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

**Solution 1.**

A lambda function, also known as an anonymous function, is a small and concise function in Python that is defined without a name. It is created using the lambda keyword and is typically used for simple, one-line operations.

Lambda functions differ from regular functions in a few ways:

* Anonymous: Lambda functions don't have a name. They are defined inline and can be used directly without assigning them to a variable.
* Simplicity: Lambda functions are typically used for simple and concise operations. They are handy for tasks that require a short function definition without the need for a full-fledged named function.
* Single expression: Lambda functions can only consist of a single expression, which is evaluated and returned automatically.
* Limited functionality: Due to their simplicity, lambda functions are limited in their functionality. They are best suited for quick, one-off operations and are not intended for complex logic or multi-line functions.

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

**Solution 2.**

Yes, a lambda function in Python can have multiple arguments. You can define multiple arguments for a lambda function by separating them with commas in the argument list. Here's the general syntax:

lambda arg1, arg2, ...: expression

Here's an example of a lambda function with multiple arguments that calculates the sum of two numbers:

addition = lambda x, y: x + y

In this example, the lambda function takes two arguments x and y and returns their sum.

We can use a lambda function with multiple arguments in the same way as a lambda function with a single argument. Here's how you can use the addition lambda function:

result = addition(3, 5)

print(result) # Output: 8

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

**Solution 3.**

Lambda functions in Python are typically used in situations where a small, anonymous function is needed for a specific task, and defining a separate named function would be unnecessary or cumbersome. Here is an example use case where lambda functions are commonly employed.

Sorting: Lambda functions are often used as key functions in sorting operations. For instance, you can use a lambda function to sort a list of strings based on their lengths:

strings = ['apple', 'banana', 'cherry', 'date']

strings.sort(key=lambda s: len(s))

print(strings) # Output: ['date', 'apple', 'cherry', 'banana']

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

**Solution 4.**

**Advantages of lambda functions:**

* Concise syntax: Lambda functions allow you to define small, one-line functions without the need for explicit function definitions, making the code more compact and readable.
* Anonymous: Lambda functions are anonymous functions, meaning they don't require a name. This is useful when you only need a function for a specific task and don't want to define a named function.
* Easy to use with higher-order functions: Lambda functions work well with higher-order functions like map(), filter(), and reduce(). They can be passed as arguments to these functions without the need for separate function definitions.
* Inline usage: Lambda functions can be used directly at the point of use without assigning them to a variable, making the code more concise and expressive.

**Limitations of lambda functions:**

* Limited functionality: Lambda functions are designed for simple, one-line expressions. They cannot contain multiple statements or complex logic like regular functions.
* Lack of documentation: Since lambda functions are anonymous, they lack the ability to provide a descriptive name or include docstrings to document their purpose and usage.
* Reduced reusability: Lambda functions are primarily intended for short, one-time use cases. They are not suitable for reusable code blocks or when the same functionality needs to be used in multiple places.
* Readability concerns: Although lambda functions can make code more concise, overly complex or nested lambda expressions can lead to reduced readability and maintainability.

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

Solution 5.

Yes, lambda functions in Python can access variables defined outside of their own scope. This is possible due to a concept called "lexical scoping" or "closure." When a lambda function is defined, it retains access to variables in the outer scope where it was defined.

Here's an example to illustrate this:

def outer\_function():

x = 10

lambda\_func = lambda y: x + y

return lambda\_func

# Create a lambda function that adds a value to x

add\_to\_x = outer\_function()

# Call the lambda function with an argument

result = add\_to\_x(5)

print(result) # Output: 15

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

**Solution 6.**

Here's an example of a lambda function that calculates the square of a given number:

square = lambda x: x \*\* 2

# Example usage

result = square(5)

print(result) # Output: 25

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

**Solution 7.**

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

numbers = [10, 25, 7, 42, 18]

maximum = lambda lst: max(lst)

result = maximum(numbers)

print(result) # Output: 42

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

**Solution 8.**

Here's an example of a lambda function that filters out all the even numbers from a list of integers:

numbers = [10, 25, 7, 42, 18]

evens = lambda lst: list(filter(lambda x: x % 2 == 0, lst))

result = evens(numbers)

print(result) # Output: [10, 42, 18]

In the above code, we define a lambda function called evens that takes a list as an argument (lst). Inside the lambda function, we use the filter() function along with another lambda function to filter out only the even numbers from the list.

