Fix the linter and implement all the lint checks

Google Summer Of Code

Anshul Kumar Hudda

About Me

Why are you interested in working with Oppia, and on your chosen project?

Oppia's mission is to "provide high-quality education to those who lack access to it." And myself belonging to a farmer's family know the importance of education. I know how education can change lives. I'm very fascinated by Oppia's mission. It is a great learning experience to work with Oppia, I learned so much in just a few months of working with Oppia. I find the Oppia community to be super friendly. They helped me with every issue I face and are ready to guide me. It's been a great learning experience for me so far. I would continue to contribute to Oppia even after the GSoC period ends.

What interests me about this project?

I chose the linter project because I like to work on the backend part of the projects and also, it is written in Python. I always want to learn more about Python and I think it is a fun way to learn Python more deeply by working on a real world project.

The project aims to fix the linter and implement new lint checks. This project is really important for the following reasons:

I. Lint checks are important to reduce errors and improve the overall quality of your code.
II. Linting code increases the readability which makes the understanding code easier.
III. Linting code reduces the chances of syntax errors.

An open source organisation like Oppia needs the linting checks because the developers from around the world contribute to the oppia repository and everyone has a different coding style. Lint checks makes sure that they do not create a mess in the code and the next coder is easily able to read it.

Prior experience

I have been working with Python for the last two years. I am a backend developer and work primarily with Django. I have also been involved in Machine Learning since the past few months.

Some of my projects include:

- ❏ I made a blogging web app for adding posts for users using Django.
- ❏ I worked on a college project named 'Attendance management system' which is going to be live from next semester. This web app is made with Django and React and will be used for registering the attendance of the students.
- ❏ I created a bot for playing Nintendo's Mario using NEAT (Neuro Evolution of Augmented topologies) Algorithm as a college project.

Links to PRs and Issues to Oppia

- PR #8656 (Split the linter to run for a specific file extension.)
- PR #8947 (Addition of Python, HTML, CSS, Js/Ts lint tests.)
- PR #7929 (Added lint check for a newline above args in python doc string.)
- PR #8066 (Custom pylint extension to ensure that comments follow correct format.)
- PR #8034 (Custom pylint extension for checking division operator)

Here is the complete set of PRs opened by me.

Issues opened by me:

- Issue #8625

I am also helping fellow contributors by reviewing their PRs.

Contact info and timezone(s)

I will be in New Delhi (India) throughout the summer (Timezone: UTC+05:30)

