

AI Assisted Coding – Lab 13.5

Code Refactoring – Improving Legacy Code with AI

Student & Course Details

Field	Information
Course Code	23CS002PC304
Course Title	AI Assisted Coding
Student Name	Ravula seetharam reddy
Roll Number	2303A52440
Batch	34
Program	B.Tech – CSE (AI & ML)
Year/Sem	3rd Year – 2nd Semester
Regulation	R23

Lab Objectives

-  Identify code smells and inefficiencies in legacy Python scripts.
 -  Use AI-assisted coding tools to refactor for readability and performance.
 -  Apply modern Python best practices while ensuring output correctness.
-

Task 1: Refactoring – Removing Global Variables

Objective

To eliminate global variables and improve modularity by passing required values as parameters.

Refactored Code

```
def calculate_interest(amount, rate):
    return amount * rate

print(calculate_interest(1000, 0.1))
```

Explanation

Global dependency removed. Function is now reusable, modular, and testable.

Screenshots



The screenshot shows a code editor interface with a file named 'main.py' on the left and its output on the right. The code defines a function 'calculate_interest' that takes 'amount' and 'rate' as parameters and returns their product. It then calls this function with arguments 1000 and 0.1, resulting in an output of 100.0.

```
main.py + 44etrw7rd ⚡
1 def calculate_interest(amount, rate):
2     return amount * rate
3
4 print(calculate_interest(1000, 0.1))
5
6
7
8
```

Output:
100.0

Task 2: Refactoring Deeply Nested Conditionals

Refactored Code

```
score = 78

if score >= 90:
    print("Excellent")
elif score >= 75:
    print("Very Good")
elif score >= 60:
    print("Good")
else:
    print("Needs Improvement")
```

Explanation

Flattened nested conditions using `elif` for better readability.

Screenshots

≡ <> OneCompiler



The screenshot shows a Python code editor with a file named `main.py`. The code contains a series of nested if statements to determine a grade based on a score. The output window shows the result "Very Good".

```
main.py + 44etrw7rd ⚡
1 score = 78
2
3 if score >= 90:
4     print("Excellent")
5 elif score >= 75:
6     print("Very Good")
7 elif score >= 60:
8     print("Good")
9 else:
10    print("Needs Improvement")
11
12
13
14
15
```

Output:
Very Good

Task 3: Refactoring Repeated File Handling Code

Refactored Code

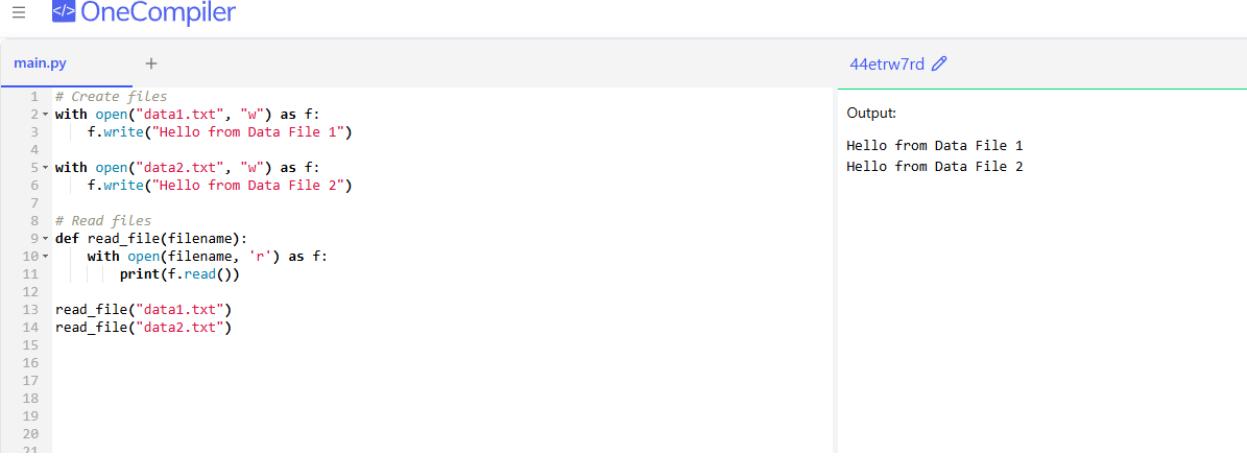
```
def read_file(filename):
    with open(filename, 'r') as f:
        print(f.read())

read_file("data1.txt")
read_file("data2.txt")
```

Explanation

Used reusable function and context manager (`with`) to follow DRY principle.

