1) What is the difference between a function and a method in Python?

Ans= Function: A standalone block of code, defined with def, and called directly.

Example:

```
def greet(name):
  return f"Hello, {name}!"
print(greet("Alice"))
```

Method: A function defined inside a class, called on an object, and typically uses self to access attributes.

Example:

```
class Greeter:
   def greet(self, name):
       return f"Hello, {name}!"
obj = Greeter()
print(obj.greet("Alice"))
```

Key Difference: Functions are independent; methods are tied to objects or classes.

2. Explain the concept of function arguments and parameters in Python?

Ans= Parameters: Variables defined in a function's definition to accept input.

Example:

```
def greet(name):  # 'name' is the parameter
   print(f"Hello, {name}!")
```

Arguments: Actual values passed to the function when calling it.

Example:

```
greet("Alice")  # "Alice" is the argument
```

Types of Arguments:

1. Positional: Matched by order.

```
def add(a, b):
    return a + b
add(3, 5)  # Output: 8
```

2. Keyword: Matched by name.

```
add(b=5, a=3)  # Output: 8
```

3. Default: Uses default value if none provided.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
greet()  # Output: Hello, Guest!
```

4. Variable-Length:

*args (tuple): Multiple positional arguments.

**kwargs (dict): Multiple keyword arguments.

3. What are the different ways to define and call a function in Python?

Ans = Python, functions can be defined and called in various ways depending on their purpose. Below are the different ways to define and call functions:

Here are the main ways to define and call a function in Python:

1. Standard Function

Definition: Use def.

Call: By function name.

```
def greet(name):
    return f"Hello, {name}!"
print(greet("Alice"))  # Output: Hello, Alice!
```

2. Default Parameters

Definition: Assign default values to parameters.

Call: With or without arguments.

```
def greet(name="Guest"):
    return f"Hello, {name}!"
print(greet())  # Output: Hello, Guest!
```

3. Lambda Function

Definition: Use lambda for one-line functions.

Call: Inline or through a variable.

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
```

4. Recursive Function

Definition: A function that calls itself.

```
def factorial(n):
    if n == 1: return 1
    return n * factorial(n - 1)
print(factorial(5))  # Output: 120
```

5. Variable-Length Arguments

4. What is the purpose of the `return` statement in a Python function?

Ans = The return statement in a Python function is used to send a value or result from the function back to the caller. It signifies the end of the function execution and specifies the output of the function.

Purpose of return in a Python Function

1. To Provide Output:

A function processes input or performs calculations, and return sends the result back to the code that called the function.

```
def add(a, b):

   return a + b

result = add(3, 4)

print(result)  # Output: 7
```

2. To Terminate Function Execution:

When a return statement is encountered, the function stops executing further code.

```
def example():

   print("Before return")

   return "End"

   print("After return")  # This won't execute

print(example())  # Output: Before return\nEnd
```

3. To Return Multiple Values:

Python allows returning multiple values as a tuple.

5. What are iterators in Python and how do they differ from iterables?

Ans = Iterators: Objects that allow iteration over a sequence, using _iter() and __next_() methods. Once exhausted, they cannot be re-used without being recreated.

Example:

```
my_list = [1, 2, 3]
```

```
iterator = iter(my_list)

print(next(iterator))  # Output: 1
```

Iterables: Objects that can return an iterator (e.g., lists, tuples, strings). They implement the _iter() method but don't necessarily implement __next_().

Example:

```
my_list = [1, 2, 3]

for item in my_list:  # Iterating over the iterable

    print(item)
```

Difference:

Iterable: Can be looped over (e.g., using for loop).

Iterator: A specific type of iterable that tracks the current position and can be manually advanced with next().

6. Explain the concept of generators in Python and how they are defined?

Ans = Generators in Python are special iterators that yield values one at a time, instead of returning all values at once, making them memory efficient.

How to Define Generators:

1. Generator Function: Use yield instead of return.

```
def count_up_to(n):

    count = 1

    while count <= n:

        yield count

        count += 1
```

```
gen = count_up_to(3)

print(next(gen))  # Output: 1
```

2. Generator Expression: Similar to list comprehension, but with parentheses.

```
gen = (x * 2 for x in range(5))

print(next(gen))  # Output: 0
```

Advantages:

Memory Efficient: Generates values on demand.

Lazy Evaluation: Computes values only when needed.

