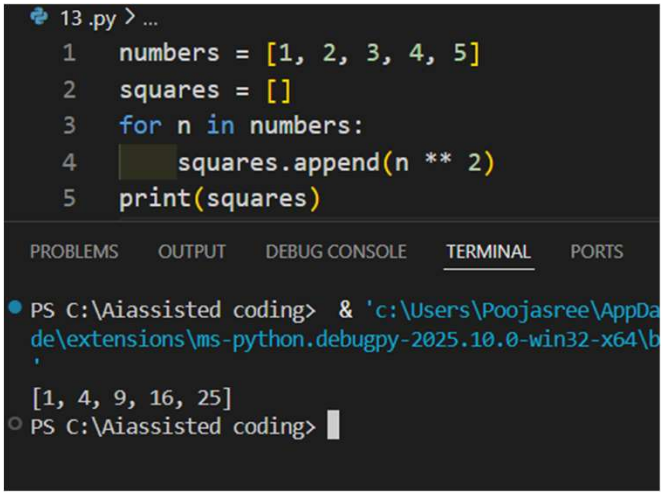
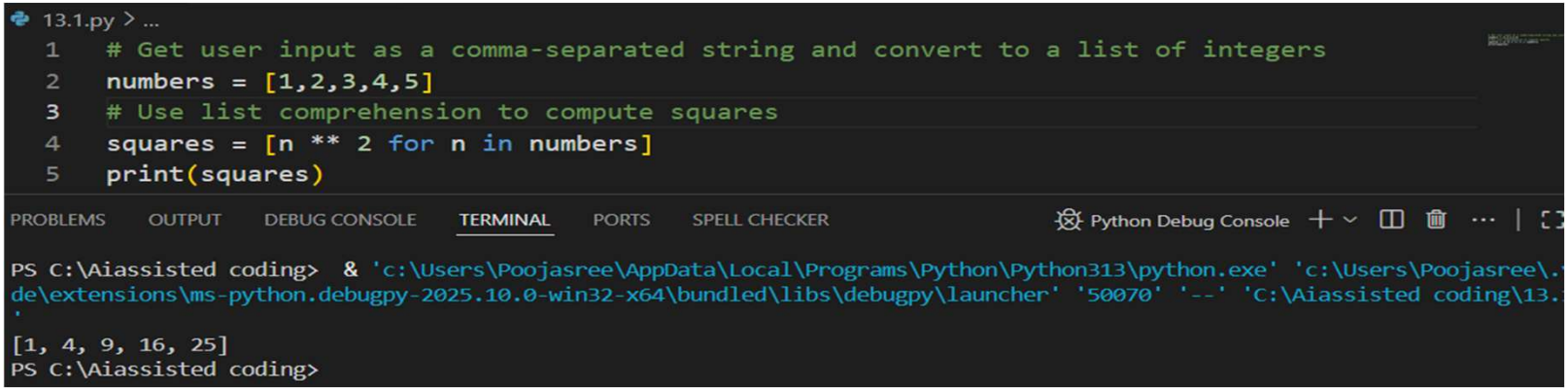
Assignment -13

Task-1:: Refactor repeated loops into a cleaner, more Pythonic approach.

Legacy code:



New code:



Explanation:
The main differences between the two codes are:

Loop vs. List Comprehension

The first code uses a for loop to iterate through numbers and appends each square to the squares list.

The second code uses a list comprehension, which is a more concise and Pythonic way to create a new list by applying an operation to each item in an existing list.

Code Length and Readability

The first code is longer and more explicit, showing each step.

The second code is shorter and easier to read for simple operations.

Efficiency

Both codes produce the same output and have similar performance for small lists, but list comprehensions are generally faster and preferred for simple transformations.

Output:

Both codes output [1, 4, 9, 16, 25].

The difference is mainly in style and conciseness, not in functionalit

Task-2:Simplify string concatenation.