Communication

- Email: anshulhudda.ssap@gmail.com
- Phone: +91 8700215154
- Github: @Hudda

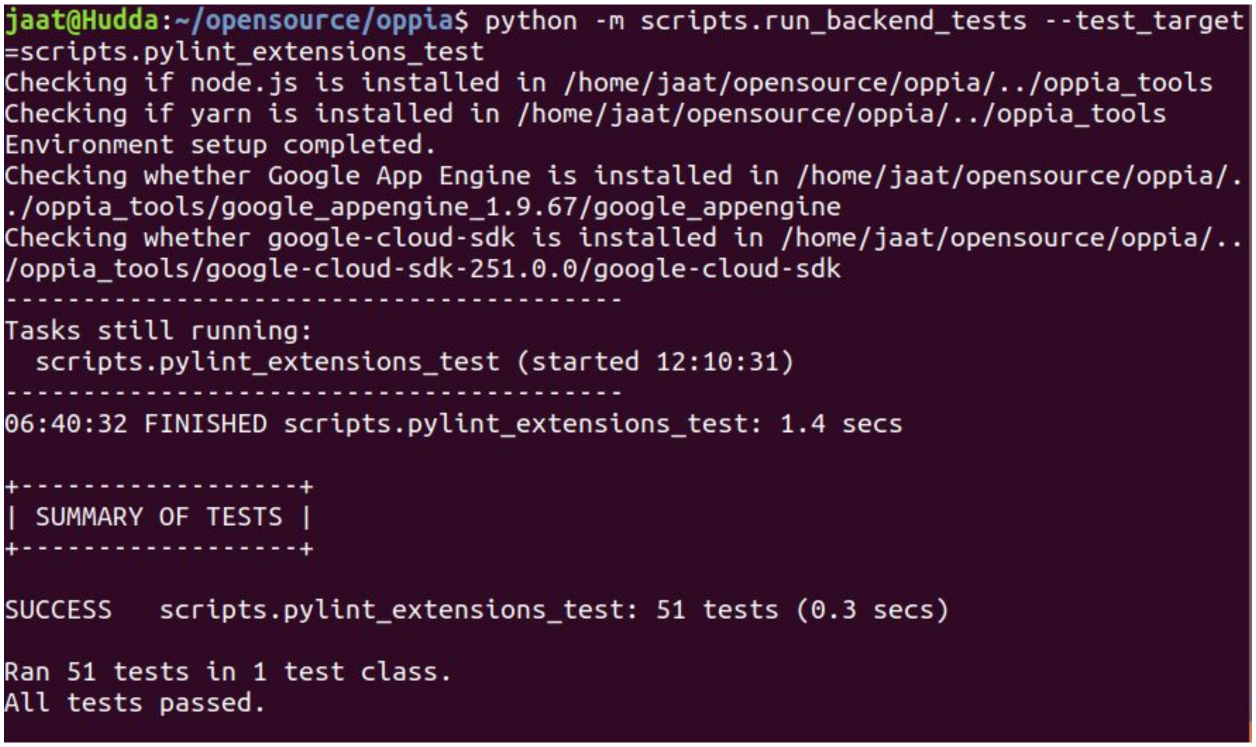
I will be in continuous touch with the mentors via email, Gitter or Hangouts. There could be biweekly (or as preferred by the mentors) meetings on Hangouts to discuss the workflow to be followed ahead.

Time commitment

I will spend 55-60 hours per week contributing.