The inner lambda function lambda x: x % 2 == 0 checks if a number is even (x % 2 == 0). The filter() function then applies this lambda function to each element of the list and returns a new list containing only the elements that satisfy the condition.

To use the lambda function, we call it with the numbers list as an argument, and it returns a new list (result) containing only the even numbers. In this example, the result is [10, 42, 18].

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

**Solution 9.**

Here's an example of a lambda function that sorts 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 s: len(s))

print(sorted\_strings) # Output: ['date', 'apple', 'cherry', 'banana', 'elderberry']

In the above code, we use the sorted() function to sort the strings list. The key parameter is set to a lambda function lambda s: len(s). This lambda function takes each string s from the list and returns its length using the len() function.

The sorted() function uses the lambda function as the key to determine the sorting order. It sorts the strings based on their lengths in ascending order. The resulting sorted list is assigned to the variable sorted\_strings.

When we print sorted\_strings, it will display ['date', 'apple', 'cherry', 'banana', 'elderberry'], which is the original list sorted in ascending order based on the length of each string.

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

**Solution 10**

Here's an example of a lambda function that takes two lists as input and returns a new list containing the common elements between the two lists:

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

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

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

print(common\_elements) # Output: [4, 5]

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

Solution 11

Here's an example of a recursive function in Python to calculate the factorial of a given positive integer:

def factorial(n):

# Base case: factorial of 0 or 1 is 1

if n == 0 or n == 1:

return 1

# Recursive case: multiply n with factorial of (n-1)

else:

return n \* factorial(n - 1)

# Example usage

number = 5

result = factorial(number)

print(f"The factorial of {number} is: {result}")

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

Solution 12.

Here's an example of a recursive function in Python to compute the nth Fibonacci number:

def fibonacci(n):

# Base cases: Fibonacci numbers for n = 0 and n = 1 are 0 and 1 respectively

if n == 0:

return 0

elif n == 1:

return 1

# Recursive case: Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

else:

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

# Example usage

number = 6

result = fibonacci(number)

print(f"The {number}th Fibonacci number is: {result}")

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

**Solution 13.**

Here's an example of a recursive function in Python to find the sum of all the elements in a given list:

def recursive\_sum(lst):

# Base case: If the list is empty, the sum is 0

if len(lst) == 0:

return 0

# Recursive case: Return the first element plus the sum of the rest of the elements

else:

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

# Example usage

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

result = recursive\_sum(numbers)

print(f"The sum of the elements is: {result}")

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

**Solution 14.**

Here's an example of a recursive function in Python to determine whether a given string is a palindrome:

def is\_palindrome(string):

# Base cases: If the string is empty or contains only one character, it is a palindrome

if len(string) <= 1:

return True

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

# and recursively check the remaining substring

else:

first\_char = string[0]

last\_char = string[-1]

if first\_char == last\_char:

# Remove the first and last characters and check recursively

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

else:

return False

# Example usage

word = "radar"

result = is\_palindrome(word)

if result:

print(f"{word} is a palindrome")

else:

print(f"{word} is not a palindrome")

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

**Solution 15.**

Here's an example of a recursive function in Python to find the greatest common divisor (GCD) of two positive integers:

def gcd(a, b):

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

if b == 0:

return a

# Recursive case: Calculate the GCD using Euclid's algorithm

else:

return gcd(b, a % b)

# Example usage

num1 = 36

num2 = 48

result = gcd(num1, num2)

print(f"The GCD of {num1} and {num2} is: {result}")