Legacy code:

```
1  words = ["AI", "helps", "in", "refactoring", "code"]
2  sentence = ""
3  for word in words:
4      sentence += word + " "
5  print(sentence.strip())
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER  PytH

PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Pytho
asree\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\
ssisted coding\13 .py'
AI helps in refactoring code
PS C:\Aiassisted coding>
```

New code:

```
13.2.py > ...
1  words = ["AI", "helps", "in", "refactoring", "code"]  # List of words to join
2  sentence = " ".join(words)  # Join words with a space separator
3  print(sentence)  # Print the resulting sentence
4
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER  Python Debug Console

PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Python\Python313\python.exe' 'c
asree\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '59830'
ssisted coding\13.2.py'
AI helps in refactoring code
PS C:\Aiassisted coding>
```

Explanation:

The difference between the two codes is:

String Concatenation Method:

The first code uses " ".join(words), which is a built-in and efficient way to join a list of words with spaces.

The second code uses a loop with += to add each word and a space to the sentence string.

Efficiency:

" ".join(words) is faster and more memory-efficient, especially for large lists, because it joins all words in one operation.

Using += in a loop creates a new string each time, which is slower and less efficient.

Readability:

" ".join(words) is more concise and easier to read.

The loop is longer and more verbose.

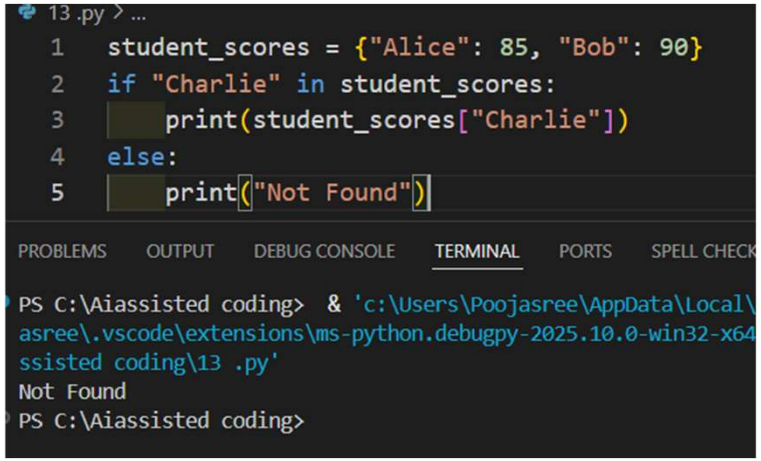
Output:

Both codes produce the same output:

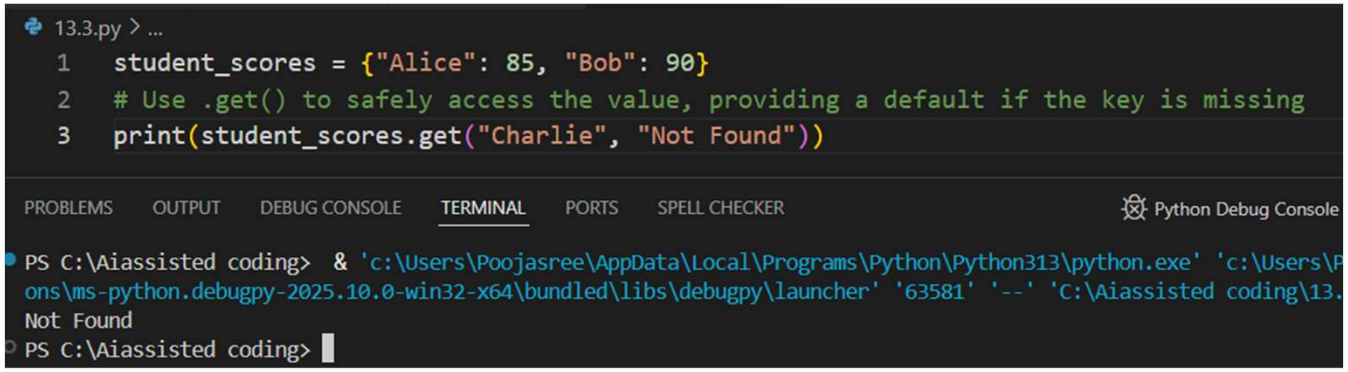
AI helps in refactoring code

Task-3:Replace manual dictionary lookup with a safer method

Legacy code:



New code:



Explanation:
The difference between these codes is:

Key Lookup Method:

The first code uses .get() to access the value for "Charlie". If "Charlie" is not found, it returns "Not Found" by default.

