ASSIGNMENT-13.4

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

Legacy Code:

```
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
squares.append(n ** 2)
print(squares)
```

Prompt:

Refactor this loop into a more pythonic version using list.

Code with Output:

Legacy code with output:

```
numbers = [1, 2, 3, 4, 5]
squares = []
for n in numbers:
    squares.append(n ** 2)
print(squares)
```

```
[1, 4, 9, 16, 25]
```

Pythonic version of code using list:

Code with output:

```
numbers = [1, 2, 3, 4, 5] # Define a list of numbers
squares = [n ** 2 for n in numbers] # Use a list comprehension to calculate the square of each number
print(squares) # Print the resulting list of squares
```

```
[1, 4, 9, 16, 25]
```

EXPLANATION:

- `numbers = [1, 2, 3, 4, 5]` : This line initializes a list named `numbers` with the integers from 1 to 5.
- `squares = [n ** 2 for n in numbers]` : This is a list comprehension. It iterates through each element (`n`) in the `numbers` list and calculates its square (`n ** 2`). The results are collected into a new list called `squares` .
- `print(squares)` : This line prints the `squares` list to the console.

The output of this code will be `[1, 4, 9, 16, 25]` .

TASK-2: Simplify string concatenation.

#Legacy Code:

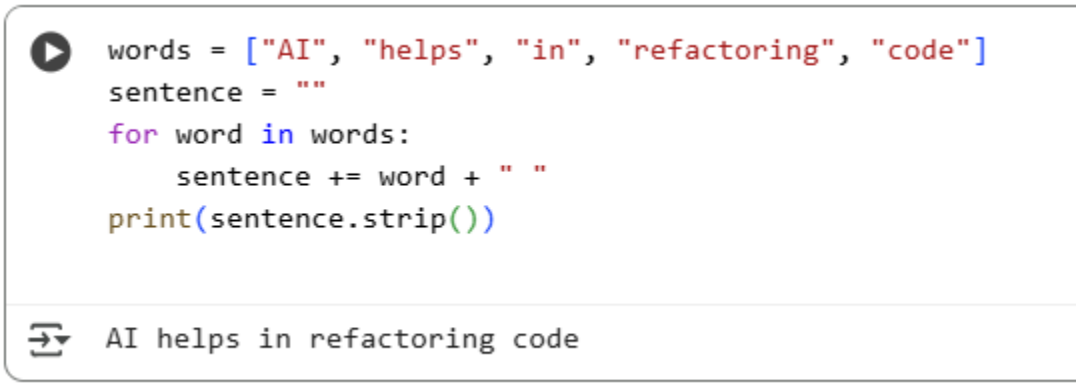
```
words = ["AI", "helps", "in", "refactoring", "code"]
sentence = ""
for word in words:
    sentence += word + " "
print(sentence.strip())
```

Prompt:

Refactor this string concatenation into a more pythonic version using join().

Code with Output:

Legacy code with output:



Pythonic version of code using join():

Code with output:

```
words = ["AI", "helps", "in", "refactoring", "code"] # Define a list of words
sentence = " ".join(words) # Join the words in the list with spaces to form a sentence
print(sentence) # Print the resulting sentence
```

```
AI helps in refactoring code
```

Explanation:

- `words = ["AI", "helps", "in", "refactoring", "code"]` : This line initializes a list named `words` containing several strings.
- `sentence = " ".join(words)` : This is the core of the refactored code. The `join()` method is called on the string `" "` (a space). This means that each element in the `words` list will be joined together with a space in between them. The resulting string is stored in the `sentence` variable.
- `print(sentence)` : This line prints the final `sentence` string to the console.

The output of this code will be `AI helps in refactoring code`.

TASK-3: Replace manual dictionary lookup with a safer method.