Screenshots



A screenshot of the OneCompiler interface. On the left, the code editor shows a file named 'main.py' with the following content:

```
1 # Create files
2 with open("data1.txt", "w") as f:
3     f.write("Hello from Data File 1")
4
5 with open("data2.txt", "w") as f:
6     f.write("Hello from Data File 2")
7
8 # Read files
9 def read_file(filename):
10     with open(filename, 'r') as f:
11         print(f.read())
12
13 read_file("data1.txt")
14 read_file("data2.txt")
15
16
17
18
19
20
21
```

The code uses 'with' statements for both writing to and reading from files. It defines a function 'read_file' that takes a filename as an argument and prints its contents. It then calls this function twice, once for 'data1.txt' and once for 'data2.txt'. The right side of the interface shows the 'Output' pane, which displays the two lines of text: 'Hello from Data File 1' and 'Hello from Data File 2'.

Task 4: Optimizing Search Logic

Refactored Code

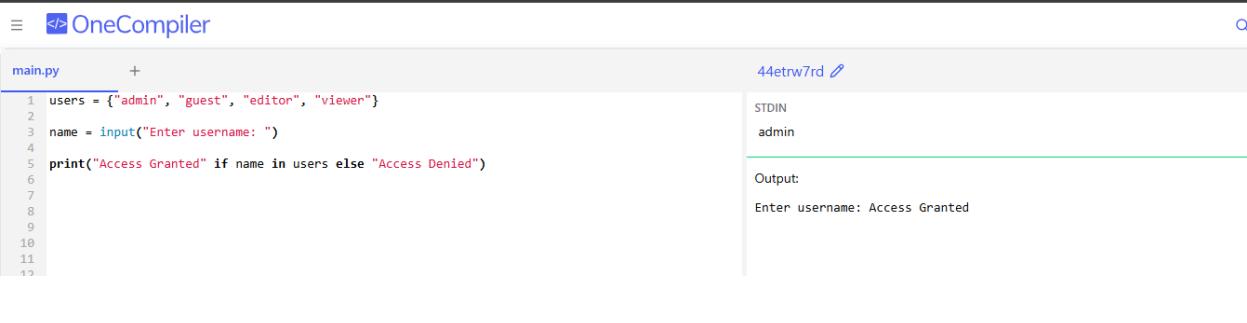
```
users = {"admin", "guest", "editor", "viewer"}
name = input("Enter username: ")

print("Access Granted" if name in users else "Access Denied")
```

Explanation

Used set for O(1) average lookup time instead of O(n) list search.

Screenshots



A screenshot of the OneCompiler interface. On the left, the code editor shows a file named 'main.py' with the following content:

```
1 users = {"admin", "guest", "editor", "viewer"}
2
3 name = input("Enter username: ")
4
5 print("Access Granted" if name in users else "Access Denied")
6
7
8
9
10
11
12
```

The code defines a set 'users' containing four strings: 'admin', 'guest', 'editor', and 'viewer'. It then prompts the user for a 'username' using the 'input' function. Finally, it prints 'Access Granted' if the entered name is in the 'users' set, or 'Access Denied' otherwise. The right side of the interface shows the 'Output' pane, which displays the message 'Enter username: Access Granted' after the user has entered 'admin'.

❖ Task 5: Refactoring Procedural Code into OOP Design

💻 Refactored Code

```
class EmployeeSalaryCalculator:
    def __init__(self, salary):
        self.salary = salary

    def calculate_tax(self):
        return self.salary * 0.2

    def calculate_net_salary(self):
        return self.salary - self.calculate_tax()

employee = EmployeeSalaryCalculator(50000)
print(employee.calculate_net_salary())
```

📝 Explanation

Encapsulated salary logic inside a class for better scalability and structure.

📸 Screenshots



The screenshot shows the OneCompiler interface. On the left, the code editor displays 'main.py' with the refactored code. On the right, the 'Output' panel shows the result of running the code, which is '40000.0'.

```
44etrw7rd ⚡
main.py
+
1 ~ class EmployeeSalaryCalculator:
2 ~     def __init__(self, salary):
3 ~         self.salary = salary
4 ~
5 ~     def calculate_tax(self):
6 ~         return self.salary * 0.2
7 ~
8 ~     def calculate_net_salary(self):
9 ~         return self.salary - self.calculate_tax()
10 ~
11 ~
12 ~ employee = EmployeeSalaryCalculator(50000)
13 ~ print(employee.calculate_net_salary())
14 ~
15 ~
16 ~
17 ~
18 ~
19 ~
```

