

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

Ans: In Python, a lambda function is a small and anonymous function that doesn't require a formal def statement. It is defined using the lambda keyword, followed by a list of parameters, a colon, and an expression. The result of the expression is automatically returned by the lambda function.

Here's an example of a lambda function that adds two numbers:

```
python
add = lambda x, y: x + y
result = add(3, 5)
print(result) # Output: 8
```

In this example, the lambda function `lambda x, y: x + y` takes two parameters `x` and `y` and returns their sum. The lambda function is assigned to the variable `add`, and then it is called with the arguments 3 and 5. The result of the lambda function, which is 8, is stored in the variable `result` and printed.

Lambda functions differ from regular functions in a few key ways:

Syntax: Lambda functions have a more concise syntax compared to regular functions. They are typically used for simple, one-line operations.

Nameless: Lambda functions are anonymous, meaning they don't have a name. They are primarily used as inline functions or for passing as arguments to other functions.

Limited functionality: Lambda functions are limited to a single expression and cannot contain multiple statements or complex logic. They are designed to be compact and focused on specific tasks.

Scope: Lambda functions have access to variables in the outer scope just like regular functions. However, they are often used with local variables defined within the lambda expression itself.

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 straightforward: you separate the arguments with commas.

Here's an example of a lambda function with multiple arguments:

```
python

multiply = lambda x, y: x * y
result = multiply(4, 6)
print(result) # Output: 24
```

In this example, the lambda function `lambda x, y: x * y` takes two arguments `x` and `y` and returns their product. The lambda function is assigned to the variable `multiply`, and then it is called with the arguments 4 and 6. The result of the lambda function, which is 24, is stored in the variable `result` and printed.

You can define lambda functions with any number of arguments by separating them with commas in the lambda expression. For example, a lambda function with three arguments would be defined like this:

```
python
```

```
my_function = lambda a, b, c: a + b + c
```

To use a lambda function with multiple arguments, you simply pass the corresponding values when calling the function, just like any other function. For instance:

```
python
```

```
result = my_function(1, 2, 3)
print(result)
```

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

Ans: Lambda functions in Python are typically used in situations where a small, anonymous function is needed for a specific task. They are commonly used in conjunction with higher-order functions, such as `map()`, `filter()`, and `reduce()`, where functions are expected as arguments.

Here's an example use case of lambda functions with the `map()` function:

```
python
```

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers) # Output: [1, 4, 9, 16, 25]
```

In this example, the `map()` function applies the lambda function `lambda x: x**2` to each element in the `numbers` list. The lambda function takes a single argument `x` and returns the square of that argument. The `map()` function iterates over the `numbers` list, applies the lambda function to each element, and returns an iterator. The `list()` function is used to convert the iterator into a list, resulting in `squared_numbers` containing the squared values.

Another common use case is with the `filter()` function to filter elements based on a condition. Here's an example:

```
python
```

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers) # Output: [2, 4]
```

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

Ans: Lambda functions in Python have several advantages and limitations compared to regular functions:

Advantages of Lambda Functions:

Concise Syntax: Lambda functions have a compact and concise syntax, allowing you to define functions in a single line of code. This can make the code more readable and reduce the need for unnecessary function definitions.

Anonymous Functions: Lambda functions are anonymous, meaning they don't require a formal name. This is useful when you need to define a small function that is used only in a specific context, without cluttering the code with named function definitions.

Higher-Order Functions: Lambda functions work well with higher-order functions, such as `map()`,

`filter()`, and `reduce()`, which expect functions as arguments. They enable you to define the function logic inline, making the code more expressive and succinct.

Simplified Functional Programming: Lambda functions support functional programming paradigms, allowing you to write code in a more declarative and expressive style. They can help in writing functional transformations and operations on data structures.

Limitations of Lambda Functions:

Single Expression: Lambda functions are limited to a single expression and cannot contain multiple statements or complex logic. This makes them unsuitable for complex tasks that require control flow structures like loops or conditional statements.

Lack of Documentation: Lambda functions don't support docstrings or other forms of documentation. It can be challenging to provide detailed explanations of their purpose and usage within the function definition itself.

Readability and Maintainability: While lambda functions can make code more concise, they can also make it less readable, especially if the expression becomes too long or complex. This can make it harder to understand the function's logic and maintain the code over time.

Limited Reusability: Lambda functions are primarily intended for short, one-time use cases. They lack the reusability and modularity of regular functions, as they cannot be easily referenced or reused in multiple parts of the codebase.

Debugging Difficulties: Lambda functions can make debugging more challenging, as they lack a formal name or traceback in error messages. When an error occurs within a lambda function, it can be more challenging to identify and locate the exact source of the issue.

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

Ans: Yes, lambda functions in Python can access variables defined outside of their own scope. They have access to variables in the enclosing scope, just like regular functions. This concept is known as "lexical scoping" or "closure."

