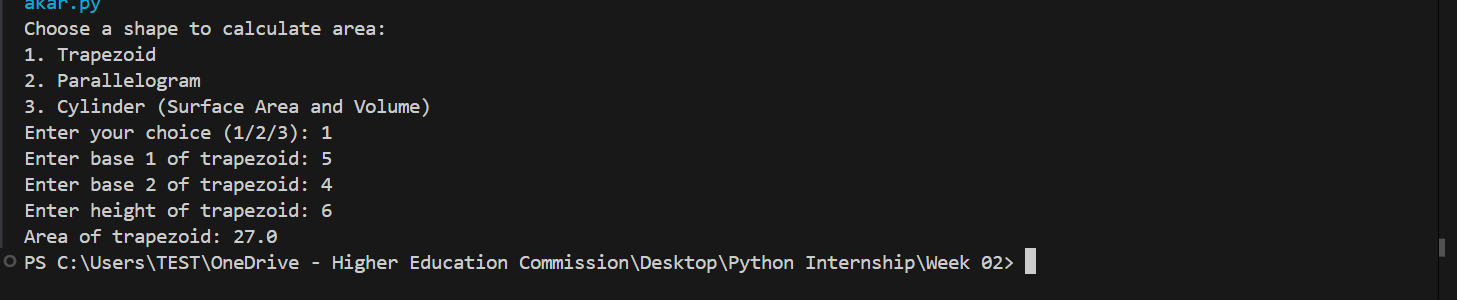
**Python Assignment 02 – Abu Bakar**

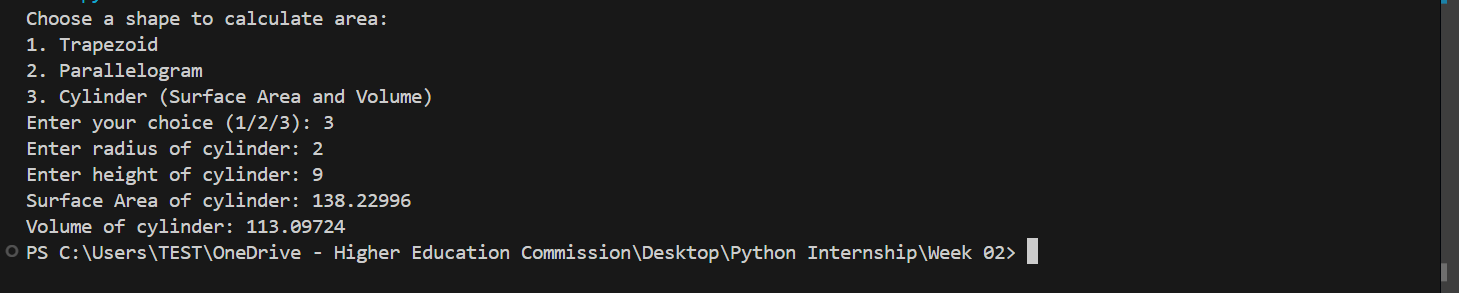
**TASK 01:**

**CODE**

|  |
| --- |
| def trapezoid\_area(base1, base2, height):      return 0.5 \* (base1 + base2) \* height  def parallelogram\_area(base, height):      return base \* height  def cylinder\_area\_volume(radius, height):      surface\_area = 2 \* 3.14159 \* radius \* (radius + height)      volume = 3.14159 \* radius\*\*2 \* height      return surface\_area, volume  if \_\_name\_\_ == "\_\_main\_\_":      print("Choose a shape to calculate area:")      print("1. Trapezoid")      print("2. Parallelogram")      print("3. Cylinder (Surface Area and Volume)")      choice = input("Enter your choice (1/2/3): ")      if choice == '1':          base1 = float(input("Enter base 1 of trapezoid: "))          base2 = float(input("Enter base 2 of trapezoid: "))          height = float(input("Enter height of trapezoid: "))          area = trapezoid\_area(base1, base2, height)          print(f"Area of trapezoid: {area}")      elif choice == '2':          base = float(input("Enter base of parallelogram: "))          height = float(input("Enter height of parallelogram: "))          area = parallelogram\_area(base, height)          print(f"Area of parallelogram: {area}")      elif choice == '3':          radius = float(input("Enter radius of cylinder: "))          height = float(input("Enter height of cylinder: "))          surface\_area, volume = cylinder\_area\_volume(radius, height)          print(f"Surface Area of cylinder: {surface\_area}")          print(f"Volume of cylinder: {volume}")      else:          print("Invalid choice") |

