AbdulMujeeb Ahmed and Malik Syed

Parag Tamhankar

Programming Learning Concepts

2 December 2024

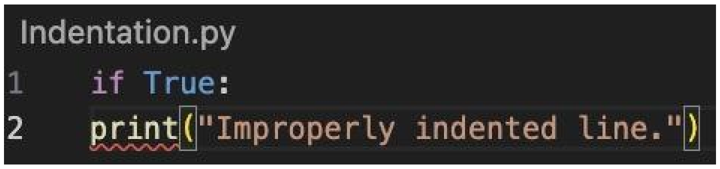
Final Project Report

For our final project, we decided to make a static analysis tool, otherwise known as a linter, in Python. This program allows users to see any syntax errors they have in their code. Our two main goals were to help enforce coding standards efficiently and in a way that is easy to read. We chose this topic because we wanted to understand how syntax is implemented and how to correct it. The key component of this project is the syntax enforcement. We coded a program that checks an example Python code for lines that exceed 80 characters, inconsistent indentation (mixing tabs and spaces), and general syntax errors like missing colons or parenthesis using AST.parse. We wanted to create this because it would help developers follow o coding standards better because the linter would provide feedback on code, ways to improvement the code as well as encourage better practices for Python programming.

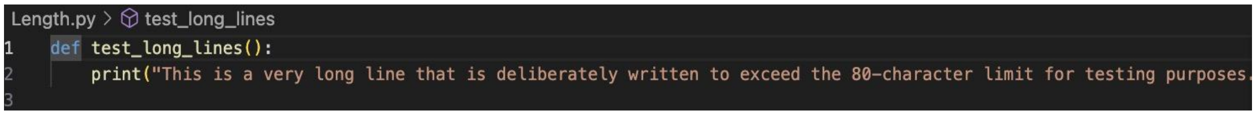
One main problem we faced while doing this program was the lack of accessible tools that provide detailed feedback on basic coding errors and adherence to style conventions in Python. While several linters exist, many are overly complex for beginners or lack the flexibility to be tailored for specific use cases. Common coding issues include lines exceeding standard length limits, inconsistent indentation - for example mixing spaces and tabs, and general syntax errors like missing colons or unmatched parentheses. These issues not only lead to runtime errors but also reduce code readability and maintainability. Our goal was to design a customizable linter that efficiently identifies these problems, offering a practical tool for developers to refine their Python code.

Our linter was designed with extensibility in mind. The core functionalities are implemented as methods within a single class, allowing for easy maintenance and future enhancements. The linter accepts one or multiple Python files for analysis and handles file input errors easily, such as missing or inaccessible files. It divides issues into categories—long_lines, indentation, and general syntax errors—and allows users to specify which classifications to check, offering a high degree of customization. The analysis uses two complementary approaches firstly line-by-line analysis and secondly Abstract Syntax Tree (AST)-based parsing. The first approach which is line-based analysis identifies issues such as exceeding line length limits and mixed indentation. The second approach, AST-based analysis parses the code to identify the structural problems like missing block indentation and syntax errors. Detected problems are categorized by severity levels that are identified as low, medium and high this provides users with a prioritized list of problems.

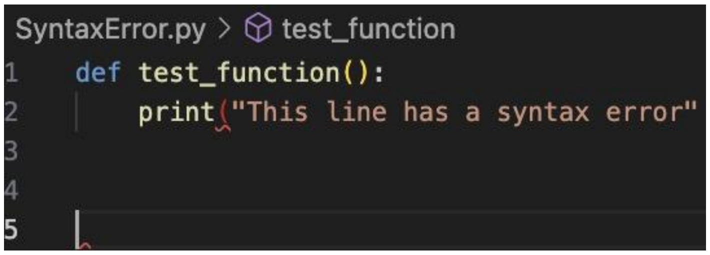
Below are some screenshots and images of our testing example:



```
Indentation.py
if True:
print("Improperly indented line.")
```