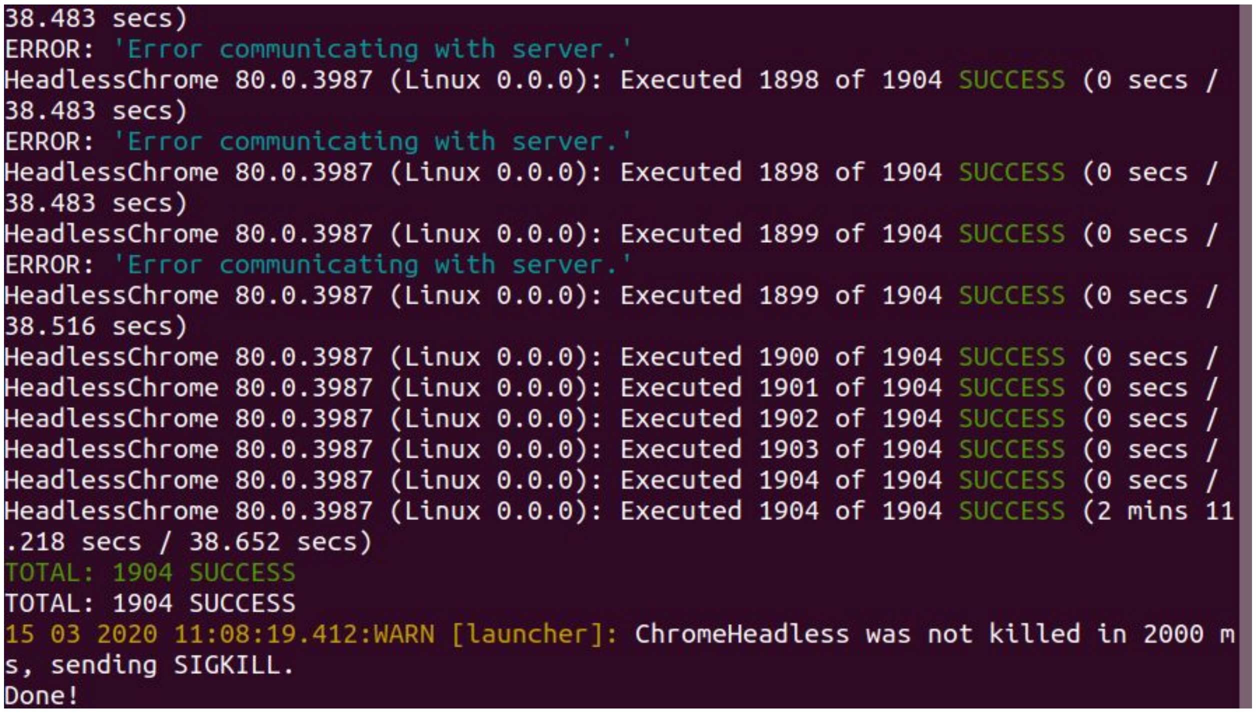
Essential Prerequisites

- *I am able to run a single backend test target on my machine.*



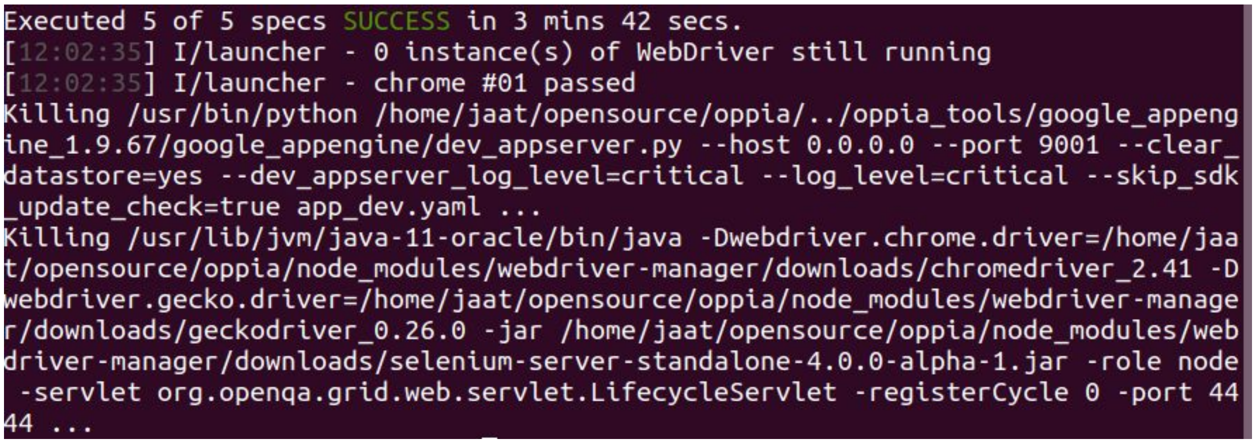
```
jaat@Hudda:~/opensource/oppia$ python -m scripts.run_backend_tests --test_target
=scripts.pylint_extensions_test
Checking if node.js is installed in /home/jaat/opensource/oppia/../oppia_tools
Checking if yarn is installed in /home/jaat/opensource/oppia/../oppia_tools
Environment setup completed.
Checking whether Google App Engine is installed in /home/jaat/opensource/oppia/.
./oppia_tools/google_appengine_1.9.67/google_appengine
Checking whether google-cloud-sdk is installed in /home/jaat/opensource/oppia/..
/oppia_tools/google-cloud-sdk-251.0.0/google-cloud-sdk
----------------------------------------
Tasks still running:
  scripts.pylint_extensions_test (started 12:10:31)
----------------------------------------
06:40:32 FINISHED scripts.pylint_extensions_test: 1.4 secs

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   scripts.pylint_extensions_test: 51 tests (0.3 secs)

Ran 51 tests in 1 test class.
All tests passed.
```