**RESULT**

****

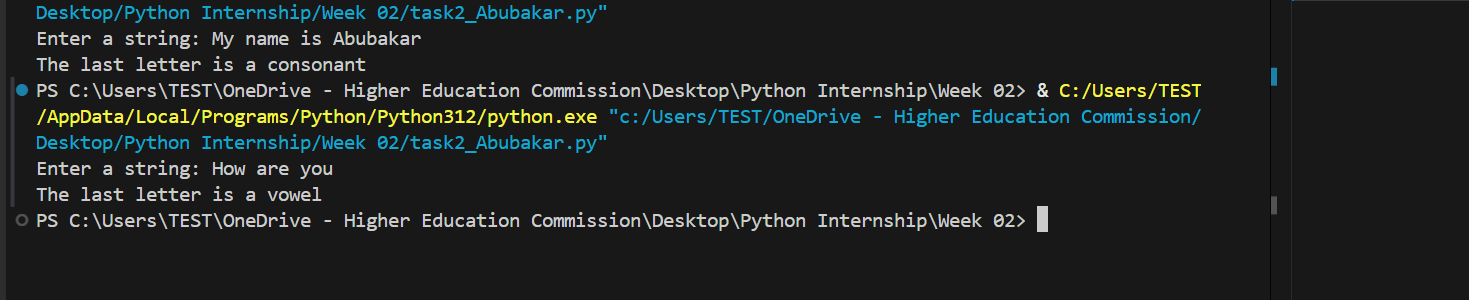
****

**TASK 02**

**CODE**

|  |
| --- |
| def check\_last\_letter\_vowel(input\_string):      vowels = "aeiouAEIOU"      if not input\_string:          return "Empty string provided"      last\_letter = input\_string[-1]      if last\_letter.isalpha():          if last\_letter in vowels:              return "vowel"          else:              return "consonant"      else:          return "Last character is not a letter"  if \_\_name\_\_ == "\_\_main\_\_":      user\_input = input("Enter a string: ")      result = check\_last\_letter\_vowel(user\_input)      print(f"The last letter is a {result}") |

**RESULT**

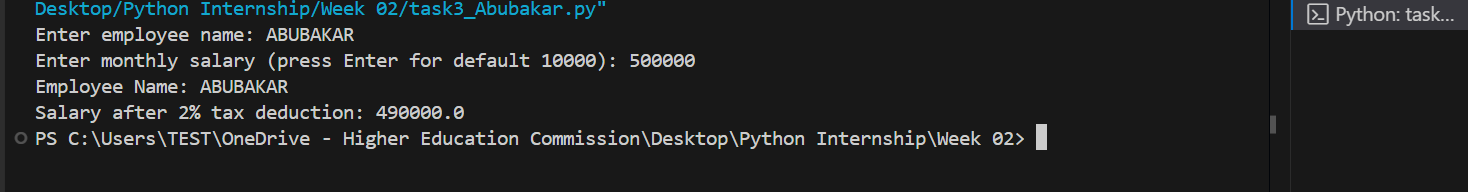
****

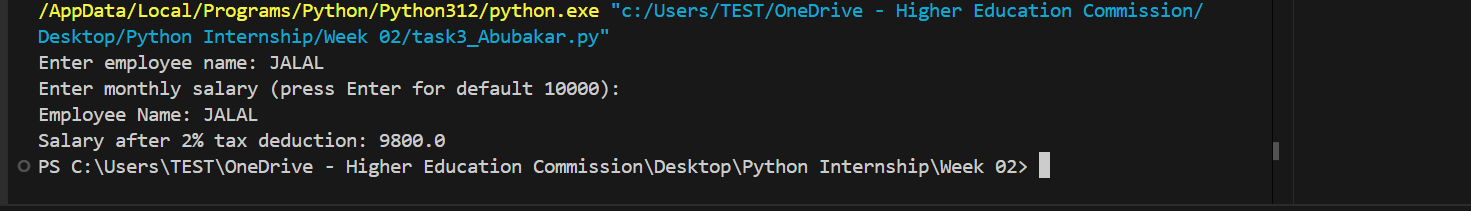
**TASK 03**

**CODE**

|  |
| --- |
| def employee\_salary(employee\_name, monthly\_salary=10000):      tax\_percentage = 2      tax\_amount = (tax\_percentage / 100) \* monthly\_salary      salary\_after\_tax = monthly\_salary - tax\_amount      print(f"Employee Name: {employee\_name}")      print(f"Salary after {tax\_percentage}% tax deduction: {salary\_after\_tax}")  if \_\_name\_\_ == "\_\_main\_\_":      name = input("Enter employee name: ")      salary\_input = input("Enter monthly salary (press Enter for default 10000): ")      if salary\_input:          salary = float(salary\_input)          employee\_salary(name, salary)      else:          employee\_salary(name) |