Output:
40000.0

❖ Task 6: Refactoring for Performance Optimization

💻 Refactored Code

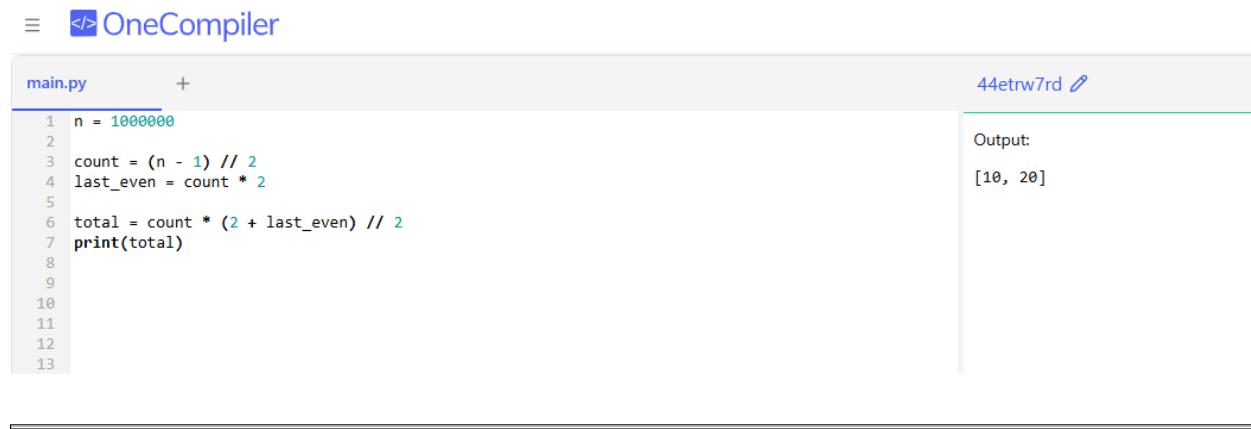
```
n = 1000000
count = (n - 1) // 2
last_even = count * 2
```

```
total = count * (2 + last_even) // 2
print(total)
```

📝 Explanation

Replaced loop ($O(n)$) with mathematical formula ($O(1)$) for performance optimization.

📸 Screenshots



A screenshot of the OneCompiler interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 n = 1000000
2
3 count = (n - 1) // 2
4 last_even = count * 2
5
6 total = count * (2 + last_even) // 2
7 print(total)
8
9
10
11
12
13
```

The code is numbered from 1 to 13. On the right, there is a "Output:" section showing the result of running the code: [10, 20]. Above the output, the URL "44etrw7rd" is visible.

❖ Task 7: Removing Hidden Side Effects

💻 Refactored Code

```
def add_item(data, x):
    return data + [x]

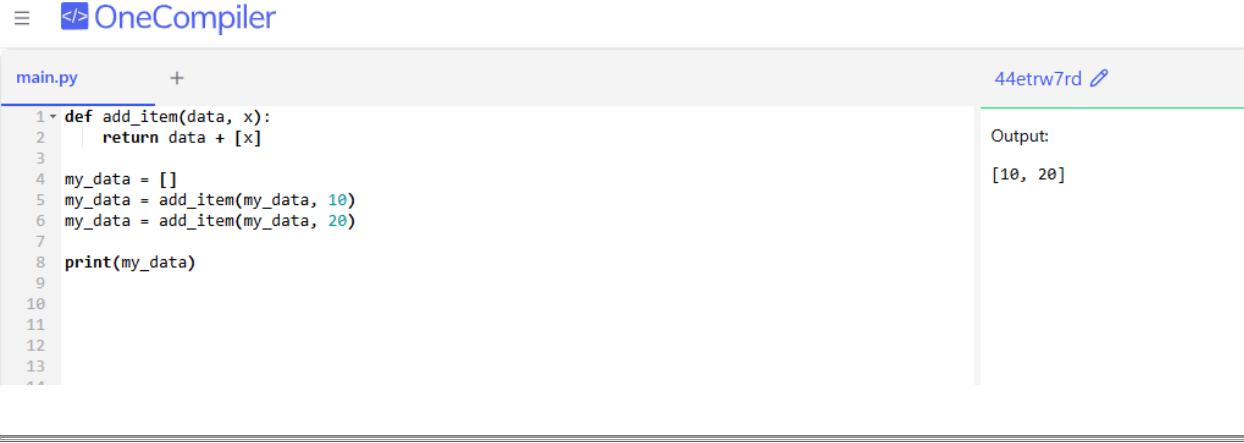
my_data = []
my_data = add_item(my_data, 10)
my_data = add_item(my_data, 20)

print(my_data)
```

📝 Explanation

Removed global mutable state. Function now returns new list instead of modifying shared data.

Screenshots



A screenshot of the OneCompiler interface. On the left, there is a code editor window titled "main.py" containing the following Python code:

