Name:Swetha Dixit

Batch-33

Hall-Ticket:2303A52415

Course:Ai coding

Assignment:13.5

Task1:
Lab 13: Code Refactoring – Improving Legacy Code with AI Suggestions
Lab Objectives:
Week7 -
Friday
• Identify code smells and inefficiencies in legacy Python scripts.
• Use AI-assisted coding tools to refactor for readability, maintainability, and performance.
• Apply modern Python best practices while ensuring output correctness.
Task Description #1 (Refactoring – Removing Global Variables)
• Task: Use AI to eliminate unnecessary global variables from the code.
• Instructions:
o Identify global variables used across functions.
o Refactor the code to pass values using function parameters.
o Improve modularity and testability.
• Sample Legacy Code:

```
rate = 0.1
def calculate_interest(amount):
```

return amount * rate
print(calculate_interest(1000))

• Expected Output:

o Refactored version passing rate as a parameter or using a configuration structure.

Prompt: > "Refactor this Python code to eliminate the global variable `rate`. Update the `calculate_interest` function so that `rate` is passed as a parameter. Apply modern Python best practices (like type hinting and a docstring) to improve readability, modularity, and testability. Ensure the final print statement outputs the correct result."

```
def calculate_interest(principal: float, rate: float, time: float) ->
float:
    """
    Calculate the simple interest given principal, rate, and time.



    Args:
        principal: The initial amount of money invested or loaned.
        rate: The annual interest rate as a decimal (e.g. 0.05 for 5%).
        time: The time period in years for which the interest is
calculated.

    Returns:
        The interest accrued over the given time period.
    """
    return principal * rate * time


def main() -> None:
    """Run a simple interest calculation example."""
    principal: float = 1000.0
    rate: float = 0.05    # 5% annual interest, passed explicitly
    time: float = 2.0     # 2 years

    interest: float = calculate_interest(principal, rate, time)
    print(interest)
```

```python
if __name__ == "__main__":
    main()
```

Task Description #2 : (Refactoring Deeply Nested Conditionals)

• Task: Use AI to refactor deeply nested if–elif–else logic into a cleaner structure.

• Focus Areas:

o Readability

o Logical simplification

o Maintainability

Legacy Code:

score = 78

if score >= 90:

print("Excellent")

else:

if score >= 75:

print("Very Good")

else:

if score >= 60:

print("Good")

else:

print("Needs Improvement")

Expected Outcome:

o Flattened logic using guard clauses or a mapping-based approach.

> **Prompt:** "Refactor this deeply nested Python conditional logic into a cleaner structure. First, wrap the logic in a function so you can use **guard clauses** (early returns) to flatten the `if-else` chain. Then, provide a second version using a **mapping-based approach** (like iterating through a

list of tuples) to determine the output. Ensure both versions include type hints and a docstring for maintainability."

```
from typing import Callable, List, Literal, Tuple


Outcome = Literal["low", "medium", "high", "invalid"]
Rule = Tuple[Callable[[int], bool], Outcome]


def classify_score_guard(score: int) -> Outcome:
    """
    Classify a numeric score into a category using guard clauses.

    Args:
        score: The numeric score to classify.

    Returns:
        A string label describing the score category.
    """
    if score < 0:
        return "invalid"
    if score < 50:
        return "low"
    if score < 80:
        return "medium"
    if score <= 100:
        return "high"

    return "invalid"


def classify_score_mapping(score: int) -> Outcome:
    """
    Classify a numeric score into a category using a list of rules.

    This demonstrates a mapping-based (data-driven) approach where each
rule
    consists of a condition function and the label it produces.
```

```
    Args:
        score: The numeric score to classify.

    Returns:
        A string label describing the score category.
    """
    rules: List[Rule] = [
        (lambda s: s < 0, "invalid"),
        (lambda s: 0 <= s < 50, "low"),
        (lambda s: 50 <= s < 80, "medium"),
        (lambda s: 80 <= s <= 100, "high"),
    ]

    for condition, label in rules:
        if condition(score):
            return label

    return "invalid"


def main() -> None:
    """Simple demo of both classification strategies."""
    test_score: int = 72

    print("Guard:", classify_score_guard(test_score))
    print("Mapping:", classify_score_mapping(test_score))


if __name__ == "__main__":
    main()
```

Task 3 (Refactoring Repeated File Handling Code)

• Task: Use AI to refactor repeated file open/read/close logic.