**RESULT**

****

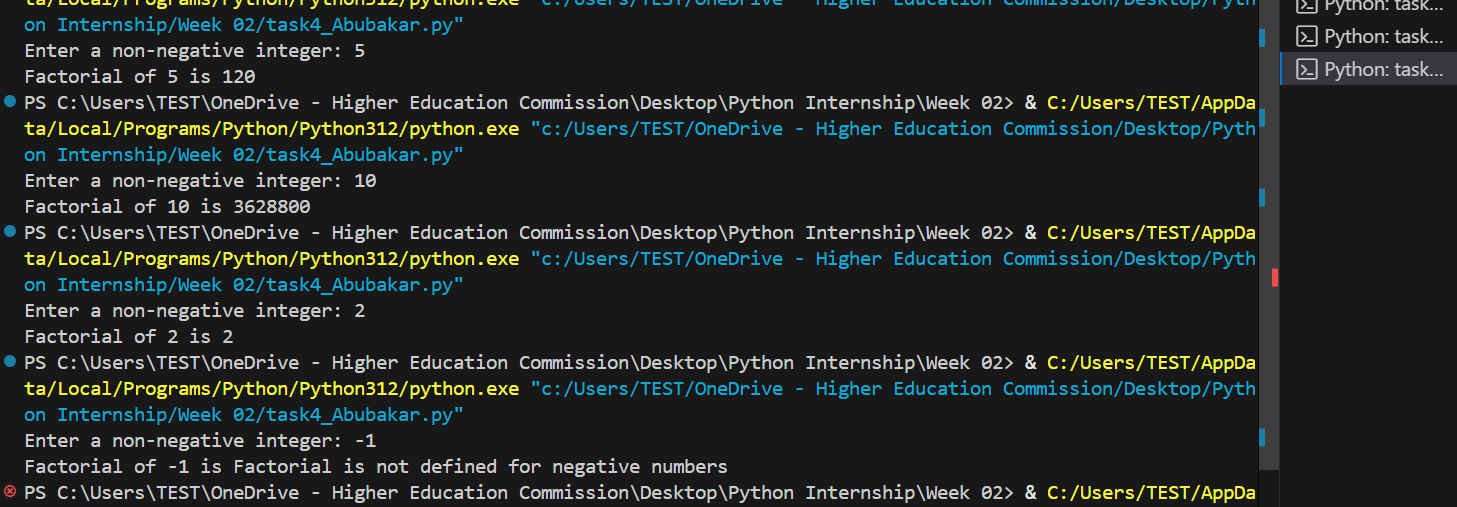
****

**TASK 04**

**CODE**

|  |
| --- |
| def calculate\_factorial(n):      if n < 0:          return "Factorial is not defined for negative numbers"      elif n == 0:          return 1      else:          factorial = 1          for i in range(1, n + 1):              factorial \*= i          return factorial  if \_\_name\_\_ == "\_\_main\_\_":      num = int(input("Enter a non-negative integer: "))      result = calculate\_factorial(num)      print(f"Factorial of {num} is {result}") |

**RESULT**

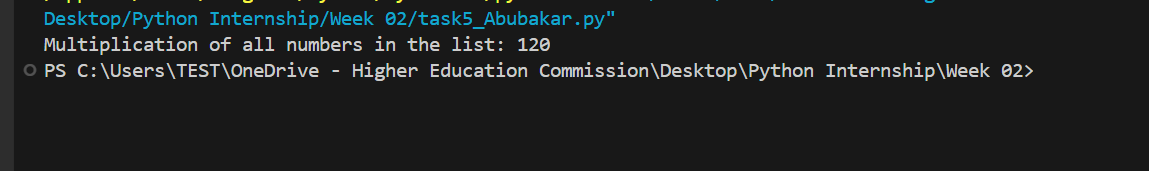
****

**TASK 05**

**CODE**

|  |
| --- |
| def multiply\_list\_numbers(numbers\_list):      if not numbers\_list:          return 1  # Return 1 for an empty list as the multiplicative identity      product = 1      for number in numbers\_list:          product \*= number      return product  if \_\_name\_\_ == "\_\_main\_\_":      numbers = [1, 2, 3, 4, 5]      result = multiply\_list\_numbers(numbers)      print(f"Multiplication of all numbers in the list: {result}") |

