

AI ASSISTED CODING LAB TEST 04

K . VIKAS
2403A52126

SET - B

Question 1: Refactor Nested Conditional Statements

Task:

You are given the following legacy code that determines discounts based on customer type and purchase amount:

```
def calculate_discount(customer_type, amount):
    if customer_type == "regular":
        if amount > 5000:
            discount = amount * 0.05
        else:
            discount = 0
    else:
        if amount > 3000:
            discount = amount * 0.10
        else:
            discount = amount * 0.05
    return amount - discount
```

Your task:

Use an AI-assisted code tool to refactor the function by:

1. Simplifying nested if-else statements for better readability.
2. Introducing cleaner logic using dictionaries or guard clauses.
3. Adding meaningful docstrings and inline comments.

Instructions:

- Verify the correctness by testing multiple input cases.
- List the AI's key refactoring suggestions and explain how they improved maintainability.

GENERATED REFACTORED CODE WITH OUTPUT :

```
def calculate_discount(customer_type, amount):
    """
    Calculates the discounted amount based on customer type and purchase amount.

    Args:
        customer_type (str): The type of customer ("regular" or other).
        amount (float): The total purchase amount.

    Returns:
        float: The discounted amount.
    """
    # Define discounts based on customer type and amount thresholds
    discounts = {
        "regular": {
            'threshold': 5000,
            'rate': 0.05
        },
        "other": {
            'threshold': 3000,
            'rate': 0.10,
            'default_rate': 0.05
        }
    }

    # Get the discount information for the customer type
    customer_discounts = discounts.get(customer_type, discounts["other"])

    # Calculate discount based on amount and thresholds
    if amount > customer_discounts['threshold']:
        discount = amount * customer_discounts['rate']
    else:
        # Apply default rate for "other" customer type if amount is below threshold
        discount = amount * customer_discounts.get('default_rate', 0)
```

```
discount = amount * customer_discounts.get('default_rate', 0)

return amount - discount

# Test cases
print(f"Regular customer, amount 6000: {calculate_discount('regular', 6000)}")
print(f"Regular customer, amount 4000: {calculate_discount('regular', 4000)}")
print(f"Other customer, amount 4000: {calculate_discount('other', 4000)}")
print(f"Other customer, amount 2000: {calculate_discount('other', 2000)}")
```

```
Regular customer, amount 6000: 5700.0
Regular customer, amount 4000: 4000
Other customer, amount 4000: 3600.0
Other customer, amount 2000: 1900.0
```

EXPLANATION OF THE CODE :

- `def calculate_discount(customer_type, amount):` : This line defines the function `calculate_discount` that takes two arguments: `customer_type` (a string) and `amount` (a float).
- `discounts = { ... }` : This dictionary stores the discount rates and thresholds for different customer types ("regular" and "other").
- `customer_discounts = discounts.get(customer_type, discounts["other"])` : This line retrieves the discount information for the given `customer_type`. If the `customer_type` is not found in the `discounts` dictionary, it defaults to the "other" customer type's discount information.
- `if amount > customer_discounts['threshold']:` : This checks if the purchase amount is greater than the threshold defined for the customer type.
- `discount = amount * customer_discounts['rate']` : If the amount is greater than the threshold, the discount is calculated by multiplying the amount by the specified rate.
- `else:` : If the amount is not greater than the threshold, the code enters this block.
- `discount = amount * customer_discounts.get('default_rate', 0)` : This line calculates the discount for amounts below the threshold. For "other" customers, it uses the `default_rate` if it exists; otherwise, the discount is 0.
- `return amount - discount` : The function returns the final discounted amount by subtracting the calculated discount from the original amount.

Question 2: Refactor Repetitive Code Using Functions

Task:

The following script calculates total sales for different product categories but repeats logic unnecessarily:

```
electronics = [1200, 1500, 1000, 800]
clothing = [500, 700, 600]
groceries = [200, 150, 180, 220]
total_electronics = 0
for e in electronics:
    total_electronics += e
total_clothing = 0
for c in clothing:
    total_clothing += c
total_groceries = 0
for g in groceries:
    total_groceries += g
print("Total Electronics:", total_electronics)
print("Total Clothing:", total_clothing)
print("Total Groceries:", total_groceries)
```

Your task:

Use AI-assisted refactoring to:

1. Remove repetitive code by creating reusable functions or loops.
2. Make the script scalable for any number of categories.
3. Apply modern Python practices (e.g., `sum()`, dictionary iteration).

Instructions:

- Compare old and new code performance and readability.
- Document which AI-generated improvements you used.

GENERATED REFACTORED CODE WITH OUTPUT :

```
▶ def calculate_total_sales(sales_data):
    """
    Calculate total sales for multiple product categories.

    Parameters:
        sales_data (dict): A dictionary where keys are category names and
                           values are lists of sales amounts.

    Returns:
        dict: A dictionary containing total sales for each category.
    """
    # Use dictionary comprehension with built-in sum() for concise aggregation
    return {category: sum(amounts) for category, amounts in sales_data.items()}

    # Sales data for different categories
    sales_data = {
        "Electronics": [1200, 1500, 1000, 800],
        "Clothing": [500, 700, 600],
        "Groceries": [200, 150, 180, 220]
    }

    # Calculate totals using reusable function
    totals = calculate_total_sales(sales_data)

    # Display results
    for category, total in totals.items():
        print(f"Total {category}: {total}")

...
Total Electronics: 4500
Total Clothing: 1800
Total Groceries: 750
```

EXPLANATION OF THE CODE :

- `def calculate_total_sales(sales_data):`: This line defines the function `calculate_total_sales` that takes one argument: `sales_data` (a dictionary).
- `return {category: sum(amounts) for category, amounts in sales_data.items()}`: This line uses a dictionary comprehension to efficiently calculate the total sales for each category. It iterates through the `sales_data` dictionary, where each `category` is a key and `amounts` is a list of sales figures for that category. For each category, it uses the built-in `sum()` function to add up all the sales amounts in the list and creates a new dictionary where the keys are the categories and the values are the calculated total sales.
- `sales_data = { ... }`: This dictionary contains sample sales data for different categories: "Electronics", "Clothing", and "Groceries". Each category has a list of sales amounts associated with it.
- `totals = calculate_total_sales(sales_data)`: This line calls the `calculate_total_sales` function with the `sales_data` and stores the returned dictionary of total sales in the `totals` variable.
- `for category, total in totals.items():`: This loop iterates through the `totals` dictionary.
- `print(f"Total {category}: {total}")`: This line prints the total sales for each category in a formatted string.