, have their own scope, and can include more complex logic.

Readability: Lambda functions are useful for writing short, concise code when the logic is simple and easy to understand in a single line. However, using them for complex operations can make the code harder to read and maintain. Regular functions provide a clearer structure and allow for more detailed documentation and comments, making the code more readable and understandable.

In summary, lambda functions are a compact way to define small, anonymous functions for simple operations, whereas regular functions are used for more complex logic and can be reused throughout the codebase. The choice between lambda functions and regular functions depends on the specific requirements and context of your program.

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. The syntax for defining a lambda function with multiple arguments is as follows:

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

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

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

Ans: Lambda functions, also known as anonymous functions, are a feature in Python that allow you to create small, one-line functions without explicitly defining them using the def keyword. They are typically used in situations where you need a simple function for a short period of time or as an argument to a higher-order function that expects a function as input.

Eg:

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

output:

```
[2, 4, 6, 8, 10]
```

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

Ans: Advantages of Lambda Functions:

Concise Syntax: Lambda functions allow you to define simple, one-line functions without the need for the **def** keyword. This makes them handy for quick, short-lived functions that don't require a full function definition.

Readability: Lambda functions are often used in situations where a small, inline function is more readable than defining a separate named function. They can make the code more compact and focused.

Convenience: Lambda functions are typically used in functional programming paradigms, such as with higher-order functions like **map()**, **filter()**, and **reduce()**. Using lambda functions in these cases can be more convenient and expressive than writing a separate named function.

Immediate Execution: Lambda functions are often used when you need to define a function and immediately execute it, without assigning it a name. This can be useful in situations where you only need the function once and don't want to clutter your code with unnecessary function definitions.

Limitations of Lambda Functions:

Limited Functionality: Lambda functions are restricted to a single expression, which means they can only contain a single line of code. They cannot contain multiple statements or complex logic. If you need to write more complex functions, regular functions with multiple lines of code are more appropriate.

No Documentation or Type Hints: Since lambda functions are anonymous, they lack the ability to provide documentation or type hints. This can make them less self-explanatory and harder to understand, especially for other developers who may work on the code.

Reduced Reusability: Lambda functions are primarily used for simple, one-off operations. They are not intended for reuse or as part of a larger code structure. If you have a function that you need to reuse in multiple places or want to incorporate in a broader codebase, a regular named function is a better choice.

Debugging Challenges: Lambda functions can be harder to debug because they don't have a name or a dedicated traceback associated with them. When an error occurs within a lambda function, it can be more challenging to track down the source of the issue.

In summary, lambda functions offer conciseness and convenience for small, one-off operations, particularly in functional programming contexts. However, they have limitations in terms of functionality, reusability, documentation, and debugging, making regular functions more suitable for complex and reusable code.

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

Ans: In Python, lambda functions, also known as anonymous functions, have access to variables defined outside of their own scope. This is because lambda functions can capture variables from the surrounding scope in a concept called "lexical scoping" or "closure."

```
def outer_function():
    message = 'Hello'

    # Define a lambda function inside the outer function
    inner_lambda = lambda name: message + ', ' + name

    return inner_lambda

# Call the outer function to get the lambda function
my_lambda = outer_function()

# Call the lambda function
result = my_lambda('John')

print(result)
```

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

Ans:
```
square=lambda x:x**2
n1=int(input("enter the number:"))
print(square(n1))
```

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

Ans:
```
my_list=[12,23,45,12,34]
 Max_value=lambda lst:max(lst)
result=max_value(my_list)
print(result)
```

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

Ans:
```
my_list=[13,15,16,57,59,30,46,78,24]
even_no=lambda lst: [x for x in lst if x%2==0]
result=even_no(my_list)
print(result)
```

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)
```

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(set(list1) & set(list2))    #set remove the duplicate number and & give the common elements
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:
```
n1=int(input("enter the number:"))
def factorial(n):
    if n==0:
        return 1
    else:
        return n*factorial(n-1)
result= factorial(n1)
print(result)
```

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

Ans:
```
def fibronacci (n):
        If n<=1
         Return n
        Else:
         Return fibronacci(n-1)+fibronacci(n-2)
         Print(fibronacci)
```

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:])
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.

Ans:
```
def is_palindrome(num):
    # Convert the number to a string
    num_str = str(num)

    # Reverse the string
    reversed_str = num_str[::-1]

    # Compare the original string with the reversed string
    if num_str == reversed_str:
        return True
    else:
        return False
```

```
# Example usage
number = 121
if is_palindrome(number):
    print(f"{number} is a palindrome number.")
else:
    print(f"{number} is not a palindrome number.")
```

15. Implement a recursive function to find the greatest common divisor (GCD) of two positive integers.
C
Ans:
```
def gcd_recursive(a, b):
    if b == 0:
        return a
    else:
        return gcd_recursive(b, a % b)

num1 = 84
num2 = 18
result = gcd_recursive(num1, num2)
print("The GCD of", num1, "and", num2, "is:", result)
```