**RESULT**

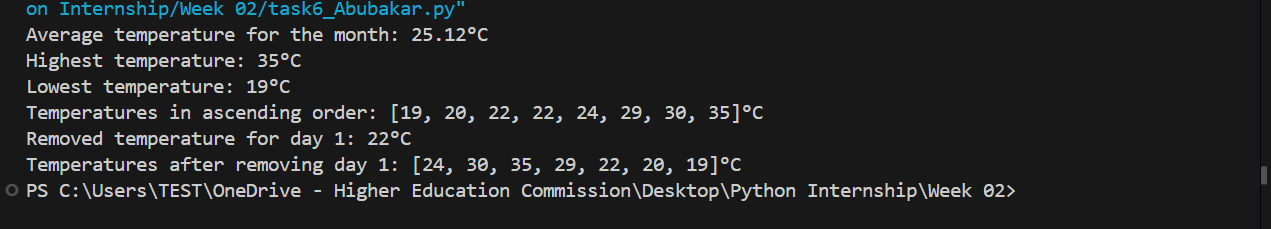
****

**TASK 06**

**CODE**

|  |
| --- |
| def calculate\_average\_temperature(temperatures):      if not temperatures:          return 0  # Return 0 if the list is empty to avoid division by zero      return sum(temperatures) / len(temperatures)  def find\_highest\_temperature(temperatures):      if not temperatures:          return None # Return None for an empty list      return max(temperatures)  def find\_lowest\_temperature(temperatures):      if not temperatures:          return None # Return None for an empty list      return min(temperatures)  def sort\_temperatures\_ascending(temperatures):      return sorted(temperatures)  def remove\_temperature\_by\_index(temperatures, index\_to\_remove):      if 0 <= index\_to\_remove < len(temperatures):          removed\_temperature = temperatures.pop(index\_to\_remove)          return removed\_temperature, temperatures      else:          return None, temperatures # Return None for removed temperature if index is invalid  if \_\_name\_\_ == "\_\_main\_\_":      temperatures\_celsius = [22, 24, 30, 35, 29, 22, 20, 19]      # Calculate and print the average temperature      average\_temp = calculate\_average\_temperature(temperatures\_celsius)      print(f"Average temperature for the month: {average\_temp:.2f}°C")      # Find and print the highest and lowest temperatures      highest\_temp = find\_highest\_temperature(temperatures\_celsius)      lowest\_temp = find\_lowest\_temperature(temperatures\_celsius)      print(f"Highest temperature: {highest\_temp}°C")      print(f"Lowest temperature: {lowest\_temp}°C")      # Sort the temperatures in ascending order      sorted\_temperatures = sort\_temperatures\_ascending(temperatures\_celsius)      print(f"Temperatures in ascending order: {sorted\_temperatures}°C")      # Remove the temperature record for a specific day (e.g., remove the first day's record)      day\_to\_remove\_index = 0 # Removing the first day      removed\_temperature, updated\_temperatures = remove\_temperature\_by\_index(temperatures\_celsius, day\_to\_remove\_index)      if removed\_temperature is not None:          print(f"Removed temperature for day {day\_to\_remove\_index + 1}: {removed\_temperature}°C")          print(f"Temperatures after removing day {day\_to\_remove\_index + 1}: {updated\_temperatures}°C")      else:          print("Invalid day index to remove.") |