- *I am able to run all the frontend tests at once on my machine.*



```
38.483 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1898 of 1904 SUCCESS (0 secs /
38.483 secs)
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1898 of 1904 SUCCESS (0 secs /
38.483 secs)
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1899 of 1904 SUCCESS (0 secs /
ERROR: 'Error communicating with server.'
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1899 of 1904 SUCCESS (0 secs /
38.516 secs)
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1900 of 1904 SUCCESS (0 secs /
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1901 of 1904 SUCCESS (0 secs /
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1902 of 1904 SUCCESS (0 secs /
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1903 of 1904 SUCCESS (0 secs /
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1904 of 1904 SUCCESS (0 secs /
HeadlessChrome 80.0.3987 (Linux 0.0.0): Executed 1904 of 1904 SUCCESS (2 mins 11
.218 secs / 38.652 secs)
TOTAL: 1904 SUCCESS
TOTAL: 1904 SUCCESS
15 03 2020 11:08:19.412:WARN [launcher]: ChromeHeadless was not killed in 2000 m
s, sending SIGKILL.
Done!
```

- *I am able to run one suite of e2e tests on my machine.*



```
Executed 5 of 5 specs SUCCESS in 3 mins 42 secs.
[12:02:35] I/launcher - 0 instance(s) of WebDriver still running
[12:02:35] I/launcher - chrome #01 passed
Killing /usr/bin/python /home/jaat/opensource/oppia/../oppia_tools/google_appeng
ine_1.9.67/google_appengine/dev_appserver.py --host 0.0.0.0 --port 9001 --clear_
datastore=yes --dev_appserver_log_level=critical --log_level=critical --skip_sdk
_update_check=true app_dev.yaml ...
Killing /usr/lib/jvm/java-11-oracle/bin/java -Dwebdriver.chrome.driver=/home/jaa
t/opensource/oppia/node_modules/webdriver-manager/downloads/chromedriver_2.41 -D
webdriver.gecko.driver=/home/jaat/opensource/oppia/node_modules/webdriver-manage
r/downloads/geckodriver_0.26.0 -jar /home/jaat/opensource/oppia/node_modules/web
driver-manager/downloads/selenium-server-standalone-4.0.0-alpha-1.jar -role node
 -servlet org.openqa.grid.web.servlet.LifecycleServlet -registerCycle 0 -port 44
44 ...
```

Other summer obligations

I have no other summer obligations.

Communication channels

I am comfortable with all the modes of communication, be it Gitter or Hangouts or Whatsapp and am willing to choose any mode used by the mentors.

