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

- A lambda function is a small anonymous function that can be used as an argument to another function. It is defined using the lambda keyword, followed by the arguments in parentheses, and then the expression to be evaluated. For example, the following lambda function adds 1 to the number passed to it:

```
lambda x: x + 1
```

- Lambda functions are often used when you need a function for a short period of time, or when you need to pass a function as an argument to another function. They are also useful for creating functions that take a variable number of arguments. Lambda functions differ from regular functions in several ways.
- First, lambda functions do not have a name. Second, lambda functions can only have one expression. Third, lambda functions are always defined within the scope of another function. Despite these differences, lambda functions can be used for the same tasks as regular functions. They can be used to perform calculations, to filter data, and to map data.

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. To define a lambda function with multiple arguments, simply list the arguments in parentheses, separated by commas. For example, the following lambda function takes two arguments, x and y, and returns their sum:

- lambda x, y: x + y

- You can use a lambda function with multiple arguments in the same way that you would use a regular function with multiple arguments. For example, the following code uses a lambda function to calculate the sum of two numbers:

```
x = 10
y = 20

sum = lambda x, y: x + y

print(sum(x, y))
```

- The output of this code will be 30.

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

- Lambda functions are typically used in Python when you need a function for a short period of time, or when you need to pass a function as an argument to another function. They are also useful for creating functions that take a variable number of arguments. One common use case for lambda functions is to filter data. For example, the following code uses a lambda function to filter a list of numbers:

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

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

  print(even_numbers)

- The output of this code will be [2, 4].

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

- Lambda functions have several advantages over regular functions in Python. First, they are more concise, which can make your code more readable. Second, they can be used as arguments to other functions, which can make your code more modular. Third, they can be created and used within a single expression, which can make your code more concise.
- However, lambda functions also have some limitations. First, they cannot have a name, which can make it difficult to track their usage. Second, they can only have one expression, which can limit their functionality. Third, they are not as flexible as regular functions, and they cannot be used in all situations.
- Overall, lambda functions are a powerful tool that can be used to simplify your code and make it more readable. However, it is important to be aware of their limitations before using them.

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

Explain with an example.

- yes, lambda functions in Python are able to access variables defined outside of their own scope. This is because lambda functions are closures, which means that they capture the environment in which they are defined. This means that they can access any variables that are defined in the enclosing scope, even if they are not explicitly passed as arguments to the lambda function. For example, the following code defines a lambda function that takes a number as an argument and returns the square of that number. The lambda function also accesses the variable `x`, which is defined in the enclosing scope.

```
x = 5

square = lambda y: y * y

print(square(x))
```

- The output of this code is 25. It is important to note that lambda functions can only access variables that are defined in the enclosing scope. They cannot access variables that are defined in a nested scope. For example, the following code will not work:

```
x = 5

def outer_function():
y = 10

square = lambda z: z * z

print(square(y))

outer_function()
```

- This code will generate an error because the variable `y` is not defined in the scope of the lambda function. Lambda functions are a powerful tool that can be used to simplify code and make it more readable. However, it is important to understand how lambda functions work in order to use them effectively.

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

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

print(square(5))
```

o/p: 25

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

```
max_num = lambda x,y: if x>y else y

print(max_num(10,20))
```

o/p : 20

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

```
even_odd = lambda x: 'even' if x%2==0 else'odd'
print(even_odd(22))
```

o/p : even

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

```
product = [{"n":"laptop","price":450000},
        {"n":"headphone","price":5000},
        {"n":"phone","price":25000}]
sorted_products = sorted(product,key= lambda x:x['n']) #by default it is in increasing order
for i  in sorted_products:
   print(i)
```

o/p :

```
 'n': 'headphone', 'price': 5000}
{'n': 'laptop', 'price': 450000}
{'n': 'phone', 'price': 25000}
```

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

```
list1 = list(input("enter list 1 :"))
list2 = list(input("enter list 2 :"))
common_elements= lambda list1, list2:[element for element in list1 if element in list2]
print(common_elements(list1,list2))
```

o/p:

```
enter list 1 :abc
enter list 2 :abd
['a', 'b']
```

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

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
result = factorial(5)
print(result)
```

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

```
n = int(input("enter n :"))
def print_fibonnaci(n):
    a, b = 0, 1
    for i in range(n):
        print(a)
        a, b = b, a + b
print(print_fibonnaci(n))
```

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

```
list1 = list(input("enter list"))
def sum_list(list1):
  if len(list1) == 0:
    return 0
  else:
    return list1[0] + sum_list(list1[1:])
result = sum_list(list1)
print(result)
```

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

```
def is_palindrome(string):
  if len(string) <= 1:
    return True
  else:
    return string[0] == string[-1] and is_palindrome(string[1:-1])
print(is_palindrome("madam"))
```

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

```
def gcd(a, b):
    if b == 0:
        return a
    else:
     return gcd(b, a % b)


print(gcd(10,2))
```