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

**Ans 1:** A lambda function in Python is an anonymous function defined using the lambda keyword instead of the standard def keyword used for regular functions. Lambda functions are typically used for short, simple operations where defining a full-fledged function using def would be overkill. They are often used in situations where a function is needed for a short period and doesn't need a specific name associated with it.

Syntax of a lambda function:

lambda arguments: expression

For example :

add = lambda x, y: x + y

print(add(3, 5)) # Output: 8

The main differences between lambda functions and regular functions are:

**Syntax**:

* Lambda functions are defined using the lambda keyword followed by parameters (if any), a colon, and the expression to be evaluated.
* Regular functions are defined using the def keyword, followed by the function name, parameters in parentheses, a colon, and the block of code to be executed.

**Return Statement**:

* Lambda functions implicitly return the result of the expression they evaluate.
* Regular functions require an explicit return statement to return a value.

**Complexity**:

* Lambda functions are limited to a single expression and are suitable for simple operations.
* Regular functions can contain multiple statements and are suitable for more complex logic.

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

**them?**

**Ans 2:** Yes, lambda functions in Python can have multiple arguments. You can define and use them in a similar way as with single-argument lambda functions, separating the arguments with commas. Here's how you can define and use a lambda function with multiple arguments.

# Define a lambda function with multiple arguments

multiply = lambda x, y: x \* y # Use the lambda function with multiple arguments

result = multiply(3, 5) print(result) # Output: 15

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

**Ans 3:** Lambda functions in Python are typically used in situations where you need a short, anonymous function for a specific task, often as a one-liner within higher-order functions like map(), filter(), or sorted(). They are particularly useful when you need a function for a short period and don't want to define a full-fledged function using def, or when you want to write more concise and readable code.

Example use case of lambda functions with the sorted() function:

# List of dictionaries representing people with 'name' and 'age' keys

people = [ {'name': 'Alice', 'age': 30},

{'name': 'Bob', 'age': 25},

{'name': 'Charlie', 'age': 35} ]

# Sort the list of dictionaries by age using lambda function

sorted\_people = sorted(people, key=lambda x: x['age'])

# Print the sorted list of dictionaries

for person in sorted\_people:

print(person)

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

**Python?**

**Ans 4:**

**Advantages:**

* **Conciseness:** Lambda functions allow you to write small, anonymous functions in a single line of code. They are useful for simple operations where defining a separate named function using def would be unnecessary verbosity.
* **Readability:** Lambda functions can make the code more readable when used appropriately, especially within higher-order functions like map(), filter(), and sorted(), where a short and concise function is needed.
* **Functional Programming:** Lambda functions are commonly used in functional programming paradigms. They enable a more functional style of programming, where functions are treated as first-class citizens and can be passed around as arguments to other functions.

**Limitations:**

* **Single Expression:** Lambda functions can only contain a single expression. They are limited to simple operations and cannot contain multiple statements or complex logic.
* **Single Expression:** Lambda functions can only contain a single expression. They are limited to simple operations and cannot contain multiple statements or complex logic.
* **Readability Concerns:** While lambda functions can improve readability in some cases, they can also decrease readability if overused or used for complex operations. Using lambda functions for more complex tasks may make the code harder to understand and maintain.
* **Debugging:** Lambda functions lack meaningful names, making debugging more challenging. When an error occurs within a lambda function, it can be more difficult to identify and debug the issue compared to a regular function with a descriptive name.

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

**Explain with an example.**

**Ans 5:** Yes, lambda functions in Python can access variables defined outside of their own scope. When a lambda function is created, it retains access to the variables in the enclosing scope at the time of its creation. This behavior is known as "lexical scoping" or "closure”.

Example to illustrate how lambda functions can access variables defined outside of their own scope:

def outer\_function():

x = 10

# Define a lambda function that uses the variable 'x' from the enclosing scope

lambda\_func = lambda y: x + y

return lambda\_func

# Call the outer function to create a lambda function

my\_lambda = outer\_function()

# Call the lambda function with an argument

result = my\_lambda(5)

print("Result:", result) # Output: Result: 15

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

**Ans 6:**

square = lambda x: x\*\*2

# Test the lambda function

number = 5

result = square(number)

print("Square of", number, "is:", result) # Output: Square of 5 is: 25

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

**Ans 7:**

find\_max = lambda lst: max(lst)

# Test the lambda function

numbers = [3, 7, 2, 9, 5]

max\_value = find\_max(numbers)

print("Maximum value in the list:", max\_value) # Output: Maximum value in the list: 9

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

**Ans 8:**

# List of integers

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

# Use lambda function with filter() to filter out even numbers

filtered\_numbers = list(filter(lambda x: x % 2 == 0, numbers))

# Print the filtered list of even numbers

print("Filtered list of even numbers:", filtered\_numbers)

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

**string.**

**Ans 9:**

# List of strings

strings = ["apple", "banana", "orange", "kiwi", "grape"]

# Sort the list of strings based on the length of each string

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

# Print the sorted list of strings

print("Sorted list of strings based on length:", 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 10:**

# Define two lists

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

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

# Use lambda function with filter() to filter out common elements

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

# Print the list of common elements

print("Common elements between the two lists:", common\_elements)

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

**Ans 11:**

def factorial(n):

# Base case: factorial of 0 is 1

if n == 0:

return 1

# Recursive case: factorial of n is n times factorial of (n-1)

else:

return n \* factorial(n - 1)

# Test the recursive function

number = 5

print("Factorial of", number, "is:", factorial(number)) # Output: Factorial of 5 is: 120

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

**Ans 12:**

def fibonacci(n):

# Base cases: Fibonacci of 0 is 0, Fibonacci of 1 is 1

if n == 0:

return 0

elif n == 1:

return 1

# Recursive case: Fibonacci of n is the sum of Fibonacci of (n-1) and Fibonacci of (n-2)

else:

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

# Test the recursive function

n = 6

print("Fibonacci number at position", n, "is:", fibonacci(n)) # Output: Fibonacci number at position 6 is: 8

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

**Ans 13:**

def recursive\_sum(lst):

# Base case: if the list is empty, return 0

if not lst:

return 0

# Recursive case: sum the first element with the sum of the rest of the list

else:

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

# Test the recursive function

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

print("Sum of all elements in the list:", recursive\_sum(numbers)) # Output: Sum of all elements in the list: 15

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

**Ans 14:**

def is\_palindrome(s):

# Base case: if the length of the string is 0 or 1, it's a palindrome

if len(s) <= 1:

return True

# Recursive case: check if the first and last characters are equal,

# and then recursively check the substring without the first and last characters

else:

return s[0] == s[-1] and is\_palindrome(s[1:-1])

# Test the recursive function with some examples

print(is\_palindrome("radar")) # Output: True

print(is\_palindrome("level")) # Output: True

print(is\_palindrome("python")) # Output: False

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

**Ans 15.**

def gcd(a, b):

# Base case: if b is 0, the GCD is a

if b == 0:

return a

# Recursive case: recursively call gcd with b and the remainder of a divided by b

else:

return gcd(b, a % b)

# Test the recursive function

num1 = 48

num2 = 18

print("GCD of", num1, "and", num2, "is:", gcd(num1, num2)) # Output: GCD of 48 and 18 is: 6