```
1 def add_item(data, x):
2     return data + [x]
3
4 my_data = []
5 my_data = add_item(my_data, 10)
6 my_data = add_item(my_data, 20)
7
8 print(my_data)
9
10
11
12
13
```

To the right of the code editor is a vertical toolbar with a "44etrw7rd ⚡" button. Below the toolbar, the word "Output:" is followed by the list "[10, 20]".

Task 8: Refactoring Complex Input Validation Logic

Refactored Code

```
def is_long_enough(password):
    return len(password) >= 8

def has_digit(password):
    return any(c.isdigit() for c in password)

def has_uppercase(password):
    return any(c.isupper() for c in password)

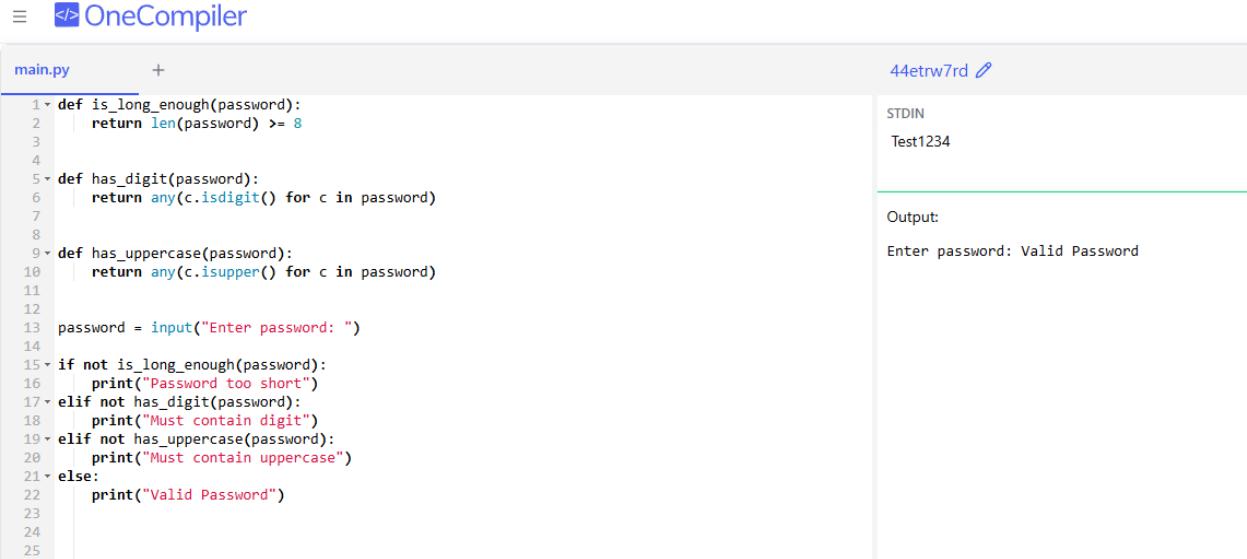
password = input("Enter password: ")

if not is_long_enough(password):
    print("Password too short")
elif not has_digit(password):
    print("Must contain digit")
elif not has_uppercase(password):
    print("Must contain uppercase")
else:
    print("Valid Password")
```

Explanation

Separated validation into small reusable functions for clarity and testability.

Screenshots



The screenshot shows a Python script named main.py being run on OneCompiler. The code defines three functions: is_long_enough, has_digit, and has_uppercase. It then prompts the user for a password and prints a message indicating whether it is valid based on length, digits, and uppercase letters.

```
main.py + 44etrw7rd Ø
1 def is_longEnough(password):
2     return len(password) >= 8
3
4
5 def hasDigit(password):
6     return any(c.isdigit() for c in password)
7
8
9 def hasUppercase(password):
10    return any(c.isupper() for c in password)
11
12
13 password = input("Enter password: ")
14
15 if not is_longEnough(password):
16     print("Password too short")
17 elif not hasDigit(password):
18     print("Must contain digit")
19 elif not hasUppercase(password):
20     print("Must contain uppercase")
21 else:
22     print("Valid Password")
23
24
25
```

STDIN
Test1234

Output:
Enter password: Valid Password

Overall Conclusion

This lab shows how AI-assisted refactoring improves readability, maintainability, modularity, and performance. Legacy code was successfully transformed into modern, efficient, and scalable Python code using best practices.

 **Formatting Recommendation for Final PDF:** - Font: Times New Roman - Size: 12 - Line Spacing: 1.5 - Keep headings bold - Add page breaks between tasks before final submission