#Legacy code

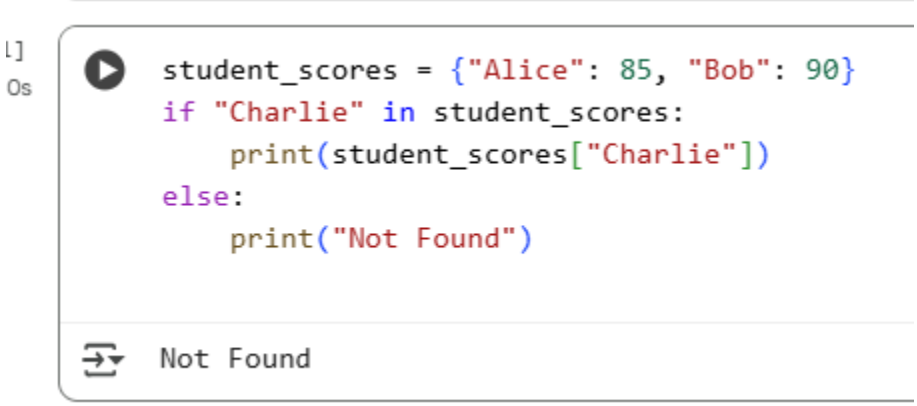
```
student_scores = {"Alice": 85, "Bob": 90}
if "Charlie" in student_scores:
    print(student_scores["Charlie"])
else:
  print("Not Found")
```

Prompt:

Refactor this dictionary into a more pythonic version using get().

Code with output:

Legacy code with output:



Pythonic version of code using get() method:

Code with output:

```
student_scores = {"Alice": 85, "Bob": 90} # Define a dictionary of student scores
# Use the get() method to retrieve the score for "Charlie", providing a default value if the key is not found
print(student_scores.get("Charlie", "Not Found"))
```

```
Not Found
```

Explanation:

- `student_scores = {"Alice": 85, "Bob": 90}` : This line initializes a dictionary named `student_scores` with two key-value pairs, representing student names and their scores.
- `print(student_scores.get("Charlie", "Not Found"))` : This line uses the `get()` method to retrieve the value associated with the key "Charlie" from the `student_scores` dictionary.
    - If "Charlie" exists as a key in the dictionary, its corresponding value (the score) would be returned.
    - However, since "Charlie" is not a key in `student_scores`, the `get()` method returns the specified default value, which is the string "Not Found".
- The `print()` function then displays the returned value.

The output of this code will be `Not Found`. This is a more "pythonic" way to handle potential `KeyError` exceptions that would occur if you tried to access a non-existent key using square brackets (e.g., `student_scores["Charlie"]`).

TASK-4: Refactor repetitive if-else blocks.

#Legacy code

```
operation = "multiply"
a, b = 5, 3
if operation == "add":
    result = a + b
elif operation == "subtract":
    result = a - b
elif operation == "multiply":
    result = a * b
else:
    result = None
print(result)
```

Prompt:

Refactor

Code with output:

Legacy code with output:

```
operation = "multiply"
a, b = 5, 3

if operation == "add":
    result = a + b
elif operation == "subtract":
    result = a - b
elif operation == "multiply":
    result = a * b
else:
    result = None

print(result)
```

```
15
```

Pyhtonic version of code using dictionary mapping:

Code with output:

```
operation = "multiply"
a, b = 5, 3

# Define a dictionary mapping operation names to lambda functions
operations = {
    "add": lambda x, y: x + y,       # Addition operation
    "subtract": lambda x, y: x - y,    # Subtraction operation
    "multiply": lambda x, y: x * y,   # Multiplication operation
}

# Get the corresponding function from the dictionary, with a default of None
result = operations.get(operation)

# If a valid function is found, apply it to a and b
if result:
    result = result(a, b)
else:
    result = None # Handle the case where the operation is not found

print(result)
```

```
15
```

Explanation:

1. `operation = "multiply"`: This line sets a variable `operation` to the string "multiply". This string will determine which mathematical operation is performed.
2. `a, b = 5, 3`: This line assigns the value 5 to variable `a` and 3 to variable `b`. These are the numbers the operations will be performed on.
3. `operations = { ... }`: This creates a dictionary called `operations`.
    - The keys of this dictionary are strings representing the names of the operations ("add", "subtract", "multiply").
    - The values are `lambda` functions. Lambda functions are small, anonymous functions. Each lambda function takes two arguments (`x` and `y`) and performs a specific mathematical operation (addition, subtraction, or multiplication).
4. `operation_func = operations.get(operation)`: This line uses the `.get()` method to retrieve the value associated with the key stored in the `operation` variable ("multiply") from the `operations` dictionary.
    - If the key is found, `operation_func` will hold the corresponding lambda function (in this case, the multiplication function).
    - If the key is not found, `.get()` returns `None` by default.
5. `if operation_func:`: This checks if `operation_func` is not `None` (meaning the operation was found in the dictionary).
6. `result = operation_func(a, b)`: If `operation_func` is not `None`, this line calls the retrieved lambda function with `a` and `b` as arguments and assigns the returned value to the `result` variable.
7. `else: result = None`: If `operation_func` is `None` (the operation was not found), `result` is set to `None`.
8. `print(result)`: Finally, this line prints the value stored in the `result` variable.

In this specific execution, since `operation` is "multiply", the multiplication lambda function is retrieved, and `result` will be `5 * 3`, which is `15`.

TASK-5: Optimize nested loops for searching.

#Legacy code

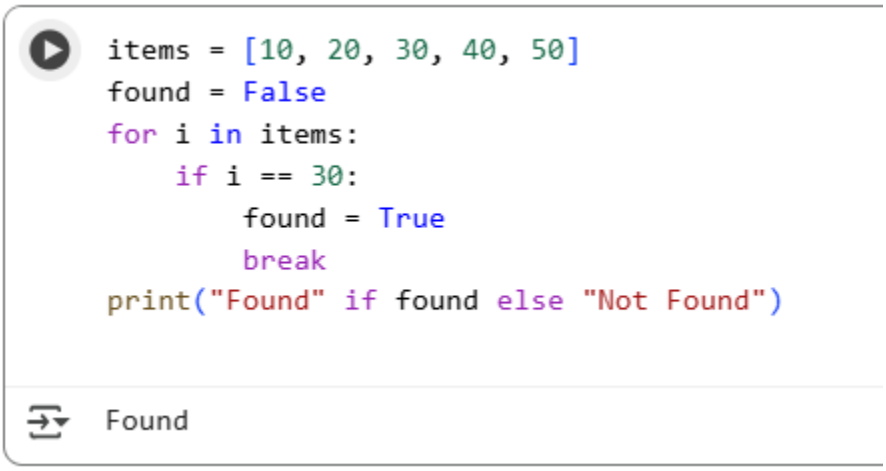
```
items = [10, 20, 30, 40, 50]
found = False
for i in items:
    if i == 30:
        found = True
        break
print("Found" if found else "Not Found")
```

Prompt:

Refactor this code into a more pythonic version using python keywords.

Code with Output:

Legacy code with output:



Pythonic version of code using python keywords:

Code with Output:

```
items = [10, 20, 30, 40, 50] # Define a list of items
# Check if 30 is present in the list using the 'in' keyword
print("Found" if 30 in items else "Not Found")
```

```
Found
```

Explanation:

- `items = [10, 20, 30, 40, 50]`: This line initializes a list named `items` with several integer values.
- `print("Found" if 30 in items else "Not Found")`: This line does two things:
  - `30 in items`: This is the core of the membership test. The `in` keyword checks if the value `30` is present as an element within the `items` list. This expression evaluates to either `True` (if 30 is found) or `False` (if 30 is not found).
  - `"Found" if ... else "Not Found"`: This is a conditional expression (sometimes called a ternary operator). It's a concise way to choose between two values based on a condition. If the condition (`30 in items`) is `True`, the expression evaluates to `"Found"`. If the condition is `False`, it evaluates to `"Not Found"`.
- The `print()` function then displays the result of the conditional expression.

In this case, since `30` is in the `items` list, the `in` check is `True`, and the code will print `Found`. This approach is generally preferred over manual looping for checking membership as it's more readable and often more performant.