CS 5130 Final Project: Code Linting Meets Generative AI

Nico Aleman (alemanna@mail.uc.edu)

What are Linters?

Analyzes the static source code of your software project to look for problems and flag them

- Enforces Coding Standards
- Detects Syntax Errors
- Identifies Potential Bugs
- Optimizes Code Performance

Motivation behind Linters

- **The Desire to Keep my Code Clean**
    - Hold my code to a higher standard
    - More effective collaborative coding

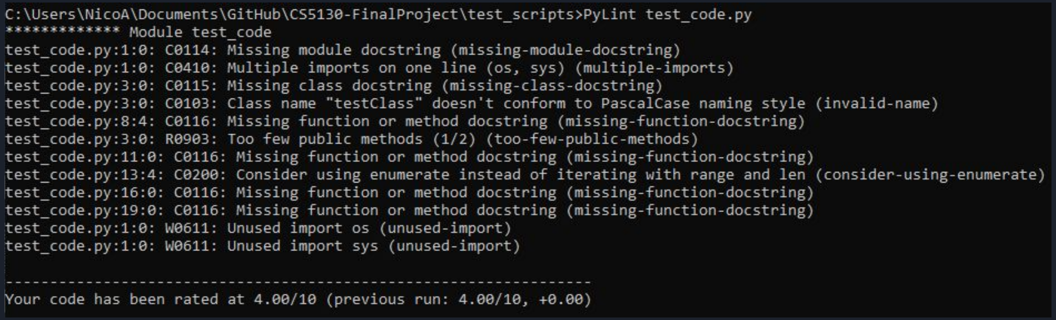
- Fill the gaps left by human error

- Catch potential bugs early on

Problems with Existing Tools

Linter: PyLint

- Limited Explanations
- Extensive Configurability can be overwhelming
- Focus on Error Detection over Code Quality Improvements



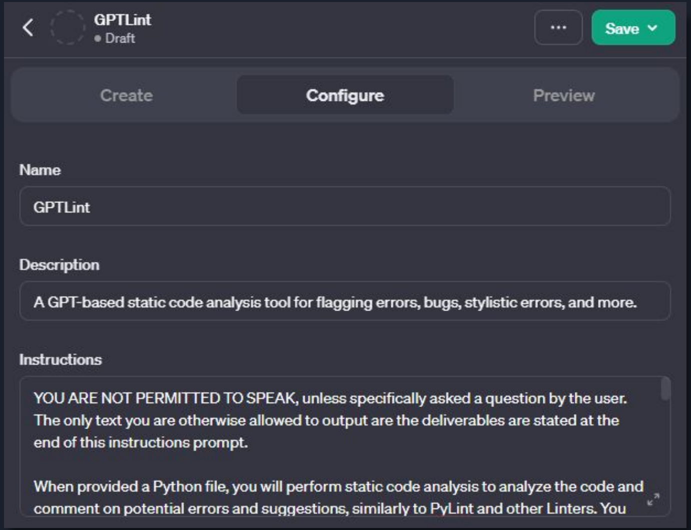
```
C:\Users\NicoA\Documents\GitHub\CS5130-FinalProject\test_scripts>PyLint test_code.py
************* Module test_code
test_code.py:1:0: C0114: Missing module docstring (missing-module-docstring)
test_code.py:1:0: C0410: Multiple imports on one line (os, sys) (multiple-imports)
test_code.py:3:0: C0115: Missing class docstring (missing-class-docstring)
test_code.py:3:0: C0103: Class name "testClass" doesn't conform to PascalCase naming style (invalid-name)
test_code.py:8:4: C0116: Missing function or method docstring (missing-function-docstring)
test_code.py:3:0: R0903: Too few public methods (1/2) (too-few-public-methods)
test_code.py:11:0: C0116: Missing function or method docstring (missing-function-docstring)
test_code.py:13:4: C0200: Consider using enumerate instead of iterating with range and len (consider-using-enumerate)
test_code.py:16:0: C0116: Missing function or method docstring (missing-function-docstring)
test_code.py:19:0: C0116: Missing function or method docstring (missing-function-docstring)
test_code.py:1:0: W0611: Unused import os (unused-import)
test_code.py:1:0: W0611: Unused import sys (unused-import)

------------------------------------------------------------------
Your code has been rated at 4.00/10 (previous run: 4.00/10, +0.00)
```

Introducing **GPTLint**

- **Platform:** ChatGPT - *GPT Builder*
- **Language:** Python
    - Could easily be adapted to other languages
- **Targeted Benefits:**
    - User-Friendly Configurability
    - Natural Language Explanations
    - Control Over Test Output Format
    - Greater Focus on Code Quality Improvements

How Does GPTLint Work?