The second code checks if "Charlie" is in the dictionary using if "Charlie" in student_scores:. If found, it prints the value; otherwise, it prints "Not Found".

Conciseness:

The .get() method is more concise and requires only one line.

The if...else approach is longer and uses multiple lines.

Readability and Pythonic Style:

.get() is considered more Pythonic and is easier to read for simple lookups with a default value.

The if...else method is more explicit but less concise.

Functionality:

Both codes produce the same output: Not Found if "Charlie" is not in the dictionary.

Summary:

Using .get() is shorter, more Pythonic, and handles missing keys gracefully in a single line.
The if...else approach is more verbose but functionally equivalent.

Task-4:Refactor repetitive if-else blocks

Legacy code:

```
13 .py > ...
1  operation = "multiply"
2  a, b = 5, 3
3  if operation == "add":
4      result = a + b
5  elif operation == "subtract":
6      result = a - b
7  elif operation == "multiply":
8      result = a * b
9  else:
10     result = None
11 print(result)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORT

PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\A
ons\ms-python.debugpy-2025.10.0-win32-x64\bundled
15
PS C:\Aiassisted coding>
```

New code:

```
1  operation = "multiply"
2  a, b = 5, 3
3  # Dictionary mapping operation names to functions
4  operations = {
5      "add": lambda x, y: x + y,
6      "subtract": lambda x, y: x - y,
7      "multiply": lambda x, y: x * y}
8  # Use .get() to handle missing operations gracefully
9  result = operations.get(operation, lambda x, y: None)(a, b)
10 print(result)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  SPELL CHECKER

PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Python\Python313\python.e
ons\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '63683' '--' 'C:\Aiassi
15
PS C:\Aiassisted coding>
```

Explanation:
The differences between these two codes are:

Approach:

The first code uses a **dictionary mapping** (operations) to associate operation names with functions (lambdas). It then retrieves and executes the correct function based on the operation variable.

The second code uses a series of **if-elif-else statements** to check the value of operation and perform the corresponding calculation.

Scalability:

The dictionary mapping approach is **more scalable** and easier to extend. To add a new operation, you just add another key-value pair to the dictionary.

The if-elif-else approach becomes longer and harder to manage as you add more operations.

Readability and Conciseness:

The dictionary mapping is **more concise** and keeps the code cleaner, especially for many operations.

The if-elif-else approach is more verbose and repetitive.

Handling Missing Operations:

The dictionary mapping uses .get() with a default lambda that returns None if the operation is not found, making it **safer and more Pythonic**.

The if-elif-else approach handles unknown operations in the else block.

Summary:

Both codes produce the same output, but the dictionary mapping approach is more Pythonic, concise, and easier to maintain or extend.

Task-5:Optimize nested loops for searching.

Legacy code

```
items = [10, 20, 30, 40, 50]
found = False
for i in items:
    if i == 30:
        found = True
        break
print("Found" if found else "Not Found")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ···  Pytho

PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\P
'c:\Users\Poojasree\.vscode\extensions\ms-python.debugpy-2025.1
uncher' '63722' '--' 'C:\Aiassisted coding\13 .py'
Found
PS C:\Aiassisted coding>
```

New code :

```
items = [10, 20, 30, 40, 50]
# Use 'in' keyword for efficient search
found = 30 in items
print("Found" if found else "Not Found")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  ···  Python Debug Console

PS C:\Aiassisted coding>  & 'c:\Users\Poojasree\AppData\Local\Programs\Python
'c:\Users\Poojasree\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\
uncher' '63760' '--' 'C:\Aiassisted coding\13.5.py'
Found
PS C:\Aiassisted coding>
```

Explanation:

Approach:

The legacy code uses a for loop with an if statement and a break to search for the element.

The optimized code uses the in keyword for direct membership testing.

Code Length:

The legacy code is longer and more verbose.

The optimized code is concise and fits in a single line for the search.

Readability:

The optimized code is easier to read and understand.

The legacy code is less readable due to extra lines and logic.

Performance:

Both have similar performance for lists, but the in keyword is implemented efficiently in Python.

Pythonic Style:

The optimized code follows Python best practices and is more Pythonic.

The legacy code is more like traditional programming in other languages.