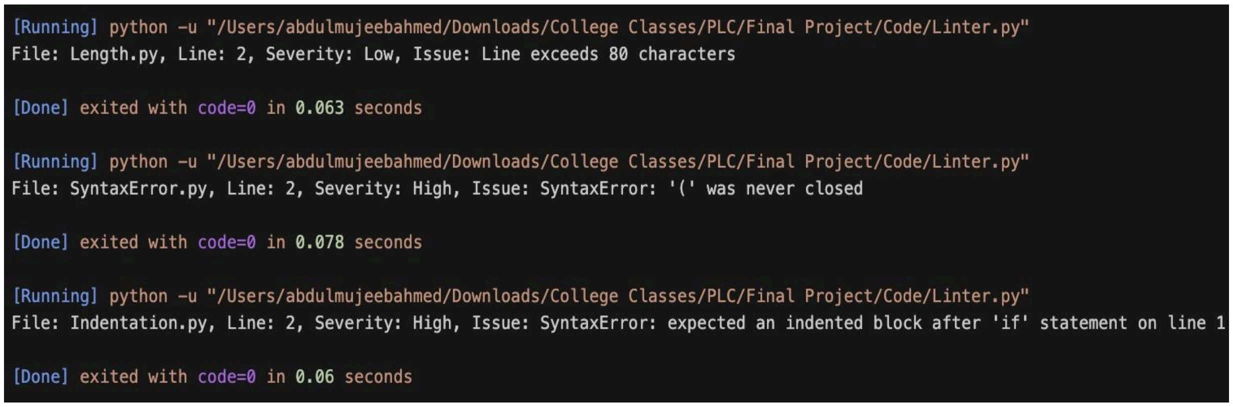


```
Length.py > test_long_lines
def test_long_lines():
    print("This is a very long line that is deliberately written to exceed the 80-character limit for testing purposes.
```



```
SyntaxError.py > test_function
def test_function():
    print("This line has a syntax error"
```

Here is a picture of our output:



```
[Running] python -u "/Users/abdulmujeebahmed/Downloads/College Classes/PLC/Final Project/Code/Linter.py"
File: Length.py, Line: 2, Severity: Low, Issue: Line exceeds 80 characters

[Done] exited with code=0 in 0.063 seconds

[Running] python -u "/Users/abdulmujeebahmed/Downloads/College Classes/PLC/Final Project/Code/Linter.py"
File: SyntaxError.py, Line: 2, Severity: High, Issue: SyntaxError: '(' was never closed

[Done] exited with code=0 in 0.078 seconds

[Running] python -u "/Users/abdulmujeebahmed/Downloads/College Classes/PLC/Final Project/Code/Linter.py"
File: Indentation.py, Line: 2, Severity: High, Issue: SyntaxError: expected an indented block after 'if' statement on line 1

[Done] exited with code=0 in 0.06 seconds
```

The tool's implementation leverages Python’s built-in libraries, primarily os for file handling and AST for syntax tree parsing. To detect lines exceeding length limits, the linter checks each line's character count and flags lines exceeding the user-defined maximum length of 80 characters. For inconsistent indentation, the program examines the leading whitespace of each line to detect the mixing of spaces and tabs. The linter uses AST parsing to detect structural syntax errors and missing block indentation in constructs such as if statements and function definitions. The linter also includes error-handling mechanisms that allow it to continue analyzing other files even when critical errors occur, providing feedback for issues like missing files or unreadable content.

This project aligns with several key programming language concepts. The design is object-oriented, encapsulating functionality within the linter class. The program uses error handling to ensure stability, easily managing exceptions such as syntax errors and missing files. The use of AST parsing demonstrates how static code analysis can go beyond text-based checks to understand code structure. Additionally, the modularity of the design allows developers to add new features, enhancing the tool’s versatility easily.

While our linter is functional and effective, several potential enhancements must be considered. Expanding its ability to analyze directories or multiple files simultaneously would improve its utility for larger projects. Enhanced reporting features, such as detailed summaries with visual elements like graphs, could make the feedback more user-friendly. Sorting detected issues by severity or type would enable users to prioritize their corrections so they know which ones are more urgent. Integration with popular IDEs, such as Visual Studio Code or PyCharm, would provide real-time feedback, making the linter a great tool for developers.

In conclusion, this linter project effectively combines the principles of object-oriented programming, error handling, and AST parsing to provide a reliable tool for Python static analysis. Addressing common coding issues helps developers write cleaner, more maintainable code. Our modular and extensible design ensures that the tool can grow with user needs, and its simplicity makes it accessible to developers of all skill levels. With potential enhancements such as directory-level analysis and IDE integration, this tool can become a valuable asset in Python development workflows.