Here's an example that demonstrates how a lambda function can access variables from the enclosing scope:

```
python
def outer_function():
    x = 10
    inner_lambda = lambda y: x + y
    return inner_lambda
```

Example usage:

```
lambda_func = outer_function()
result = lambda_func(5)
print(result) # Output: 15
```

In this example, the `outer_function()` defines a variable `x` with a value of 10. Inside `outer_function()`, a lambda function is defined and assigned to the variable `inner_lambda`.

The lambda function takes a parameter y and returns the sum of x and y.

When `outer_function()` is called and its result is assigned to `lambda_func`, it returns the lambda function object. Later, `lambda_func` is called with the argument 5, and it adds x (which is 10) to y (which is 5), resulting in 15.

The lambda function defined inside `outer_function()` has access to the variable x in its enclosing scope. This behavior is achieved through lexical scoping, where the lambda function "closes over" the variable x and retains access to it even after `outer_function()` has finished executing.

It's important to note that lambda functions can access variables from the enclosing scope, but they are limited to reading and using those variables. They cannot modify variables in the enclosing scope directly, unless they are marked as `nonlocal` or `global` using appropriate keywords (`nonlocal` or `global`).

```
python
def outer_function():
    x = 10
    inner_lambda = lambda y: nonlocal x; x + y # Invalid: SyntaxError
    return inner_lambda
```

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

Ans: Certainly! Here's the lambda function to calculate the square of a given number:

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

In this lambda function, x is the input parameter, and `x ** 2` is the expression that calculates the square of x.

You can use this lambda function to calculate the square of any number by calling it with the desired number as an argument.

Here's an example of using the lambda function to calculate the square of a number:

```
python
number = 5
result = square(number)
print(result) # Output: 25
```

In this example, the lambda function `square` is called with the argument 5, which calculates the square of 5 using the expression `5 ** 2`. The result, 25, is stored in the variable `result` and printed.

You can modify the number variable to calculate the square of any other number by calling the lambda function with a different argument.

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

Ans: Certainly! Here's a lambda function that finds the maximum value in a list of integers:

```
python
find_max = lambda lst: max(lst)
```

In this lambda function, `lst` is the input parameter, and `max(lst)` is the expression that finds the maximum value in the list.

You can use this lambda function to find the maximum value in any list of integers by calling it with the desired list as an argument.

Here's an example of using the lambda function to find the maximum value in a list:

```
python
numbers = [5, 2, 8, 1, 9, 3]
result = find_max(numbers)
print(result) # Output: 9
```

In this example, the lambda function `find_max` is called with the argument `numbers`, which is a list of integers. The lambda function uses the `max()` function to find the maximum value in the list. The result, 9, is stored in the variable `result` and printed.

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

Ans: Certainly! Here's a lambda function that filters out all the even numbers from a list of integers:

```
python
filter_even = lambda lst: list(filter(lambda x: x % 2 == 0, lst))
```

In this lambda function, `lst` is the input parameter, and `list(filter(lambda x: x % 2 == 0, lst))` is the expression that filters out the even numbers from the list.

You can use this lambda function to filter out the even numbers from any list of integers by calling it with the desired list as an argument.

Here's an example of using the lambda function to filter out the even numbers from a list:

```
python
numbers = [5, 2, 8, 1, 9, 3]
result = filter_even(numbers)
print(result) # Output: [2, 8]
```

In this example, the lambda function `filter_even` is called with the argument `numbers`, which is a list of integers. The lambda function uses the `filter()` function with the lambda function `lambda x: x % 2 == 0` to filter out the even numbers from the list. The result, `[2, 8]`, is stored in the variable `result` and printed.

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

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

Ans: Certainly! Here's a lambda function that takes two lists as input and returns a new list containing the common elements between the two lists:

```
python
find_common_elements = lambda list1, list2: list(filter(lambda x: x in list2, list1))
```

In this lambda function, `list1` and `list2` are the input parameters, and `list(filter(lambda x: x in list2, list1))` is the expression that filters out the common elements between the two lists.

You can use this lambda function to find the common elements between any two lists by calling it with the desired lists as arguments.

Here's an example of using the lambda function to find the common elements between two lists:

```
python
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
result = find_common_elements(list1, list2)
print(result) # Output: [4, 5]
```

In this example, the lambda function `find_common_elements` is called with the arguments `list1` and `list2`, which are two lists of integers. The lambda function uses the `filter()` function with the lambda function `lambda x: x in list2` to filter out the common elements between the two lists. The result, `[4, 5]`, is stored in the variable `result` and printed.