• Focus Areas:

o DRY principle

o Context managers

o Function reuse

Legacy Code:
f = open("data1.txt")
print(f.read())
f.close()
f = open("data2.txt")
print(f.read())
f.close()
Expected Outcome:
o Reusable function using with open() and parameters.
Prompt: "Refactor this repeated Python file handling logic to adhere strictly to the DRY (Don't Repeat Yourself) principle. Create a reusable function that accepts a filename as a parameter. Inside the function, use a context manager (`with open(...)`) to safely open, read, and print the file's contents. Add type hints, a docstring, and a simple `try-except` block to handle `FileNotFoundError` gracefully."

```
from pathlib import Path

def print_file_contents(filename: str) -> None:
    """
    Open a text file and print its contents safely.

    Uses a context manager to ensure the file is closed properly and
handles
    missing files gracefully.

    Args:
        filename: Path to the file whose contents should be printed.
    """
    try:
        path = Path(filename)
        with path.open("r", encoding="utf-8") as file:
            print(file.read())
    except FileNotFoundError:
        print(f"File not found: {filename}")
```

```python
def main() -> None:
    """Example usage of the reusable file printing function."""
    # Replace these with actual filenames you want to test with.
    print_file_contents("example1.txt")
    print_file_contents("example2.txt")


if __name__ == "__main__":
    main()
```

Task 4 (Optimizing Search Logic)
• Task: Refactor inefficient linear searches using appropriate data structures.
• Focus Areas:
o Time complexity
o Data structure choice
Legacy Code:
users = ["admin", "guest", "editor", "viewer"]
name = input("Enter username: ")
found = False
for u in users:
if u == name:
found = True
print("Access Granted" if found else "Access Denied")
Expected Outcome:
o Use of sets or dictionaries with complexity justification.

Prompt: "Refactor this Python search logic to optimize time complexity. Replace the linear list iteration with a set to achieve $O(1)$ average lookup time. Wrap the logic in a function with type hints, and include comments explicitly justifying the time complexity improvement (from $O(n)$ to $O(1)$) and the choice of data structure."

```python
from typing import Hashable, Iterable
```

```python
def contains_value_optimized(values: Iterable[Hashable], target: Hashable)
-> bool:
    """
    Check whether a target value exists in a collection using an optimized
lookup.

    This refactored version converts the input collection to a set once
and then
    performs membership checks against the set.
    """
    # Converting the iterable to a set is O(n) for n input elements, but
    # subsequent membership checks are O(1) on average. This is a strict
    # improvement over scanning a list linearly, which would be O(n) per
lookup.
    lookup = set(values)

    # Using a set here is preferable because it is implemented as a hash
table,
    # giving average O(1) time complexity for membership tests like
`target in lookup`.
    return target in lookup


def main() -> None:
    """Simple demonstration of the optimized search logic."""
    items = ["apple", "banana", "cherry"]
    print(contains_value_optimized(items, "banana"))  # Expected: True
    print(contains_value_optimized(items, "orange"))  # Expected: False


if __name__ == "__main__":
    main()
```

Task 5 (Refactoring Procedural Code into OOP Design)

• Task: Use AI to refactor procedural code into a class-based design.

• Focus Areas:

o Object-Oriented principles
o Encapsulation
Legacy Code:
salary = 50000
tax = salary * 0.2
net = salary - tax
print(net)
Expected Outcome:
o A class like EmployeeSalaryCalculator with methods and attributes.
Prompt: "Refactor this procedural Python code into an Object-Oriented design. Create a class named `EmployeeSalaryCalculator`. Use the `__init__` method to encapsulate the `salary` and `tax_rate` as instance attributes. Create a method called `calculate_net_salary` to compute and return the net pay. Include type hints, a docstring, and demonstrate how to instantiate the class and print the result."

```python
class EmployeeSalaryCalculator:
    """
    Calculate an employee's net salary given a gross salary and tax rate.

    The tax rate is expressed as a decimal (e.g. 0.22 for 22%).
    """

    def __init__(self, salary: float, tax_rate: float) -> None:
        """
        Initialize the salary calculator with salary and tax rate.

        Args:
            salary: The gross salary amount.
            tax_rate: The tax rate as a decimal (e.g. 0.22 for 22%).
        """
        self.salary: float = salary
        self.tax_rate: float = tax_rate

    def calculate_net_salary(self) -> float:
```

```python
        """
        Compute the net salary after tax.

        Returns:
            The net salary after applying the tax rate.
        """
        tax_amount = self.salary * self.tax_rate
        return self.salary - tax_amount


def main() -> None:
    """Demonstrate using EmployeeSalaryCalculator to compute net
salary."""
    gross_salary: float = 5000.0
    tax_rate: float = 0.22

    calculator = EmployeeSalaryCalculator(gross_salary, tax_rate)
    net_salary: float = calculator.calculate_net_salary()
    print(net_salary)


if __name__ == "__main__":
    main()
```

