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

* In Python, a lambda function is an anonymous function that is defined without a name. It differs from a regular function by its conciseness, single expression and no function name.

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. We can define and use multiple arguments in a lambda function by separating them with commas, similar to how we define multiple parameters in a regular function.

For e.g.:

sum = lambda x, y: x + y

print(sum(3, 4)) #Output: 7

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, anonymous function is required for a specific task and defining a regular named function would be unnecessary.

For e.g.:

students = [

{"name": "Alisa", "marks": 85},

{"name": "Arpita", "marks": 92},

{"name": "Aditya", "marks": 88},

{"name": "Anup", "marks": 79}

]

# Sort the list of students by their marks

sorted\_students = sorted(students, key=lambda student: student["marks"])

print(sorted\_students)

#Output is as follows:

#[{'name': 'Anup', 'marks': 79}, {'name': 'Alisa', 'marks': 85}, {'name': 'Aditya', 'marks': 88}, #{'name': 'Arpita', 'marks': 92}]

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

* The advantages of lambda functions include conciseness in coding, not requiring a formal name and enabling functional programming. The limitations of lambda functions include limited readability due to single line expression and debugging problems due to lack of a formal name.

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 are able to access variables defined outside of their own scope. This concept is known as "lexical scoping" or "closure." When a lambda function is defined, it retains access to variables in the surrounding scope where it was created, even if that scope is no longer active. For e.g.:

def f():

x = 10

return lamda y: x + y

my\_lamda = f()

result = m\_lambda(5)

print(result) #Output: 15

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

* A lambda function to calculate the square of a given number is as follows:

square = lambda x: x \*\* 2

result = square(5) #Example usage

print(result) #Output: 25

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

* A lambda function to calculate the maximum value in a list of integers is as follows:

n = [5, 10, 3, 8, 15, 1]

maximum = lambda n: max(n)

result = maximum(n)

print(result) # Output: 15

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

* A lambda function to filter out all the even numbers from a list of integers is as follows:

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

even = list(filter(lambda x: x % 2 == 0, n))

print(even) #Output: [2, 4, 6, 8, 10]

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

* A lambda function to sort a list of strings in ascending order based on the length of each string is as follows:

strings = ["apple", "banana", "cherry", "pear", "pineapple"]

sorted\_strings = sorted(strings, key=lambda s: len(s))

print(sorted\_strings)

#Output: ['pear', 'apple', 'cherry', 'banana', 'pineapple']

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

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

list1 = [1, 29, 13, 14, 65]

list2 = [14, 65, 6, 73, 88]

common = lambda a, b: list(filter(lambda x: x in b, a))

result = common(list1, list2)

print(result) #Output: [14, 65]

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

* A recursive function to calculate the factorial of a given positive integer is as follows:

def factorial(n):

if n == 1:

return 1

elif n <= 0:

return “Enter a positive integer.”

else:

return n \* factorial(n-1)

#Example usage

factorial(5) #Output: 120

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

* A recursive function to compute the nth Fibonacci number is as follows:

def fibonacci(n):

if n <= 0:

return “Enter a positive integer.”

elif n == 1:

return 0

elif n == 2:

return 1

else:

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

#Example usage

fibonacci(13) #Output: 144

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

* A recursive function to find the sum of all the elements in a given list is as follows:

def list\_sum(n):

if not n:

return 0

else:

return n[0] + list\_sum(n[1:])

#Example usage

list\_sum([1, 2, 3]) #Output: 6

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

* A recursive function to determine whether a given string is a palindrome is as follows:

def is\_palindrome(n):

if len(n) <= 1:

return True

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

return False

else:

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

#Example usage

is\_palindrome(‘one’) #Output: False

is\_palindrome(‘mom’) #Output: True

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

* A recursive function to find the greatest common divisor (GCD) of two positive integers is as follows:

def gcd(a, b):

if b == 0:

return a

else:

return gcd(b, a % b)

#Example usage

gcd(2, 4) #Output: 2