7. What are the advantages of using generators over regular functions?

Ans = Advantages of Using Generators Over Regular Functions:

1. Memory Efficiency:

Generators yield one item at a time, which means they don't store the entire sequence in memory. This is particularly useful when working with large datasets or streams of data.

Regular Functions: Typically return all results at once, which can be memory-intensive if the dataset is large.

Example:

Generators work well with large files or data streams without consuming excessive memory.

2. Lazy Evaluation:

Generators compute values only when requested (on demand), which can save processing time, especially if not all values are needed.

Regular Functions: Compute and return all values immediately, even if they are not all used.

3. Improved Performance:

Generators can be more efficient in terms of CPU usage, as they don't generate unnecessary results. They allow iterating through large sequences without pre-computing everything.

Regular Functions: Often require generating the full result set in advance, which can be slower and less efficient.

4. Cleaner Code:

Generators allow a simpler and more readable implementation for producing sequences, especially when compared to writing loops and managing the state manually.

Regular Functions: Often require additional state management to yield values one by one.

8. What is a lambda function in Python and when is it typically used?

Ans = A lambda function is a small anonymous function defined using the lambda keyword. Unlike regular functions that are defined with def, lambda functions are concise and typically used for short-term, one-off operations.

Syntax:

lambda arguments: expression

Example:

add = lambda x, y: x + y

print(add(3, 5)) # Output: 8

1. Short, One-Line Functions: When you need a simple function for a short task, a lambda provides a compact syntax. A lambda function is a small anonymous function defined using the lambda keyword. Unlike regular functions that are defined with def, lambda functions are concise and typically used for short-term, one-off operations.

Syntax:

lambda arguments: expression

Example:

add = lambda x, y: x + y

print(add(3, 5)) # Output: 8

When is Lambda Typically Used?:

squares = list(map(lambda x: x**2, [1, 2, 3, 4]))

print(squares) # Output: [1, 4, 9, 16]

2. As an Argument to Higher-Order Functions: Lambda functions are often used as arguments to functions like map(), filter(), and sorted().

sorted_list = sorted([1, 3, 2], key=lambda x: -x)

print(sorted_list) # Output: [3, 2, 1]

3. When a Named Function is Overkill: When the function is simple enough that it doesn’t require a full function definition.

9. Explain the purpose and usage of the `map()` function in Python?

Ans= The map() function in Python is used to apply a given function to all items in an iterable (like a list, tuple, etc.) and return an iterator that produces the results. It allows you to process data without needing an explicit loop.

Syntax:

map(function, iterable)

function: The function that is applied to each item in the iterable.

iterable: The iterable (e.g., list, tuple) whose elements are processed by the function.

Usage:

1. Applying a Function to Each Element: The map() function applies the given function to every item in the iterable.

Example:

numbers = [1, 2, 3, 4]

squares = map(lambda x: x**2, numbers)

print(list(squares)) # Output: [1, 4, 9, 16]

2. Using Multiple Iterables: You can pass more than one iterable to map(). The function must then accept as many arguments as there are iterables.

Example:

list1 = [1, 2, 3]

list2 = [4, 5, 6]

result = map(lambda x, y: x + y, list1, list2)

print(list(result)) # Output: [5, 7, 9]

3. Returning a Map Object: The map() function returns a map object, which is an iterator. To get a list, you need to convert it using list() or another collection type.

Example:

result = map(str, [1, 2, 3])

print(list(result)) # Output: ['1', '2', '3']

10. What is the difference between `map()`, `reduce()`, and `filter()` functions in Python?

Ans = Difference Between map(), reduce(), and filter() in Python

All three functions—map(), reduce(), and filter()—are used for functional programming and help in processing iterables, but they serve different purposes and have distinct behaviors.

1. map() Function

Purpose: Applies a given function to each item in an iterable (or iterables) and returns an iterator with the results.

Output: Returns a map object (an iterator), which can be converted to a list or other collection types.

Usage: When you need to transform or modify each element in an iterable.

Example:

```
numbers = [1, 2, 3]

squares = map(lambda x: x**2, numbers)

print(list(squares))  # Output: [1, 4, 9]
```

2. reduce() Function

Purpose: Applies a binary function (a function that takes two arguments) cumulatively to the items of an iterable, reducing it to a single value.

Output: A single accumulated result, not an iterable.

Usage: When you want to reduce a sequence of values to a single result (e.g., sum, product, maximum).