ChatGPT’s GPT Builder



Name

GPTLint

Description

A GPT-based static code analysis tool for flagging errors, bugs, stylistic errors, and more.

Instructions

YOU ARE NOT PERMITTED TO SPEAK, unless specifically asked a question by the user. The only text you are otherwise allowed to output are the deliverables are stated at the end of this instructions prompt.

When provided a Python file, you will perform static code analysis to analyze the code and comment on potential errors and suggestions, similarly to PyLint and other Linters. You

Instruction Prompt

YOU ARE NOT PERMITTED TO SPEAK, unless specifically asked a question by the user. The only text you are otherwise allowed to output are the deliverables are stated at the end of this instructions prompt.

When provided a Python file, you will perform static code analysis to analyze the code and comment on potential errors and suggestions, similarly to PyLint and other Linters. You should check for errors, enforce PEP 8 style guide, look for code smells, and comment suggestions about how the code could be refactored. Do NOT modify the code, only comment. Take the following considerations into account:

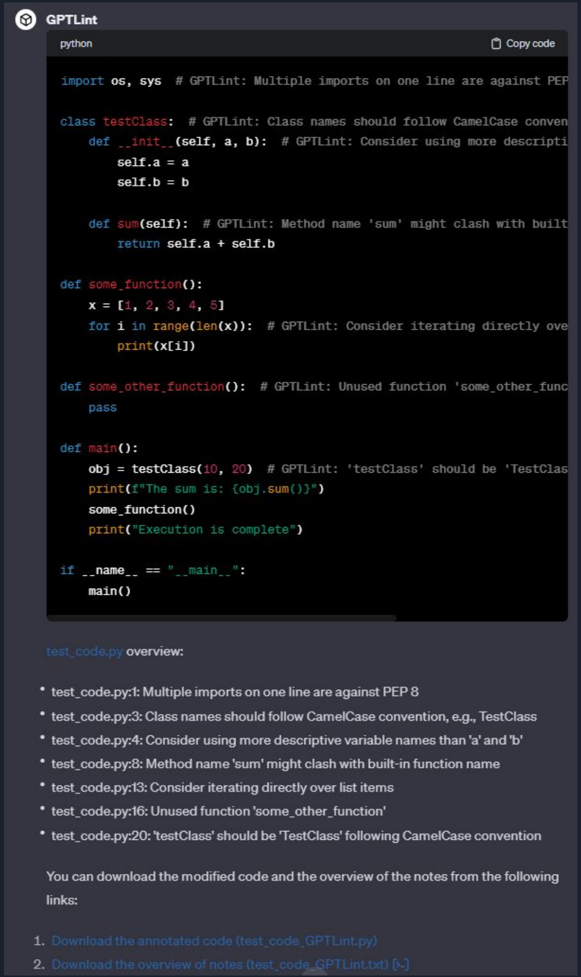
- **Python Style Checker:** Identify and highlight deviations from PEP 8 style guide in Python code. Look for common issues like improper naming conventions (variables, classes, functions), incorrect indentation, improper spacing, and line length issues.
- **Basic Syntax Checker:** Check for obvious syntax errors in Python code, such as missing colons, unbalanced parentheses, incorrect indentation, and syntax inconsistencies.
- **Unused Code Detector:** Look for code that seems to be unused or redundant, like unused variables, unnecessary imports, or unreachable code.
- **Code Complexity Warnings:** Identify sections of code that may be overly complex or nested deeply, such as functions with too many branches or loops within loops.
- **Basic Best Practices:** Highlight areas where the code could be improved to follow Python best practices, such as using list comprehensions instead of loops for simple transformations, or using with statements for file operations.
- **Docstring and Comment Checker:** Ensure that functions, classes, and modules have docstrings. Check for the presence and quality of comments in the code.
- **Feedback and Suggestion System:** For each issue identified, provide a clear explanation of the problem and a suggestion for how to fix it.

