

AI Assisted Coding

Assignment-10.3

Akula Vivek

(2403A51258)

Batch11

Task-1:

Prompt:

Refactor the following nested conditional Python code for better readability. Aim to simplify the logic using cleaner structures such as dictionaries, helper functions, or other Pythonic approaches:

```
def discount(price, category):  
    if category == "student":  
        if price > 1000:  
            return price * 0.9  
        else:  
            return price * 0.95  
    else:  
        if price > 2000:  
            return price * 0.85  
        else:  
            return price
```

Expected Output:

- Refactored code using cleaner logic, possibly a dictionary or separate helper functions.

Code:

```
10.3.1.py > ...
Tabnine | Edit | Test | Explain | Document
1 def student_discount(price):
2     return price * 0.9 if price > 1000 else price * 0.95
3
4 Tabnine | Edit | Test | Explain | Document
5 def regular_discount(price):
6     return price * 0.85 if price > 2000 else price
7
8 discount_strategies = {
9     "student": student_discount,
10    "regular": regular_discount
11 }
12
13 Tabnine | Edit | Test | Explain | Document
14 def discount(price, category):
15     strategy = discount_strategies.get(category, regular_discount)
16     return strategy(price)
17
18 # Example outputs
19 print(discount(1200, 'student')) # Expected: 1080.0
20 print(discount(900, 'student')) # Expected: 855.0
21 print(discount(2500, 'regular')) # Expected: 2125.0
22 print(discount(1500, 'regular')) # Expected: 1500
```

Output:

```
... Filter Code
[Running] python -u "e:\Aicoding\Aicoding\10.3.1.py"
1080.0
855.0
2125.0
1500
[Done] exited with code=0 in 1.049 seconds
```

Task-2:

Prompt:

Refactor the following Python code to eliminate redundant nested loops. Optimize it using more efficient Python features such as sets:

```
def find_common(a, b):
    res = []
    for i in a:
        for j in b:
            if i == j:
                res.append(i)
    return res
```

Expected Output:

Cleaner version using Python sets (set(a) & set(b))

Code:

```
10.3.2.py > ...  
  
1 def find_common(a, b):  
2     return list(set(a) & set(b))  
3  
4 # Example output  
5 print(find_common([1, 2, 3, 4], [3, 4, 5, 6])) # Expected: [3, 4]  
6 print(find_common(['apple', 'banana'], ['banana', 'cherry'])) # Expected: ['banana']  
7
```

Output:

```
[Running] python -u "e:\Aicoding\Aicoding\10.3.2.py"  
[3, 4]  
['banana']  
  
[Done] exited with code=0 in 0.306 seconds
```

Task-3:

Prompt:

Refactor the following Python class to improve readability and maintainability. Apply proper naming conventions, encapsulation, and clear method responsibilities. Add docstrings for better understanding.

```
class emp:  
def __init__(self,n,s):  
self.n=n  
self.s=s  
def inc(self,p):  
self.s=self.s+(self.s*p/100)  
def pr(self):  
print("emp:",self.n,"salary:",self.s)
```

Expected Output:

- Employee class with meaningful methods (increase_salary, display_info), formatted output, and added docstrings.

Code:

```

10.3.3.py > Employee
1 class Employee:
2     """Represents an employee with a name and salary."""
3
4     Tabnine | Edit | Test | Explain | Document
5     def __init__(self, name, salary):
6         """Initialize employee with name and salary."""
7         self._name = name
8         self._salary = salary
9
10    Tabnine | Edit | Test | Explain | Document
11    def increase_salary(self, percent):
12        """Increase salary by a given percentage."""
13        self._salary += self._salary * percent / 100
14
15    Tabnine | Edit | Test | Explain | Document
16    def display_info(self):
17        """Display employee information."""
18        print(f"Employee: {self._name}, Salary: {self._salary:.2f}")
19
20    # Example usage
21    emp1 = Employee("Alice", 5000)
22    emp1.increase_salary(10)
23    emp1.display_info() # Expected: Employee: Alice, Salary: 5500.00

```

Output:

```

[Running] python -u "e:\Aicoding\Aicoding\10.3.3.py"
Employee: Alice, Salary: 5500.00

[Done] exited with code=0 in 1.289 seconds

```

Task-4:

Prompt:

Refactor the following long, unstructured Python function by breaking it into smaller, reusable helper functions. Improve readability and maintainability by modularizing the logic.

```
def process_scores(scores):
```

```
total = 0
```

```
for s in scores:
```

```
total += s
```

```
avg = total / len(scores)
```

```
highest = scores[0]
```

```
for s in scores:
```

```
if s > highest:
```

```
highest = s
```

```
lowest = scores[0]
```

```
for s in scores:
```

```
if s < lowest:
```

```
lowest = s
```

```
print("Average:", avg)
```

```
print("Highest:", highest)
```

```
print("Lowest:", lowest)
```

Expected Output:

- Split into functions: calculate_average, find_highest, find_lowest.
- Clean main process_scores() using helper functions.