Project Details

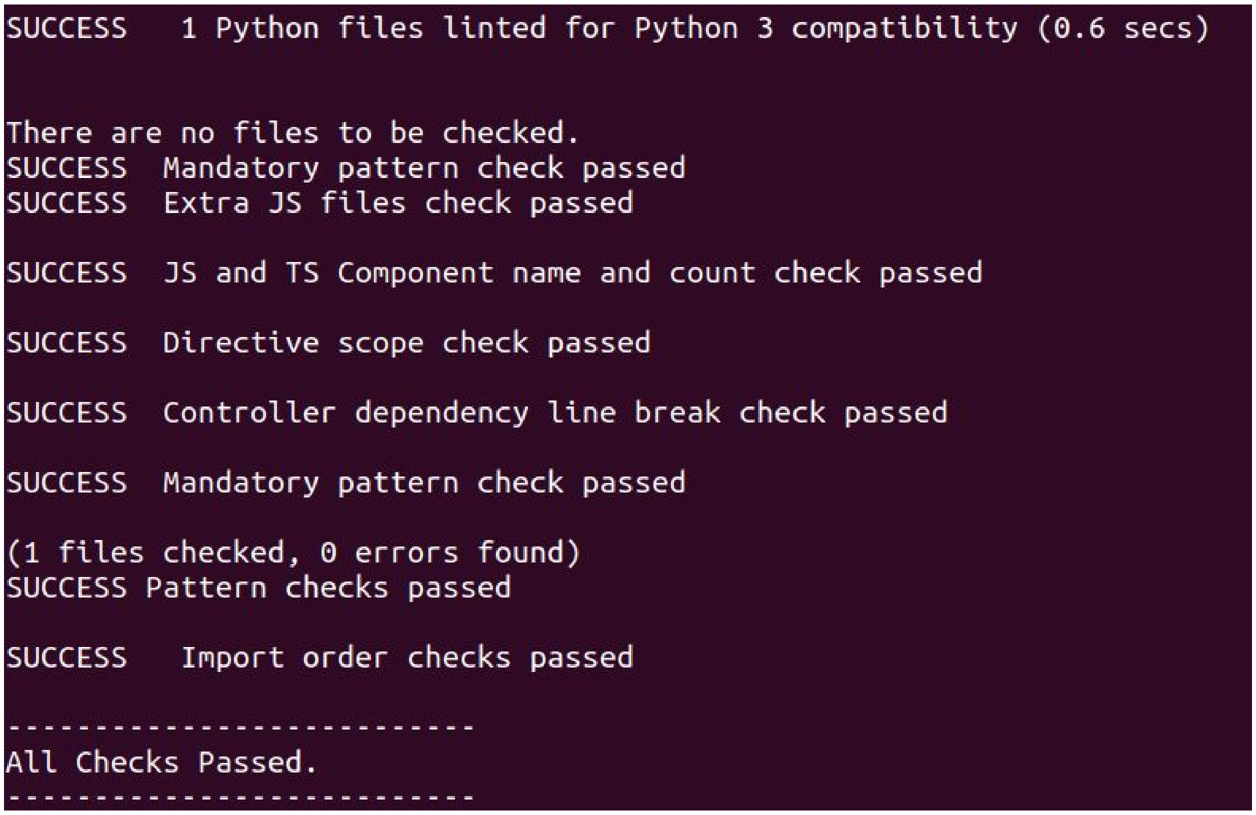
Product Design

This project will help developers on the Oppia team in writing clean code with fewer syntax errors and good readability. Linting is used to detect style errors in the codebase and for writing cleaner code so that everyone can understand it easily.

Catching errors using exception handling

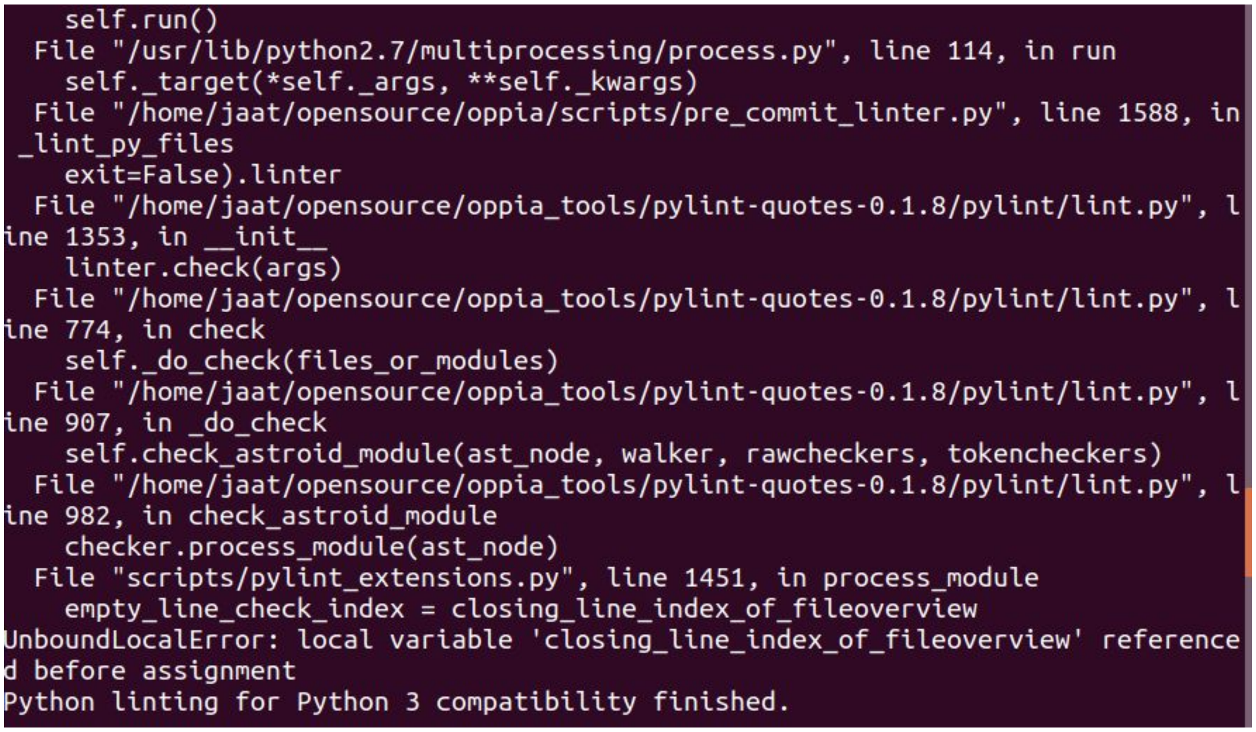
The Oppia development workflow uses lint checks to help detect style errors before they reach the review phase. But error detection for the linter itself is missing. There had been several cases in the past where the linter happily let everything through, one time it was '*ASCII encoding*' error and one time it was '*variable referenced before initialization*' error which caused the **Pylint** to stop working, and this was not detected until several months. During this period a lot of lint errors build up and we had to spend some time fixing those errors. To make sure this won't happen again. I will add exception handling to the linter so that if any unexpected error happens it gets caught and linter raises an exception.