**RESULT**

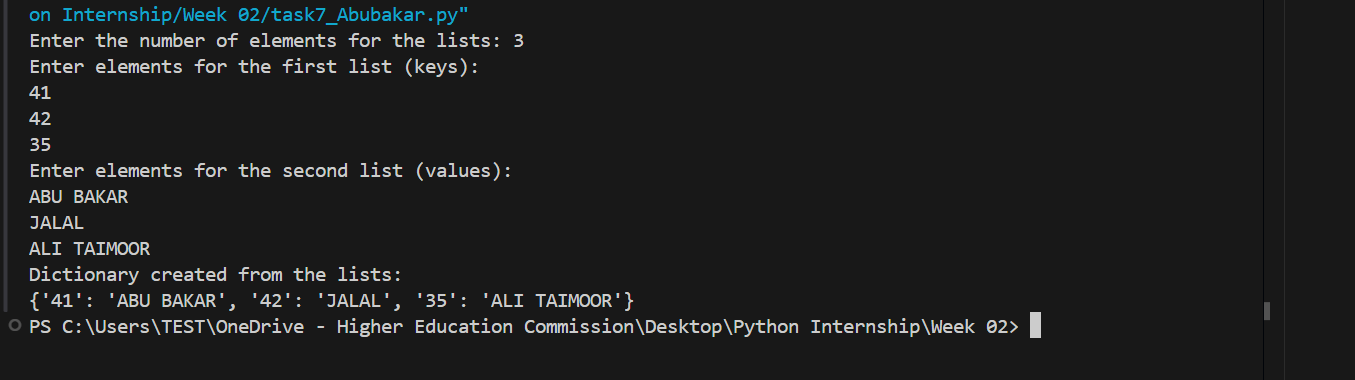
****

**TASK 07**

**CODE**

|  |
| --- |
| def list\_to\_dictionary(keys\_list, values\_list):      if len(keys\_list) != len(values\_list):          return "Error: Lists must have the same number of elements."      result\_dict = {}      for i in range(len(keys\_list)):          result\_dict[keys\_list[i]] = values\_list[i]      return result\_dict  if \_\_name\_\_ == "\_\_main\_\_":      keys = []      values = []      n = int(input("Enter the number of elements for the lists: "))      print("Enter elements for the first list (keys):")      for \_ in range(n):          keys.append(input())      print("Enter elements for the second list (values):")      for \_ in range(n):          values.append(input())      dictionary = list\_to\_dictionary(keys, values)      if isinstance(dictionary, str): # Check if an error message string was returned          print(dictionary)      else:          print("Dictionary created from the lists:")          print(dictionary) |

**RESULT**

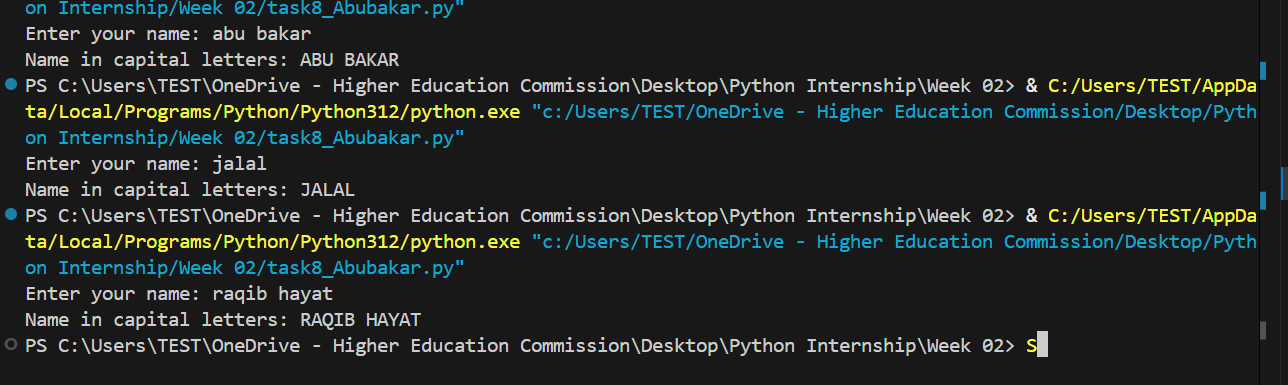
****

**TASK 08**

**CODE**

|  |
| --- |
| def Capital\_Convertor():      name = input("Enter your name: ")      return name.upper()  if \_\_name\_\_ == "\_\_main\_\_":      capital\_name = Capital\_Convertor()      print(f"Name in capital letters: {capital\_name}") |

**RESULT**

****

**Task 9: Why do we use function in Python?**

**Explanation:**

Functions are essential in Python (and programming in general) for several reasons. They provide significant advantages in terms of code organization, reusability, and maintainability. Here are three key advantages:

1. **Code Reusability:** Functions allow you to write a block of code once and reuse it multiple times throughout your program. Instead of repeating the same code logic in different parts of your program, you can encapsulate it within a function and call that function whenever you need to perform that specific task.

**Example:**

|  |
| --- |
| def greet(name):  print(f"Hello, {name}!")  greet("Alice") # Reusing the greet function  greet("Bob") # Reusing it again |

1. **Code Modularity and Organization:** Functions help break down complex programs into smaller, more manageable, and logical modules. Each function can be designed to perform a specific task, making the overall program structure clearer and easier to understand. This modularity makes it simpler to develop, debug, and maintain larger programs.