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

Ans: Certainly! Here's a recursive function in Python to calculate the factorial of a given positive integer:

```
python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

In this recursive function, the base case is when `n` equals 0, in which case the function returns 1 since the factorial of 0 is defined as 1. For any other positive integer `n`, the function recursively calls itself with `n - 1` and multiplies the result by `n`.

Here's an example of using the recursive factorial function to calculate the factorial of a number:

```
python
number = 5
result = factorial(number)
print(result) # Output: 120
```

In this example, the factorial function is called with the argument 5, which calculates the factorial of 5. The recursive calls are made as follows:

```
scss
factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1 * factorial(0)
factorial(0) = 1
```

The final result, 120, is stored in the variable `result` and printed.

11. Implement a recursive function to compute the `n`th Fibonacci number.

Ans: Certainly! Here's a recursive function in Python to compute the `n`th Fibonacci number:

```
python
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

In this recursive function, the base cases are when `n` is less than or equal to 1. For `n` equal to 0 or 1, the function returns `n` since the Fibonacci sequence starts with 0 and 1. For any other value of `n`, the function recursively calls itself with `n - 1` and `n - 2` and returns the sum of the two previous Fibonacci numbers.

Here's an example of using the recursive fibonacci function to compute the nth Fibonacci number:

```
python
number = 6
result = fibonacci(number)
print(result) # Output: 8
```

In this example, the fibonacci function is called with the argument 6, which computes the 6th Fibonacci number. The recursive calls are made as follows:

```
scss
fibonacci(6) = fibonacci(5) + fibonacci(4)
fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(0) = 0
```

The final result, 8, is stored in the variable result and printed.

You can modify the number variable to compute the nth Fibonacci number for any other positive integer by calling the fibonacci function with a different argument.

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

Ans: Certainly! Here's a recursive function in Python to find the sum of all the elements in a given list:

```
python
def sum_list_elements(lst):
    if len(lst) == 0:
        return 0
    else:
        return lst[0] + sum_list_elements(lst[1:])
```

In this recursive function, the base case is when the length of the list lst is 0. In that case, the function returns 0 since there are no elements to sum. For any non-empty list, the function recursively calls itself with the sublist starting from the second element and returns the sum of the first element and the sum of the remaining elements.

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

```
python
numbers = [1, 2, 3, 4, 5]
result = sum_list_elements(numbers)
print(result) # Output: 15
```

In this example, the sum_list_elements function is called with the argument numbers, which is a list of integers. The recursive calls are made as follows:

```
scss
sum_list_elements([1, 2, 3, 4, 5]) = 1 + sum_list_elements([2, 3, 4, 5])
sum_list_elements([2, 3, 4, 5]) = 2 + sum_list_elements([3, 4, 5])
sum_list_elements([3, 4, 5]) = 3 + sum_list_elements([4, 5])
```

$\text{sum_list_elements}([4, 5]) = 4 + \text{sum_list_elements}([5])$

$\text{sum_list_elements}([5]) = 5 + \text{sum_list_elements}([])$

$\text{sum_list_elements}([]) = 0$

The final result, 15, is stored in the variable result and printed.

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

Ans: Certainly! Here's a recursive function in Python to determine whether a given string is a palindrome:

```
python
def is_palindrome(string):
    if len(string) <= 1:
        return True
    else:
        if string[0] == string[-1]:
            return is_palindrome(string[1:-1])
        else:
            return False
```

In this recursive function, the base case is when the length of the string string is 0 or 1. In those cases, the function returns True since a string of length 0 or 1 is considered a palindrome. For any other string, the function checks if the first character is equal to the last character. If they are equal, the function recursively calls itself with the string excluding the first and last characters (string[1:-1]). If they are not equal, the function returns False.

Here's an example of using the recursive is_palindrome function to determine whether a given string is a palindrome:

```
python
word1 = "radar"
result1 = is_palindrome(word1)
print(result1) # Output: True
```

```
word2 = "python"
result2 = is_palindrome(word2)
print(result2) # Output: False
```

In this example, the is_palindrome function is called with the arguments word1 and word2, which are strings. The recursive calls are made as follows for word1:

```
scss
is_palindrome("radar") = is_palindrome("ada")
is_palindrome("ada") = is_palindrome("d")
is_palindrome("d") = True
```

The final result, True, is stored in the variable result1 and printed. For word2, the recursive calls would be different and the result would be False.

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

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

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



```
return gcd(b, a % b)
```

In this recursive function, the base case is when b is equal to 0. In that case, the function returns a since the GCD of a and 0 is a . For any other values of a and b , the function recursively calls itself with b and the remainder of a divided by b , which is calculated using the modulo operator ($a \% b$).

Here's an example of using the recursive gcd function to find the GCD of two positive integers:

```
python
num1 = 36
num2 = 48
result = gcd(num1, num2)
print(result) # Output: 12
```

In this example, the gcd function is called with the arguments `num1` and `num2`, which are two positive integers. The recursive calls are made as follows:

```
scss
gcd(36, 48) = gcd(48, 36)
gcd(48, 36) = gcd(36, 12)
gcd(36, 12) = gcd(12, 0)
```

The final result, 12, is stored in the variable `result` and printed.