Python Basic Assignment – 11 June

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

In Python, a lambda function is a small anonymous function that can have any number of arguments but can only have one expression. It is created using the lambda keyword. The following are the key differences between lambda functions and regular functions:

* **Syntax:** Lambda functions are defined in a single line using the lambda keyword, while regular functions are defined using the def keyword and can span multiple lines.
* **Name:** Lambda functions don't have a name. In contrast, regular functions must have a name.
* **Function body**: Lambda functions can only have one expression in their body. In contrast, regular functions can have multiple statements and a more complex function body.
* **Return statement:** Lambda functions automatically return the result of the expression without needing an explicit return statement. Regular functions require a return statement to return a value explicitly.

add = lambda x, y: x + y

result = add(3, 5) # Result will be 8

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. Lambda functions can take any number of arguments, just like regular functions, but they can only have one expression in their body.

area = lambda length, width: length \* width

area\_val = area(5, 3) # Result will be 15

In this example, the lambda function ‘area’ takes two arguments length and width and returns their product, which represents the area of a rectangle. When calling the lambda function with arguments (5, 3), it returns 15, which is the area of a rectangle with length 5 and width 3.

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

Lambda functions in Python are commonly used in situations where a small, simple function is required for a short duration, and defining a full-fledged named function would be unnecessary or overly verbose. They are particularly useful when you need to pass a function as an argument to higher-order functions like map(), filter(), or sort().

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

squared\_numbers = list(map(lambda x: x \*\* 2, numbers))

print(squared\_numbers) #Output: [1, 4, 9, 16, 25]

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

**Advantages:**

* **Conciseness**: It allows you to define small, single-expression functions in a single line, which can make your code more concise and easier to read.
* **Readability:** It can make the code more readable, as you don't need to define a separate function with a name and a function body.
* **Avoiding Named Function Overhead**: In some cases, you might need a function only once and don't want to bother naming it.
* **Function as First-class Citizens:** You can pass them as arguments to other functions, return them from functions, and store them in data structures.
* **Use with Higher-order Functions**: Lambda functions are frequently used with higher-order functions like map(), filter(), sort(), and reduce(), which expect a function as one of their arguments. Using lambda functions in these cases can make the code more expressive and reduce the need for explicit loops.

**Limitations:**

* **Single Expression**: Lambda functions are restricted to a single expression in their body. This makes them unsuitable for more complex operations that require multiple statements or a more elaborate function body.
* **No Statements or Documentation**: Lambda functions can't include statements like print(), assert, or raise. They are limited to expressions only. Additionally, they can't include documentation strings (docstrings) to describe their functionality.
* **Lack of Named Functions**: Lambda functions are anonymous, which means they don't have a name. This can make debugging and error messages less clear since the function name won't be available in the traceback.
* **Reduced Readability for Complex Operations**: While lambda functions can improve readability for simple operations, they can become less readable for more complex operations. In such cases, using a regular named function with meaningful names and a clear function body might be more appropriate.
* **Limited Reusability**: Lambda functions are not reusable since they lack a name. If you find yourself needing the same function in multiple places, a regular named function is a better choice.

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

Yes, lambda functions in Python can access variables defined outside of their own scope. This ability is known as "lexical scoping" or "closure." Lambda functions can capture and remember variables from the enclosing scope where they are defined, even after the enclosing scope has finished executing.

def outer\_function(x):

inner\_lambda = lambda y: x + y

return inner\_lambda

closure\_function = outer\_function(10)

result = closure\_function(5)

print(result) # Output will be 15

This example demonstrates that lambda functions can access and "close over" variables from the surrounding scope, making them powerful tools for creating specialized functions with captured state.

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

# Lambda function to calculate the square of a number

square = lambda x: x \*\* 2

# Using the lambda function

result = square(5) # Result will be 25

print(result)

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

# List of integers

numbers = [15, 8, 23, 42, 12, 5]

# Lambda function to find the maximum value in the list

find\_max = lambda lst: max(lst)

# Using the lambda function

result = find\_max(numbers) # Result will be 42

print(result)

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

# List of integers

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

# Lambda function to filter out even numbers from the list

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

# Using the lambda function

result = filter\_even(numbers) # Result will be [2, 4, 6, 8, 10]

print(result)

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

# List of strings

words = ['apple', 'banana', 'grape', 'orange', 'kiwi', 'pear']

# Lambda function to sort the list based on string length

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

# Using the lambda function

result = sort\_by\_length(words)

# Result will be ['kiwi', 'pear', 'grape', 'apple', 'banana', 'orange']

print(result)

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

# Two input lists

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

list2 = [3, 4, 5, 6, 7]

# Lambda function to find common elements

find = lambda lst1, lst2: list(filter(lambda x: x in lst1, lst2))

# Using the lambda function

result = find (list1, list2) # Result will be [3, 4, 5]

print(result)

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

def factorial(n):

if n == 0 or n == 1:

return 1

else:

return n \* factorial(n - 1)

# Test the function

num = int(input())

result = factorial(num)

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

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

def fibonacci(n):

if n <= 0:

raise ValueError("Invalid input.")

elif n == 1:

return 0

elif n == 2:

return 1

else:

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

# Test the function

num = int(input())

result = fibonacci(num)

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

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

def recursive\_sum(lst):

if not lst:

return 0

else:

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

# Test the function

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

result = recursive\_sum(numbers)

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

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

def is\_palindrome(s):

s = s.lower()

if len(s) <= 1:

return True

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

return False

else:

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

# Test the function

string1 = input()

print(f"{string1} is a palindrome: {is\_palindrome(string1)}")

1. **Implement 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)

num1 = int(input())

num2 = int(input())

result = gcd(num1, num2)

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