# Unit-6

## ****Question 1: Running**** styleLint.py

### ****Task: Run**** styleLint.py



Figure 1: Run styleLint.py

### What happens when the code is run?

When the code is run, it executes the factorial function correctly. However, there are some formatting issues:

* The indentation is incorrect.
* The if-else block lacks proper alignment and consistent indentation.

1. **Can you modify this code for a more favorable outcome? What amendments have you made to the code?**

We can modify this code to improve readability and follow to Python's PEP8 style guide.

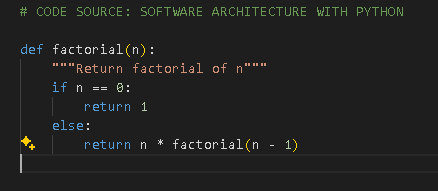


Figure 2: Updated code of styleLint.py

### Changes made:

First, we corrected the indentation levels, then added a space around operators for better readability and lastly, updated the docstring to remove extra spaces.

## Question 2: Using pylint on pylintTest.py

### Task: Ensure pylint is installed and run it on pylintTest.py.

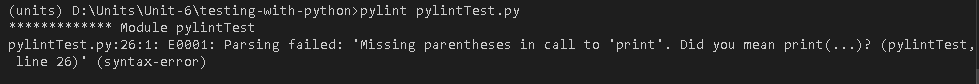


Figure 3: Executing pylintTest

### First Modification made based on the Errors Identified:

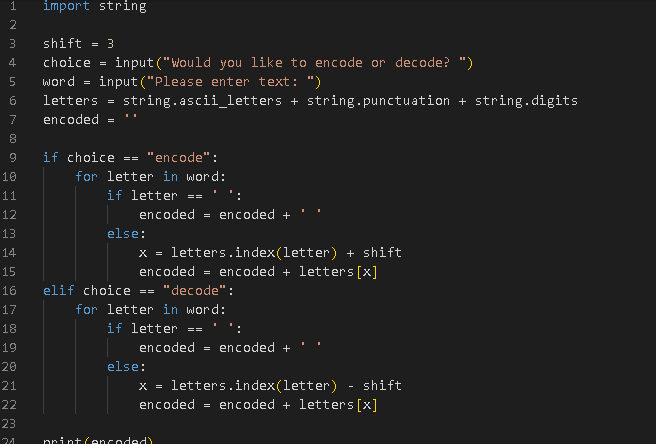


Figure 4: First Modification of code

**Output of First Modification:**

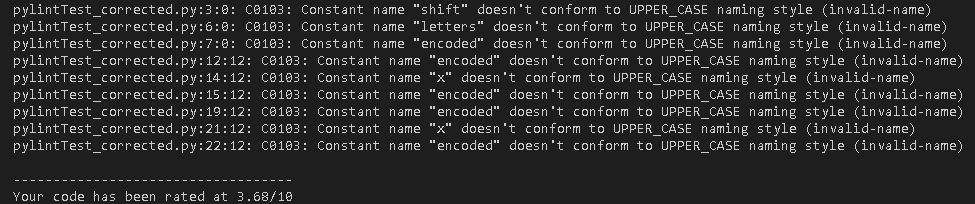


Figure 5: Output of First Modification

**Issues Identified by pylint**:

* *Trailing Whitespace (C0303)*
* *Missing Module Docstring (C0114)*
* *Invalid Module Name (C0103)*
* *Invalid Constant Name (C0103)*

### Second Modification Made:

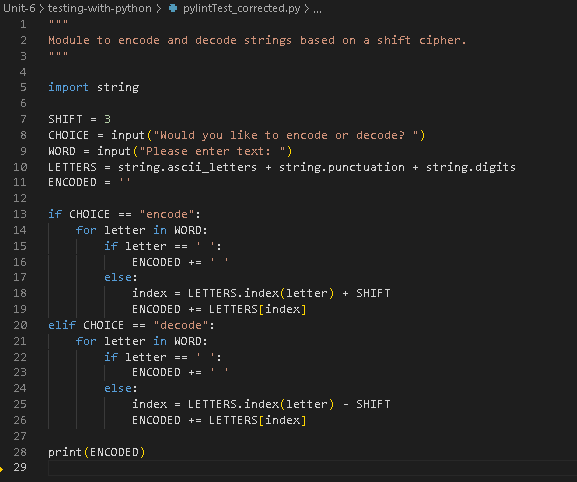
****

Figure 6: Second Modification Made

**Output**:

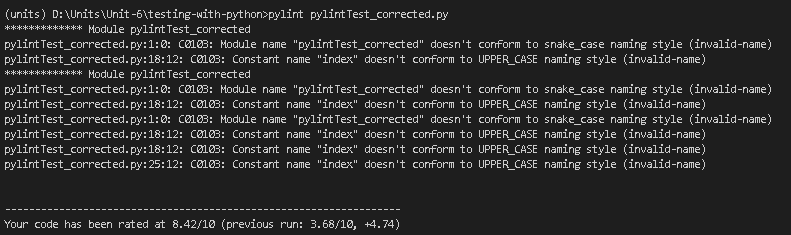


Figure 7: Output of Second Modification

**Remaining Issues:**

* *Module Name Not in Snake Case (C0103)*
* *Constant Name Not in UPPER\_CASE (C0103)*