**Example:**

Imagine a program for managing a library. You could have separate functions for:

* + add\_book(book\_title, author)
  + borrow\_book(book\_title, member\_id)
  + return\_book(book\_title, member\_id)
  + search\_book(book\_title)

Each function handles a specific part of the library management system, making the code organized and easier to work with.

1. **Code Readability and Maintainability:** By using functions, you can make your code more readable and self-explanatory. Function names act as labels that describe what a particular block of code does. This improves code readability and makes it easier for others (and your future self) to understand the program's logic. When you need to make changes or fix bugs, well-structured code with functions is much easier to maintain because changes are often localized to specific functions rather than spread throughout the entire program.

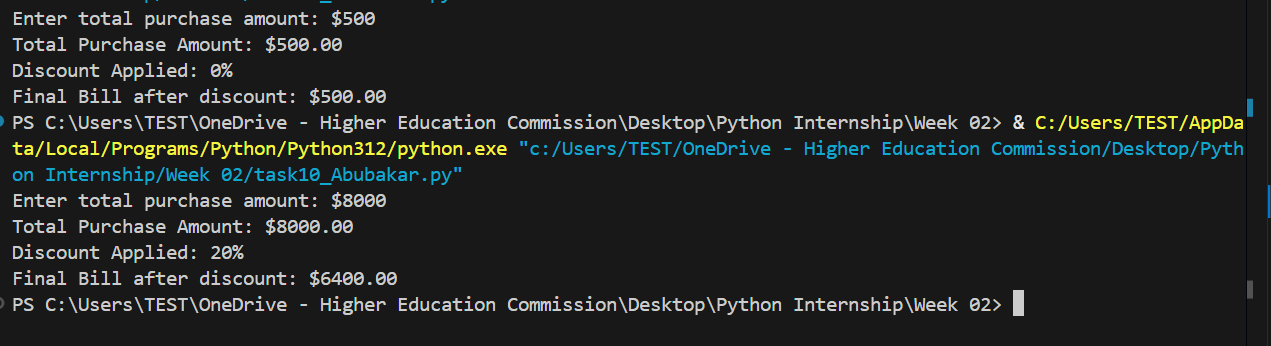
**Example:**

Consider calculating the area of different shapes. Using functions like calculate\_rectangle\_area(), calculate\_circle\_area(), etc., makes the code more readable than writing the area calculation formulas directly in the main part of the program. If you need to update the formula for circle area, you only need to modify the calculate\_circle\_area() function.

**TASK 10**

**CODE**

|  |
| --- |
| def calculate\_discounted\_bill(total\_purchase\_amount):      discount\_percentage = 0      if total\_purchase\_amount > 1000:          discount\_percentage = 20      elif total\_purchase\_amount > 500:          discount\_percentage = 10      discount\_amount = (discount\_percentage / 100) \* total\_purchase\_amount      final\_bill = total\_purchase\_amount - discount\_amount      return final\_bill, discount\_percentage  if \_\_name\_\_ == "\_\_main\_\_":      purchase\_amount = float(input("Enter total purchase amount: $"))      final\_bill, discount = calculate\_discounted\_bill(purchase\_amount)      print(f"Total Purchase Amount: ${purchase\_amount:.2f}")      print(f"Discount Applied: {discount}%")      print(f"Final Bill after discount: ${final\_bill:.2f}") |

**RESULT** ****

**Explanation for function efficiency:**

Using a function like calculate\_discounted\_bill makes the code more efficient because:

1. **Reusability:** The discount logic is encapsulated within the function. If we need to calculate discounts at multiple points in our program (e.g., for different customers or items), we can simply call this function again and again without rewriting the discount calculation logic.

2. **Modularity:** The function breaks down the larger task of bill calculation into a smaller, manageable, and logical unit (discount calculation). This makes the code easier to understand, debug, and maintain.

3. **Readability:** The function gives a clear name to the discount calculation process, making the main part of the code more readable. Instead of seeing complex discount calculation logic inline, we see a function call `calculate\_discounted\_bill`, which clearly conveys the purpose.

4. **Testability:** Functions are easier to test in isolation. We can write unit tests specifically for the `calculate\_discounted\_bill` function to ensure it correctly applies discounts for different purchase amounts, without having to test the entire shopping platform code.