Task 6 (Refactoring for Performance Optimization)

• Task: Use AI to refactor a performance-heavy loop handling large data.

• Focus Areas:

o Algorithmic optimization

o Use of built-in functions

Legacy Code:

total = 0

for i in range(1, 1000000):

if i % 2 == 0:

total += i

print(total)

Expected Outcome:
o Optimized logic using mathematical formulas or comprehensions, with time comparison.

Prompt: "Refactor this Python loop to optimize performance. Provide two optimized solutions: one using Python's built-in sum() function with an optimized range(), and another using the $O(1)$ mathematical formula for the sum of an arithmetic progression. Use the time module to run and print a time comparison between the legacy code and both optimized versions. Include type hints, docstrings, and comments explaining the time complexities."

```python
from time import perf_counter


def legacy_sum(n: int) -> int:
    """
    Legacy implementation that sums numbers from 1 to n using an explicit
loop.

    Time complexity:
        O(n) - the loop performs n additions.
    """
    total = 0
    for i in range(1, n + 1):
        total += i
    return total


def optimized_sum_builtin(n: int) -> int:
    """
    Optimized implementation using Python's built-in sum() with range().

    Time complexity:
        O(n) - sum() still iterates over the range internally, but it is
        implemented in C and is typically faster than a pure Python loop.
    """
    return sum(range(1, n + 1))


def optimized_sum_formula(n: int) -> int:
```

```
    """
    Optimized implementation using the O(1) arithmetic progression
formula.

    Uses the formula for the sum of the first n natural numbers:
        n * (n + 1) / 2

    Time complexity:
        O(1) - a constant number of arithmetic operations regardless of n.
    """
    return n * (n + 1) // 2


def time_function(fn, n: int) -> float:
    """
    Measure the execution time of a summation function for a given n.

    Args:
        fn: The summation function to time.
        n:  The upper bound of the summation.

    Returns:
        The elapsed time in seconds.
    """
    start = perf_counter()
    result = fn(n)
    end = perf_counter()

    # Ensure the result is used so the call is not optimized away.
    _ = result
    return end - start


def main() -> None:
    """Run and print a time comparison for legacy and optimized
approaches."""
    n: int = 10_000_000

    legacy_time = time_function(legacy_sum, n)
    builtin_time = time_function(optimized_sum_builtin, n)
```

```python
    formula_time = time_function(optimized_sum_formula, n)

    print(f"n = {n}")
    print(f"Legacy loop (O(n)) time:        {legacy_time:.6f} seconds")
    print(f"Built-in sum with range (O(n)): {builtin_time:.6f} seconds")
    print(f"Formula (O(1)) time:            {formula_time:.6f} seconds")


if __name__ == "__main__":
    main()
```

Task 7 (Removing Hidden Side Effects)

• Task: Refactor code that modifies shared mutable state.

• Focus Areas:

o Functional-style refactoring

o Predictability

Legacy Code:

data = []

def add_item(x):

data.append(x)

add_item(10)

add_item(20)

print(data)

Expected Outcome:

o Refactored function returning values instead of mutating globals.

