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 is defined inline. It can be used as an alternative to a regular function when you need a short, one-off function. Lambda functions are typically used in conjunction with other functions, such as map() and filter().

1. 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, you simply separate the arguments with commas. For example, the following code defines a lambda function with two arguments:

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

Lambda functions are typically used in Python in conjunction with other functions, such as map () and filter ().

map () takes a function and a list as input and returns a new list where each element is the result of applying the function to the corresponding element of the input list.

Filter () takes a function and a list as input and returns a new list where each element is the element of the input list for which the function returns True.

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

Advantages of lambda functions:

Short and concise. Lambda functions are typically much shorter and more concise than regular functions. This makes them ideal for one-off tasks or for use in conjunction with other functions, such as map() and filter().

Easier to read and understand. Lambda functions are often easier to read and understand than regular functions, especially for beginners. This is because they are written in a more natural, conversational style.

More flexible. Lambda functions can be passed around as values, which make them more flexible than regular functions. This can be useful for things like chaining together functions or for passing functions as arguments to other functions.

Limitations of lambda functions:

Limited functionality. Lambda functions are limited in their functionality compared to regular functions. They can only contain a single expression, and they cannot have a docstring or a name.

Not suitable for complex tasks. Lambda functions are not suitable for complex tasks. If you need to write a function that does more than one thing or that needs to be reusable, then you should use a regular function.

Can be overused. Lambda functions can be overused. If you find yourself using lambda functions for everything, then you should probably consider using regular functions instead.

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

No, lambda functions in Python cannot access variables defined outside of their own scope. This is because lambda functions are created inline, and they do not have their own local namespace. Instead, they inherit the local namespace of the function in which they are created.

If you need to access a variable that is defined outside of the lambda function's scope, you can do so by using the global keyword. For example, the following code defines a global variable called x, and then creates a lambda function that uses x:

Explain with an example.

x = 10

def foo():

lambda\_func = lambda: x \* 2

print(lambda\_func())

foo()

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

Here is a lambda function to calculate the square of a given number:

square = lambda x: x \*\* 2

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

Here is a lambda function to find the maximum value in a list of integers:

max\_value = lambda numbers: max(numbers)

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

Here is a lambda function to filter out all the even numbers from a list of integers:

even\_filter = lambda numbers: filter(lambda x: x % 2 == 0, numbers)

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

Here is a lambda function to sort a list of strings in ascending order based on the length of each string:

sort\_by\_length = lambda strings: sorted(strings, key=lambda x: len(x))

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

Here is a lambda function that takes two lists as input and returns a new list containing the common elements between the two lists:

common\_elements = lambda list1, list2: list(set(list1) & set(list2))

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

Here is 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)

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

Here is a recursive function to compute the nth Fibonacci number:

def fib(n):

if n == 0 or n == 1:

return n

else:

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

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

Here is a recursive function to find the sum of all the elements in a given list:

def sum\_list(list):

if len(list) == 0:

return 0

else:

return list[0] + sum\_list(list[1:])

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

Here is 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])

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

Here is 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)