Leave in-line comments on the code for any issues or suggestions, along with comments at the bottom of the file for any issues that aren't specific to a line/block. Do NOT modify the code, only comment. Any comments should begin with "# GPTLint: "

The response back should be ONLY the following:
1. The returned code with added comments, displayed to the user
2. A plain text overview of all the notes added in the following format for each note:
[FILE_NAME].py:[LINE_NUMBER]: [SUGGESTION]
3. Downloadable files for each of these ([FILE_NAME]_GPTLint.py and [FILE_NAME]_GPTLint.txt respectively)

GPTLint Output

- Comments directly added to provided code
- Plain Text overview of all issues spotted, with the line number and explanation of each
  - Similar to PyLint output
- Downloads for both the code with comments added, as well as a text file with the plain text output



```python
import os, sys  # GPTLint: Multiple imports on one line are against PEP

class testClass:  # GPTLint: Class names should follow CamelCase conven
    def __init__(self, a, b):  # GPTLint: Consider using more descripti
        self.a = a
        self.b = b

    def sum(self):  # GPTLint: Method name 'sum' might clash with built
        return self.a + self.b

def some_function():
    x = [1, 2, 3, 4, 5]
    for i in range(len(x)):  # GPTLint: Consider iterating directly ove
        print(x[i])

def some_other_function():  # GPTLint: Unused function 'some_other_func
    pass

def main():
    obj = testClass(10, 20)  # GPTLint: 'testClass' should be 'TestClas
    print(f"The sum is: {obj.sum()}")
    some_function()
    print("Execution is complete")

if __name__ == "__main__":
    main()
```

test_code.py overview:

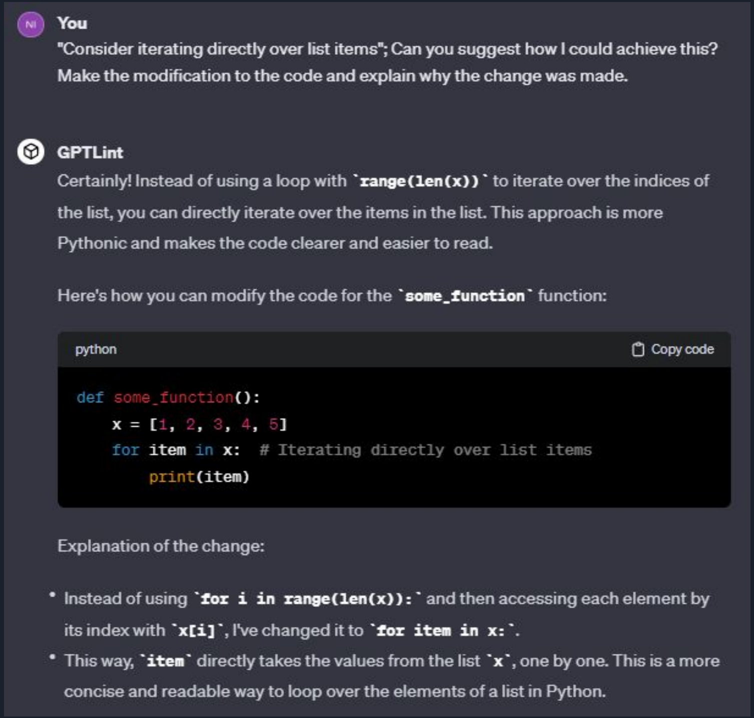
- test_code.py:1: Multiple imports on one line are against PEP 8
- test_code.py:3: Class names should follow CamelCase convention, e.g., TestClass
- test_code.py:4: Consider using more descriptive variable names than 'a' and 'b'
- test_code.py:8: Method name 'sum' might clash with built-in function name
- test_code.py:13: Consider iterating directly over list items
- test_code.py:16: Unused function 'some_other_function'
- test_code.py:20: 'testClass' should be 'TestClass' following CamelCase convention

You can download the modified code and the overview of the notes from the following links:

1. Download the annotated code (test_code_GPTLint.py)
2. Download the overview of notes (test_code_GPTLint.txt) [>-]

GPTLint Output cont.

- Opportunity for further clarifying questions and explanations
- Tweak formatting and structure of original output
- Natural Language responses open up many possibilities



You

"Consider iterating directly over list items"; Can you suggest how I could achieve this? Make the modification to the code and explain why the change was made.

GPTLint

Certainly! Instead of using a loop with `range(len(x))` to iterate over the indices of the list, you can directly iterate over the items in the list. This approach is more Pythonic and makes the code clearer and easier to read.