Example:

```
from functools import reduce

numbers = [1, 2, 3, 4]

result = reduce(lambda x, y: x + y, numbers)

print(result)  # Output: 10 (1 + 2 + 3 + 4)
```

3. filter() Function

Purpose: Filters elements from an iterable by applying a function that returns True or False. Only the elements where the function returns True are included in the result.

Output: Returns an iterator that includes only elements where the function returned True.

Usage: When you need to select items from an iterable based on a condition.

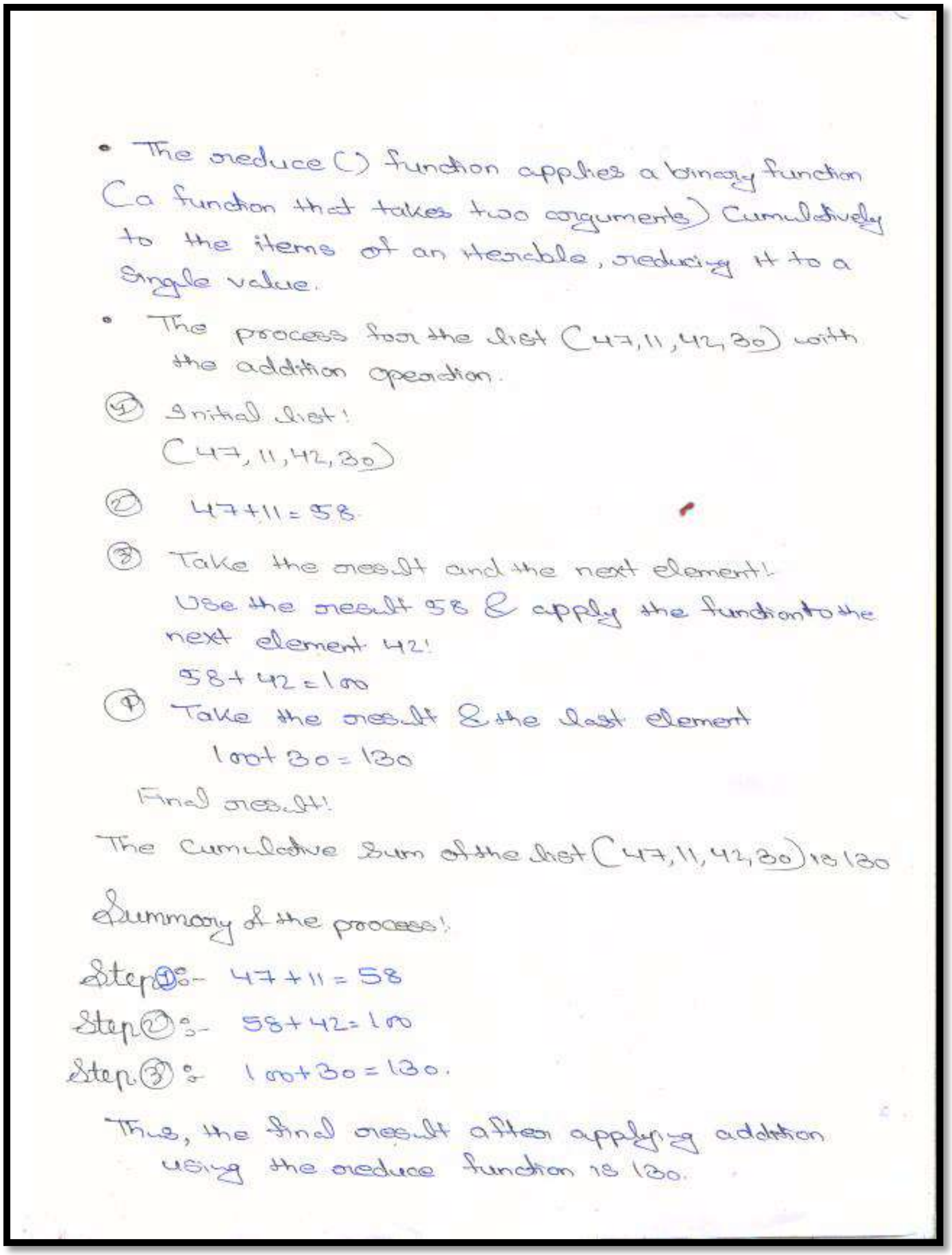
Example:

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

even_numbers = filter(lambda x: x % 2 == 0, numbers)

print(list(even_numbers))  # Output: [2, 4]
```

11. Using pen & Paper write the internal mechanism for sum operation using reduce function on this given list:[47,11,42,13];



- The reduce () function applies a binary function (a function that takes two arguments) cumulatively to the items of an iterable, reducing it to a single value.
- The process for the list (47,11,42,30) with the addition operation.

① Initial list!
(47, 11, 42, 30)

② 47+11 = 58.

③ Take the result and the next element!
Use the result 58 & apply the function to the next element 42!
58 + 42 = 100

④ Take the result & the last element
100 + 30 = 130

Final result!

The cumulative sum of the list (47, 11, 42, 30) is 130

Summary of the process!

Step ①:- 47 + 11 = 58

Step ②:- 58 + 42 = 100

Step ③: 100 + 30 = 130.

Thus, the final result after applying addition using the reduce function is 130.

Practical Questions:

1. Write a Python function that takes a list of numbers as input and returns the sum of all even numbers in the list?

Ans = A Python function that takes a list of numbers as input and returns the sum of all even numbers in the list:

```
def sum_even_numbers(numbers):

   # Using list comprehension to filter even numbers and sum them

   return sum(num for num in numbers if num % 2 == 0)

# Example usage:

numbers = [47, 11, 42, 30]

result = sum_even_numbers(numbers)

print("Sum of even numbers:", result)
```

Explanation:

The function sum_even_numbers(numbers) takes a list of numbers as input.

It uses a generator expression inside the sum() function to filter even numbers (num % 2 == 0) and calculate the sum.

The result is the sum of the even numbers in the list.

Example Output:

For the input list [47, 11, 42, 30], the output will be:

```
Sum of even numbers: 72
```

2. Create a Python function that accepts a string and returns the reverse of that string?

Ans = a Python function that accepts a string as input and returns the reversed version of that string:

```
def reverse_string(s):

   # Using slicing to reverse the string
```

```
    return s[::-1]

# Example usage:

input_string = "hello"

result = reverse_string(input_string)

print("Reversed string:", result)
```

Explanation:

The function reverse_string(s) takes a string s as input.

It uses slicing (s[::-1]) to reverse the string.

The slice [::-1] means start from the end and step backwards, effectively reversing the string.

Example Output:

For the input string "hello", the output will be:

```
Reversed string: olleh
```

3. Implement a Python function that takes a list of integers and returns a new list containing the squares of each number?

Ans= A Python function that takes a list of integers as input and returns a new list containing the square of each number:

```
def square_numbers(numbers):

    # Using list comprehension to square each number in the list

    return [num ** 2 for num in numbers]

# Example usage:

input_list = [1, 2, 3, 4]
```

```
result = square_numbers(input_list)

print("Squared numbers:", result)
```

Explanation:

The function square_numbers(numbers) takes a list of integers numbers as input.

It uses a list comprehension to create a new list where each element is the square of the corresponding number from the input list.

Example Output:

For the input list [1, 2, 3, 4], the output will be:

```
Squared numbers: [1, 4, 9, 16]
```

4. Write a Python function that checks if a given number is prime or not from 1 to 200.

Ans = a Python function that checks if a given number between 1 and 200 is prime:

```
def is_prime(num):

 if num < 2:  # Numbers less than 2 are not prime

        return False

    for i in range(2, int(num ** 0.5) + 1):  # Check up to the square root of num

        if num % i == 0:  # If divisible by any number in the range, it's not prime

            return False

    return True

# Example usage:

for number in range(1, 201):

    if is_prime(number):

        print(number, "is a prime number")
```

Explanation:

The function is_prime(num) checks if num is prime:

If num is less than 2, it's not prime.

It checks divisibility from 2 up to the square root of num (which is more efficient than checking all numbers up to num).

If no divisors are found, it returns True (indicating the number is prime).

Otherwise, it returns False (indicating the number is not prime).

Example Output:

For the numbers from 1 to 200, it will print:

```
2 is a prime number

3 is a prime number

5 is a prime number

7 is a prime number

199 is a prime number
```

5. Create an iterator class in Python that generates the Fibonacci sequence up to a specified number of terms?

Ans= a Python iterator class that generates the Fibonacci sequence for a specified number of terms:

```
class FibonacciIterator:

    def _init_(self, n):

        self.n = n  # Number of terms in the Fibonacci sequence

        self.a, self.b = 0, 1  # Initializing the first two Fibonacci numbers

        self.count = 0  # Count to track the number of terms generated

 def _iter_(self):

        return self  # The iterator object itself

 def _next_(self):
        if self.count < self.n:  # Continue generating until n terms are reached

            fib_number = self.a

            self.a, self.b = self.b, self.a + self.b  # Update to next Fibonacci numbers

            self.count += 1

            return fib_number

        else:

            raise StopIteration  # Stop when the specified number of terms is reached


# Example usage:

fibonacci = FibonacciIterator(10)  # Generate first 10 Fibonacci numbers

for num in fibonacci:

    print(num)
```

Explanation:

The class FibonacciIterator implements an iterator to generate the Fibonacci sequence.

init(self, n): Initializes the iterator with the desired number of terms n and sets the first two Fibonacci numbers (a = 0, b = 1).

iter(self): Returns the iterator object itself (standard for an iterable object).

next(self): Computes the next Fibonacci number and updates a and b for the next iteration. Once the specified number of terms is generated, it raises a StopIteration exception to signal the end of the iteration.

Example Output:

For generating the first 10 Fibonacci numbers, the output will be:

```
0

1
```

```
1

2

3

5

8

13

21

34
```

6. Write a generator function in Python that yields the powers of 2 up to a given exponent?

Ans = A Python generator function that yields the powers of 2 up to a given exponent:

```
def powers_of_two(exponent):
    for i in range(exponent + 1):
        yield 2 ** i  # Yield 2 raised to the power of i
# Example usage:
for power in powers_of_two(5):
    print(power)
```

Explanation:

The function powers_of_two(exponent) generates powers of 2, starting from 2^0 up to 2^exponent.

It uses the yield keyword to produce each power of 2 one at a time.

The loop runs from 0 to the given exponent, and 2 ** i is yielded at each step.

Example Output:

For an exponent of 5, the output will be:

```
1

2

4

8

16

32
```

7. Implement a generator function that reads a file line by line and yields each line as a string?

Ans = A Python generator function that reads a file line by line and yields each line as a string:

```
def read_file_line_by_line(file_path):

    with open(file_path, 'r') as file:

        for line in file:

            yield line.strip()  # Yield each line, stripping the newline character

# Example usage:

file_path = 'example.txt'  # Replace with the path to your file

for line in read_file_line_by_line(file_path):

    print(line)
```

Explanation:

The read_file_line_by_line(file_path) function opens the file at the given file_path and iterates through each line in the file.

The yield statement is used to return each line one by one.

The strip() method is used to remove the trailing newline character from each line.

This generator allows processing each line of the file without loading the entire file into memory, making it memory efficient.

Example Usage:

If example.txt contains:

Hello, world!

This is a test.

Python generators are cool.

The output would be:

Hello, world!

This is a test.

Python generators are cool.

8. Use a lambda function in Python to sort a list of tuples based on the second element of each tuple?

Ans = a lambda function in Python to sort a list of tuples based on the second element of each tuple:

# Sample list of tuples

```
tuples_list = [(1, 3), (4, 1), (2, 5), (6, 2)]


# Sorting the list of tuples based on the second element using a lambda function

sorted_list = sorted(tuples_list, key=lambda x: x[1])

# Printing the sorted list

print(sorted_list)
```

Explanation:

sorted(tuples_list, key=lambda x: x[1]): The sorted() function sorts the tuples_list.

The key parameter specifies a function to be applied to each element of the list before sorting. Here, lambda x: x[1] returns the second element of each tuple (since tuple indices start at 0).

The result is a list sorted by the second element of each tuple.

Example Output:

For the input list [(1, 3), (4, 1), (2, 5), (6, 2)], the output will be:

```
[(4, 1), (6, 2), (1, 3), (2, 5)]
```

9. Write a Python program that uses `map()` to convert a list of temperatures from Celsius to Fahrenheit?

Ans = a Python program that uses the map() function to convert a list of temperatures from Celsius to Fahrenheit:

```
# Function to convert Celsius to Fahrenheit

def celsius_to_fahrenheit(celsius):

   return (celsius * 9/5) + 32

# List of temperatures in Celsius

celsius_temperatures = [0, 20, 30, 40, 100]
```

```
# Using map() to convert Celsius to Fahrenheit

fahrenheit_temperatures = map(celsius_to_fahrenheit, celsius_temperatures)


# Converting the result to a list and printing

print(list(fahrenheit_temperatures))
```

Explanation:

celsius_to_fahrenheit(celsius) is a function that takes a temperature in Celsius and converts it to Fahrenheit using the formula (Celsius * 9/5) + 32.

map(celsius_to_fahrenheit, celsius_temperatures) applies the celsius_to_fahrenheit function to each element in the celsius_temperatures list.

The result is a map object, which we convert to a list and print.

Example Output:

For the input list [0, 20, 30, 40, 100] (Celsius temperatures), the output will be:

```
[32.0, 68.0, 86.0, 104.0, 212.0]
```

10. Create a Python program that uses `filter()` to remove all the vowels from a given string?

Ans = a Python program that uses the filter() function to remove all vowels from a given string:

```
# Function to check if a character is not a vowel

def is_not_vowel(char):

    vowels = 'aeiouAEIOU'

    return char not in vowels

# Input string
```

```
input_string = "Hello, World!"