Prompt: "Refactor this Python code to remove the hidden side effect of mutating a global list. Adopt a functional-style approach by creating a pure function `add_item` that takes the current list and the new item as parameters. It should return a brand-new list with the item appended, rather than mutating the original list. Include type hints and a docstring."

```python
from typing import List, TypeVar
```

```
T = TypeVar("T")


def add_item(items: List[T], new_item: T) -> List[T]:
    """
    Return a new list with `new_item` appended, without mutating the
original.

    This function follows a functional style by avoiding side effects: it
does
    not modify the input list, but instead creates and returns a new list.

    Args:
        items: The original list of items.
        new_item: The item to append to the list.

    Returns:
        A brand-new list containing all elements of `items` plus
`new_item`.
    """
    return [*items, new_item]


def main() -> None:
    """Demonstrate using add_item without mutating the original list."""
    original: List[int] = [1, 2, 3]
    updated: List[int] = add_item(original, 4)

    print(f"Original list: {original}")  # [1, 2, 3]
    print(f"Updated list:  {updated}")   # [1, 2, 3, 4]


if __name__ == "__main__":
    main()
```

Task 8 (Refactoring Complex Input Validation Logic)

• Task: Use AI to simplify and modularize complex validation rules.

• Focus Areas:
o Readability
o Testability
Legacy Code:
password = input("Enter password: ")
if len(password) >= 8:
if any(c.isdigit() for c in password):
if any(c.isupper() for c in password):
print("Valid Password")
else:
print("Must contain uppercase")
else:
print("Must contain digit")
else:
print("Password too short")
Expected Outcome:
o Separate validation functions with clear responsibility
Prompt: "Refactor this deeply nested password validation logic to improve readability and testability. Extract each validation rule (length, digit, uppercase) into its own separate, single-responsibility helper function that returns a boolean. Then, create a main `validate_password` function that uses these helpers to accumulate and return a list of all error messages. Include type hints, docstrings, and an execution block that prints 'Valid Password' or the list from typing import List

```
def is_valid_length(password: str, min_length: int = 8) -> bool:
    """
    Check whether the password meets the minimum length requirement.

    Args:
        password: The password string to validate.
        min_length: The minimum allowed length for the password.
```

```
    Returns:
        True if the password length is at least min_length, otherwise
False.
    """
    return len(password) >= min_length


def has_digit(password: str) -> bool:
    """
    Check whether the password contains at least one numeric digit.

    Args:
        password: The password string to validate.

    Returns:
        True if the password contains a digit, otherwise False.
    """
    return any(char.isdigit() for char in password)


def has_uppercase(password: str) -> bool:
    """
    Check whether the password contains at least one uppercase letter.

    Args:
        password: The password string to validate.

    Returns:
        True if the password contains an uppercase letter, otherwise
False.
    """
    return any(char.isupper() for char in password)


def validate_password(password: str) -> List[str]:
    """
    Validate a password against multiple rules and collect error messages.

    The validation rules are:
```

```python
      - Minimum length of 8 characters
      - Contains at least one digit
      - Contains at least one uppercase letter

    Args:
        password: The password string to validate.

    Returns:
        A list of error messages. An empty list indicates a valid password.
    """
    errors: List[str] = []

    if not is_valid_length(password):
        errors.append("Password must be at least 8 characters long.")
    if not has_digit(password):
        errors.append("Password must contain at least one digit.")
    if not has_uppercase(password):
        errors.append("Password must contain at least one uppercase letter.")

    return errors


def main() -> None:
    """Run a simple password validation example and print the result."""
    test_password = "Passw0rd"  # Adjust to experiment with different inputs
    validation_errors = validate_password(test_password)

    if not validation_errors:
        print("Valid Password")
    else:
        print(validation_errors)


if __name__ == "__main__":
    main()
```

Output:

```
PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\Activate.ps1"
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_1_13.5.py"
100.0
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_2_13.5.py"
Guard: medium
Mapping: medium
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_3_13.5.py"
File not found: example1.txt
File not found: example2.txt
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_4_13.5.py"
True
False
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_5_13.5.py"
3900.0
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_6_13.5.py"
n = 10000000
Legacy loop (O(n)) time:        0.344928 seconds
Built-in sum with range (O(n)): 0.157139 seconds
Formula (O(1)) time:            0.000004 seconds
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_7_13.5.py"
Original list: [1, 2, 3]
Updated list:  [1, 2, 3, 4]
(.venv) PS C:\Users\intec\Ai coding> & "C:\Users\intec\Ai coding\.venv\Scripts\python.exe" "c:/Users
/intec/Ai coding/Assignment 4.2/Task_8_13.5.py"
Valid Password
```

powersh...
powershell
Ctrl+K to generate command
Ai coding 0 0