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

Ans A lambda function in Python is an anonymous, small, and inline function defined using the `lambda` keyword. It's typically used for simple operations and short functions. Unlike regular functions defined with the `def` keyword, lambda functions are limited in their functionality and usually consist of a single expression. They're often used for situations where you need a quick, throwaway function without defining a formal function using `def`.

Here's an example of a regular function and its equivalent as a lambda function:

Regular function:

```python

def add(x, y):

return x + y

```

Lambda function:

```python

add = lambda x, y: x + y

```

In this case, the lambda function version is shorter and doesn't require a separate function definition. However, lambda functions aren't suitable for more complex logic or functions that require multiple statements or extensive documentation.

1. 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. You can define and use multiple arguments in a lambda function just like you would with a regular function. The syntax for a lambda function with multiple arguments is as follows:

```python

lambda arguments: expression

```

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

```python

multiply = lambda x, y: x \* y

result = multiply(3, 5) # This will result in 15

```

In this example, the lambda function `multiply` takes two arguments, `x` and `y`, and returns their product. You can pass values to the lambda function when calling it, just like you would with a regular function.

Keep in mind that while lambda functions are convenient for simple operations with a concise syntax, they may become less readable for more complex logic due to their limited ability to contain only a single expression.

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

Ans. Lambda functions in Python are often used in situations where you need a simple, short-lived function without the need for a formal function definition using the `def` keyword. They are particularly useful when you need to pass a function as an argument to another function or when you want to create small functions on the fly.

Here's an example use case of lambda functions: sorting a list of tuples based on the second element of each tuple.

```python

data = [(3, 12), (1, 8), (5, 6), (2, 10)]

# Using a lambda function to sort based on the second element of each tuple

sorted\_data = sorted(data, key=lambda x: x[1])

print(sorted\_data)

```

In this example, the `sorted` function takes a `key` parameter that specifies a function to determine the sorting order. We use a lambda function `lambda x: x[1]` as the key function, which extracts the second element of each tuple for comparison. As a result, the list is sorted based on the second element of each tuple, producing the output `[(5, 6), (1, 8), (2, 10), (3, 12)]`.

Lambda functions are particularly handy in scenarios like this, where you need a small function for a specific purpose without defining a full-fledged named function.

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

Ans Lambda Functions (Advantages):

1. \*\*Concise Syntax:\*\* Lambda functions are shorter and more compact, making them suitable for simple operations that don't require extensive logic.

2. \*\*Inline Usage:\*\* Lambda functions can be used directly within expressions, making code more readable by avoiding the need to define a separate named function.

3. \*\*Functional Programming:\*\* Lambda functions are often used in functional programming paradigms where functions can be treated as first-class citizens.

Lambda Functions (Limitations):

1. \*\*Limited Complexity:\*\* Lambda functions are limited to a single expression, making them unsuitable for complex logic or multiple statements.

2. \*\*Readability:\*\* Lambda functions can become less readable if the expression becomes too long or complex, reducing code maintainability.

3. \*\*Limited Documentation:\*\* Lambda functions lack the ability to include docstrings, which provide essential documentation for named functions.

4. \*\*Reduced Reusability:\*\* Since lambda functions are anonymous and cannot be reused elsewhere, they can lead to code duplication if similar logic is needed in multiple places.

5. \*\*Debugging:\*\* Debugging can be more challenging with lambda functions due to the absence of meaningful function names in error messages.

6. \*\*Scope Limitation:\*\* Lambda functions are limited to the expression's scope and cannot include statements or additional operations beyond that.

1. 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 are able to access variables defined outside of their own scope. This includes variables defined in the enclosing function or in the global scope. This behavior is known as "lexical scoping" or "closure."

Here's an example to illustrate this:

```python

def outer\_function(x):

multiplier = 2

# Define a lambda function that uses the 'multiplier' variable from the outer function

calculate = lambda y: y \* multiplier

return calculate(x)

result = outer\_function(5)

print(result) # Output will be 10

```

In this example, the lambda function `calculate` is defined inside the `outer\_function`. It uses the `multiplier` variable, which is defined in the enclosing scope of `outer\_function`. When `outer\_function` is called with the argument `5`, the lambda function is also called with `y` set to `5`, and it multiplies `5` by the `multiplier` value, resulting in `10`.

.

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

Ans. Certainly! Here's a lambda function that calculates the square of a given number:

```python

square = lambda x: x \*\* 2

```

You can use this lambda function by calling it with a number as an argument, like this:

```python

result = square(4) # This will result in 16

```

The lambda function `square` takes one argument `x` and returns its square using the exponentiation operator `\*\*`.

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

Ans.

find\_max = lambda lst: max(lst)

numbers = [8, 15, 4, 27, 12]

max\_value = find\_max(numbers)

# This will result in 27

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

Ans

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

numbers = [8, 15, 4, 27, 12]

even\_numbers = filter\_even(numbers) # This will result in [8, 4, 12]

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

Ans

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

.

words = ["apple", "banana", "cherry", "date"]

sorted\_words = sort\_by\_length(words) # This will result in ["date", "apple", "cherry", "banana"]

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

Ans

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

list1 = [2, 4, 6, 8, 10]

list2 = [6, 8, 12, 14]

common\_elements = find\_common\_elements(list1, list2) # This will result in [6, 8]

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

Ans

def factorial(n):

if n == 0 or n == 1:

return 1

else:

return n \* factorial(n - 1)

# Example usage

result = factorial(5) # This will result in 120

print(result)

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

Ans

def fibonacci(n):

if n <= 0:

return 0

elif n == 1:

return 1

else:

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

# Example usage

result = fibonacci(7) # This will result in 13

print(result)

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

Ans

def recursive\_sum(lst):

if len(lst) == 0:

return 0

else:

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

# Example usage

my\_list = [1, 2, 3, 4, 5]

total\_sum = recursive\_sum(my\_list)

print("Sum of elements:", total\_sum)

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

Ans

def is\_palindrome(s):

s = s.lower().replace(" ", "") # Convert to lowercase and remove spaces

if len(s) <= 1:

return True

elif s[0] == s[-1]:

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

else:

return False

# Example usage

test\_string = "racecar"

if is\_palindrome(test\_string):

print(f"'{test\_string}' is a palindrome.")

else:

print(f"'{test\_string}' is not a palindrome.")

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

Ans

def gcd(a, b):

if b == 0:

return a

else:

return gcd(b, a % b)

# Example usage

num1 = 48

num2 = 18

result = gcd(num1, num2)

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