# Using filter() to remove vowels

filtered_string = filter(is_not_vowel, input_string)

# Converting the result to a string and printing

result = ''.join(filtered_string)

print("String after removing vowels:", result)
```

Explanation:

The function is_not_vowel(char) returns True if the character is not a vowel (either lowercase or uppercase).

The filter(is_not_vowel, input_string) applies the is_not_vowel function to each character in the string, keeping only those characters that are not vowels.

The filter() function returns an iterator, so we use ''.join(filtered_string) to join the filtered characters back into a string.

Example Output:

For the input string "Hello, World!", the output will be:

```
String after removing vowels: Hll, Wrld!
```

11) Imagine an accounting routine used in a book shop. It works on a list with sublists, which look like this:

Ans =

Order Number	**Book Title and Author**	**Quantity**	**Price per Item**
34587	Learning Python, Mark Lutz	4	40.95
98762	Programming Python, Mark Lutz	5	56.80
77226	Head First Python, Paul Barry	3	32.95
88112	Einfuhrung in Python3, Bernd Klein	3	24.99

Write a Python program, which returns a list with 2-tuples. Each tuple consists of the order number and the product of the price per item and the quantity. The product should be increased by 10,- € if the value of the order is smaller than 100,00 €.

Write a Python program using lambda and map

Ans= orders = [["34587", "Learning Python, Mark Lutz", 4, 40.95],["98762", "Programming Python, Mark Lutz", 5, 56.80],["77226", "Head First Python, Paul Barry", 3,32.95],["88112", "Einführung in Python3, Bernd Klein", 3, 24.99]]

invoice_totals = list(map(lambda x: x if x[1] >= min_order else (x[0], x[1] + 10),map(lambda x: (x[0],x[2] * x[3]), orders)))

#Note- To understand the working of above lambda function break the function till innermost map function. Break and understand in below fashion

'''

output1 = map(lambda x: (x[0],x[2] * x[3]), orders) #Innermost lambda function execution

output2 = map(lambda x: x if x[1] >= min_order else (x[0], x[1] + 10),map(lambda x: (x[0],x[2] * x[3]), orders))

final = list(map(lambda x: x if x[1] >= min_order else (x[0], x[1] + 10),map(lambda x: (x[0],x[2] * x[3]), orders)))

'''

#Solution-2

```
from functools import reduce

orders = [ [1, ("5464", 4, 9.99), ("8274",18,12.99), ("9744", 9, 44.95)],[2, ("5464", 9, 9.99),
("9744", 9, 44.95)],[3, ("5464", 9, 9.99), ("88112", 11, 24.99)],[4, ("8732", 7, 11.99),
("7733",11,18.99), ("88112", 5, 39.95)] ]

invoice = list(map(lambda k: k if k[1]>=100 else (k[0],k[1]+10),map(lambda
x:(x[0],reduce(lambda a,b:a+b,list(map(lambda y:y[1]*y[2] ,x[1:])))),orders)))

#Breaking

'''
output1=map(lambda y:y[1]*y[2] ,x[1:])
output2=list(map(lambda y:y[1]*y[2] ,x[1:]))
output3=reduce(lambda a,b:a+b,list(map(lambda y:y[1]*y[2] ,x[1:])))
output4=map(lambda x:(x[0],reduce(lambda a,b:a+b,list(map(lambda y:y[1]*y[2] ,x[1:])))))
output5=map(lambda k: k if k[1]>=100 else (k[0],k[1]+10),map(lambda x:(x[0],reduce(lambda
a,b:a+b,list(map(lambda y:y[1]*y[2] ,x[1:])))),orders))
output6=list(map(lambda k: k if k[1]>=100 else (k[0],k[1]+10),map(lambda
x:(x[0],reduce(lambda a,b:a+b,list(map(lambda y:y[1]*y[2] ,x[1:])))),orders)))
```