### Third Modification:

* *The file is renamed to pylint\_test\_corrected.py.*
* *The variable names that are not constants remain in lowercase (i.e., encoded and index).*
* *Override Pylint's Convention Temporarily*
* *Pylint Configuration Adjustment*

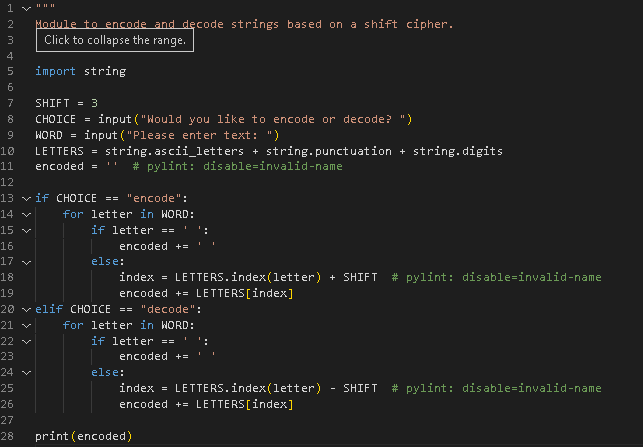


Figure 8: Third Modification

**Output:**

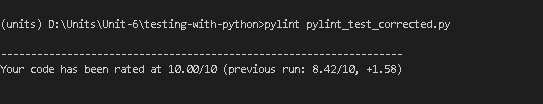
****

Figure 9: Output of last Modification

The Final Code achieved a 10.00/10 score on pylint that means this code now fully adheres to Python's best practices and style guidelines.

## ****Question 3: Using**** flake8 ****on**** pylint\_test\_corrected.py ****and**** metricTest.py

### ****Run** flake8 **on** pylint\_test\_corrected.py **and** metricTest.py**:****

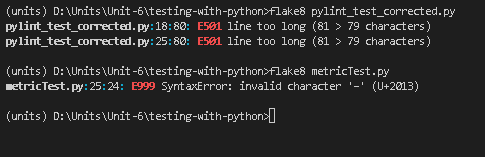


Figure 10: Run flake8 on pylint\_test\_corrected.py

### ****Analysis of**** flake8 ****Output:****

1. **pylint\_test\_corrected.py Issues:**

* **E501 line too long (81 > 79 characters)**: This error indicates that the lines at 18 and 25 are longer than the PEP 8 recommendation of 79 characters.

1. **metricTest.py Issues:**

* **E999 SyntaxError: invalid character '–' (U+2013)**: This indicates there is an invalid character, specifically an en dash (–), which is different from a regular hyphen or minus sign (-). This character can sometimes be introduced through text editors or copy-pasting from formatted documents.

### Solution

We need to shorten the lines at positions 18 and 25 to be 79 characters or fewer.

**Output**:

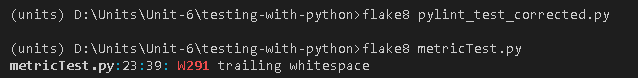


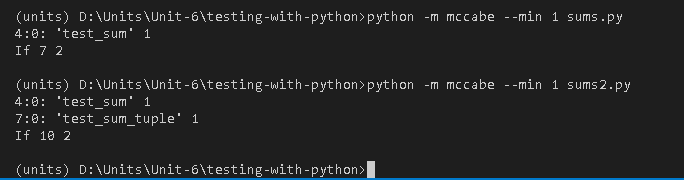
Figure 11: Output after updating the code

Summary:

In Question 3, we have learned how flake8 can be used for finding and fixing various quality aspects in Python scripts. First, we fixed syntax-related bugs and then made the code compat­ible with Python 3, followed by applying style improvements based on Python's PEP 8. Gradually, we have used these linters to fix line length warnings, indentation, variable naming, and trailing whitespaces. That process showed how important the automated tools are in keeping the code clean, readable, and, thus, maintainable, since these characteristics are important for secure software development and effective collaboration among developers.

## Question 4:

* On sums.py. And sums2.py, what is the result?



### Run mccabe on sums.py. What is the output?

Running mccabe on sums.py gives the following output and informs us that the function test\_sum has a cyclomatic complexity of 1. That is to say, there is literally only one path through the code of this function-that is, there are no branches, loops, or conditionals of any sort. Indeed, the function is simple and linear, containing but a single assertion to test the sum of a list of numbers:

### Run mccabe on sums2.py. What is the result?

Running McCabe on sums2.py, we can see that test\_sum and test\_sum\_tuple each have a cyclomatic complexity of 1. That means that, in agreement with the observations in sums.py, both functions in sums2.py are straightforward and linear. Each has only an assertion statement inside it, but no branches or conditional flow, which explains the low complexity measurement.

### What is contributing to the cyclomatic complexity of each code example?

First, there is a cyclomatic complexity of 1 for every function in both sums.py and sums2.py; there are no complex control structures: no conditionals, loops, or other branching logic. Each function is purely linear: there is only one path of execution from start to finish. That is typical of test functions, since they are supposed to check output for one specific input without anything complicated.