Code:

```
10.3.4.py > process_scores
Tabnine | Edit | Test | Explain | Document
1 def calculate_average(scores):
2     """Return the average of the scores."""
3     return sum(scores) / len(scores) if scores else 0
4
Tabnine | Edit | Test | Explain | Document
5 def find_highest(scores):
6     """Return the highest score."""
7     return max(scores) if scores else None
8
Tabnine | Edit | Test | Explain | Document
9 def find_lowest(scores):
10    """Return the lowest score."""
11    return min(scores) if scores else None
12
Tabnine | Edit | Test | Explain | Document
13 def process_scores(scores):
14     avg = calculate_average(scores)
15     highest = find_highest(scores)
16     lowest = find_lowest(scores)
17     print(f"Average: {avg:.2f}")
18     print(f"Highest: {highest}")
19     print(f"Lowest: {lowest}")
20
# Example usage
21 scores = [88, 92, 75, 63, 99]
22 process_scores(scores)
23 # Expected output:
24 # Average: 83.40
25 # Highest: 99
26 # Lowest: 63
```

Output:

```
[Running] python -u "e:\AICoding\AICoding\10.3.4.py"
Average: 83.40
Highest: 99
Lowest: 63

[Done] exited with code=0 in 0.301 seconds
```

Task-5:

Prompt:

Review and refactor the following Python code to improve error handling, naming conventions, and readability. Add a docstring that explains the function and its error handling.

```
def div(a,b):
    return a/b
print(div(10,0))
```

Expected Output:

- Function with proper error handling using try-except.
- Better naming (divide_numbers).
- AI-generated docstring explaining error handling.

Code:

```
10.3.5.py > ...
Tabnine | Edit | Test | Explain | Document
1 def divide_numbers(a, b):
2     """
3     Divide two numbers and handle division by zero.
4     Returns the result if successful, or a message if an error occurs.
5     """
6     try:
7         return a / b
8     except ZeroDivisionError:
9         return "Error: Division by zero is not allowed."
10    except TypeError:
11        return "Error: Both arguments must be numbers."
12
13    # Example output
14    print(divide_numbers(10, 0))    # Expected: Error: Division by zero is not allowed.
15    print(divide_numbers(10, 2))    # Expected: 5.0
16    print(divide_numbers(10, 'a'))  # Expected: Error: Both arguments must be numbers.
17
```

Output:

```
[Running] python -u "e:\Aicoding\Aicoding\10.3.5.py"
Error: Division by zero is not allowed.
5.0
Error: Both arguments must be numbers.

[Done] exited with code=0 in 0.188 seconds
```

Task-6:

Prompt:

Simplify the following overly complex Python function that uses deeply nested conditionals. Refactor it into a cleaner version using elif statements or a dictionary mapping for better readability and maintainability.

```
def grade(score):
```

```
if score >= 90:
```

```
    return "A"
```

```
else:
```

```
if score >= 80:
```

```
    return "B"
```

```
else:
```

```
if score >= 70:
```

```
    return "C"
```

```
else:
```

```
if score >= 60:
```

```
    return "D"
```

```
else:
```

```
    return "F"
```

Expected Output:

- Cleaner logic using elif or dictionary mapping.

Code:

```
10.3.6.py > ...
Tabnine | Edit | Test | Explain | Document
1  def grade(score):
2      if score >= 90:
3          return "A"
4      elif score >= 80:
5          return "B"
6      elif score >= 70:
7          return "C"
8      elif score >= 60:
9          return "D"
10     else:
11         return "F"
12
13     # Example output
14     print(grade(95)) # Expected: A
15     print(grade(85)) # Expected: B
16     print(grade(75)) # Expected: C
17     print(grade(65)) # Expected: D
18     print(grade(55)) # Expected: F
19
```

Output:

```
[Running] python -u "e:\Aicoding\Aicoding\10.3.6.py"
A
B
C
D
F

[Done] exited with code=0 in 0.183 seconds
```