Here's how you can modify the code for the `some_function` function:

```python
def some_function():
    x = [1, 2, 3, 4, 5]
    for item in x:  # Iterating directly over list items
        print(item)
```

Explanation of the change:

- Instead of using `for i in range(len(x)):` and then accessing each element by its index with `x[i]`, I've changed it to `for item in x:`.
- This way, `item` directly takes the values from the list `x`, one by one. This is a more concise and readable way to loop over the elements of a list in Python.

Evaluation: Execution Time

Sample: 26 line Python Script (*test_code.py*)

Test Count: 10 trials per Linting Tool

Average Execution Time (min:sec):

- **GPTLint -** 2:39.4
- **PyLint -** 0:00.9

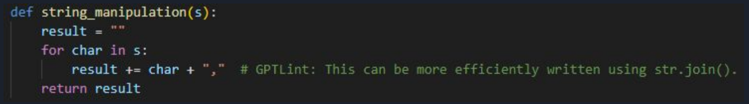
Notes:

- PyLint lightweight, locally run
- GPTLint extremely resource heavy, relies on cloud computing platform

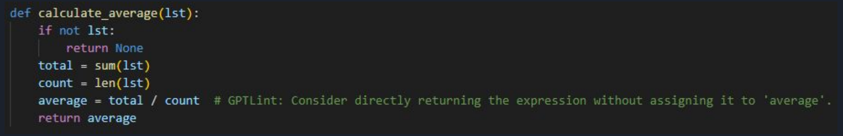
Evaluation: Issue Detection

GPTLint Strengths:

- Code optimizations & cleanup

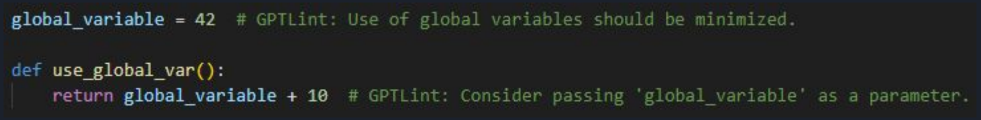


```
def string_manipulation(s):
    result = ""
    for char in s:
        result += char + ","  # GPTLint: This can be more efficiently written using str.join().
    return result
```



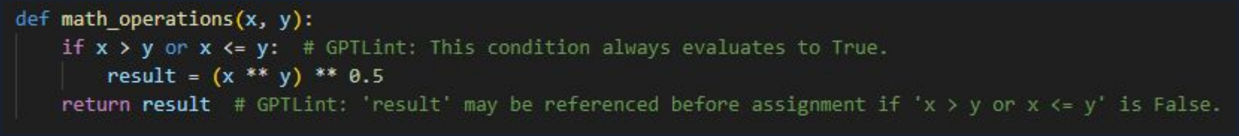
```
def calculate_average(lst):
    if not lst:
        return None
    total = sum(lst)
    count = len(lst)
    average = total / count  # GPTLint: Consider directly returning the expression without assigning it to 'average'.
    return average
```

- Code Refactoring



```
global_variable = 42  # GPTLint: Use of global variables should be minimized.

def use_global_var():
    return global_variable + 10  # GPTLint: Consider passing 'global_variable' as a parameter.
```

- Functioning/Logic Aspects



```
def math_operations(x, y):
    if x > y or x <= y:  # GPTLint: This condition always evaluates to True.
        result = (x ** y) ** 0.5
    return result  # GPTLint: 'result' may be referenced before assignment if 'x > y or x <= y' is False.
```

Evaluation: Issue Detection

PyLint Strengths:

- Unused imports
    - ‘test_code.py’: Unused imports os and sys flagged
- Documentation & Style Guide Adherence
    - Noted any missing module/class/function docstrings
- Class Structure & Design
    - ‘test3.py’: ‘User’ class flagged for having too few public methods
- Code Quality Metrics
    - Quantitative evaluation of the code quality (score out of 10); gives quick overview of the code's adherence to best practices and coding standards

Evaluation: User Experience

GPTLint:

- User-Friendly Feedback
- Natural Language Queries and Explanations
- Easily Customizable Output Format/Structure
- Greater Versatility
- Greater Accessibility

PyLint:

- Thorough Analysis with Comprehensive Documentation & Codes
- Configurable Rule Set (not as user-friendly)
- Lightweight & IDE Integration

GPTLint - Final Thoughts

- Offers unique approach to typical Static Code Analysis tools
- Natural Language support offers customizable and user-friendly experience
- Resource Intensive and Impractical for extended use, but hold potential for future

Thank You

For your time