Let's see an example of *'variable referenced before initialization'* error. Suppose there are several changes made by a developer in the codebase. Developer run linter to check if there are any lint errors in his/her code. This is what he/she sees(see screenshot below) when the linter finishes linting.



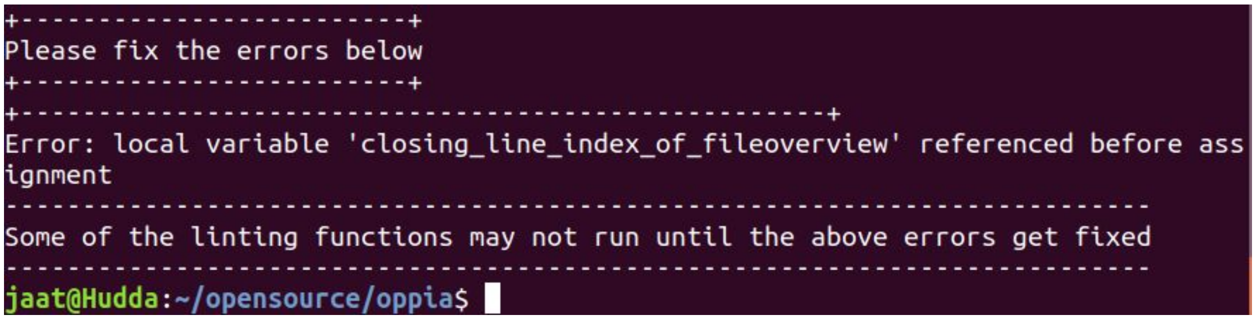
```
SUCCESS   1 Python files linted for Python 3 compatibility (0.6 secs)


There are no files to be checked.
SUCCESS  Mandatory pattern check passed
SUCCESS  Extra JS files check passed

SUCCESS  JS and TS Component name and count check passed

SUCCESS  Directive scope check passed

SUCCESS  Controller dependency line break check passed

SUCCESS  Mandatory pattern check passed

(1 files checked, 0 errors found)
SUCCESS Pattern checks passed

SUCCESS   Import order checks passed

---------------------------
All Checks Passed.
---------------------------
```

Developer think that his/her code is free of lint errors as all checks are passed. But here comes the twist, if we scroll a little up, we found that an unexpected error has happened which linter did not report(see screenshot below) and neither stops the code execution.



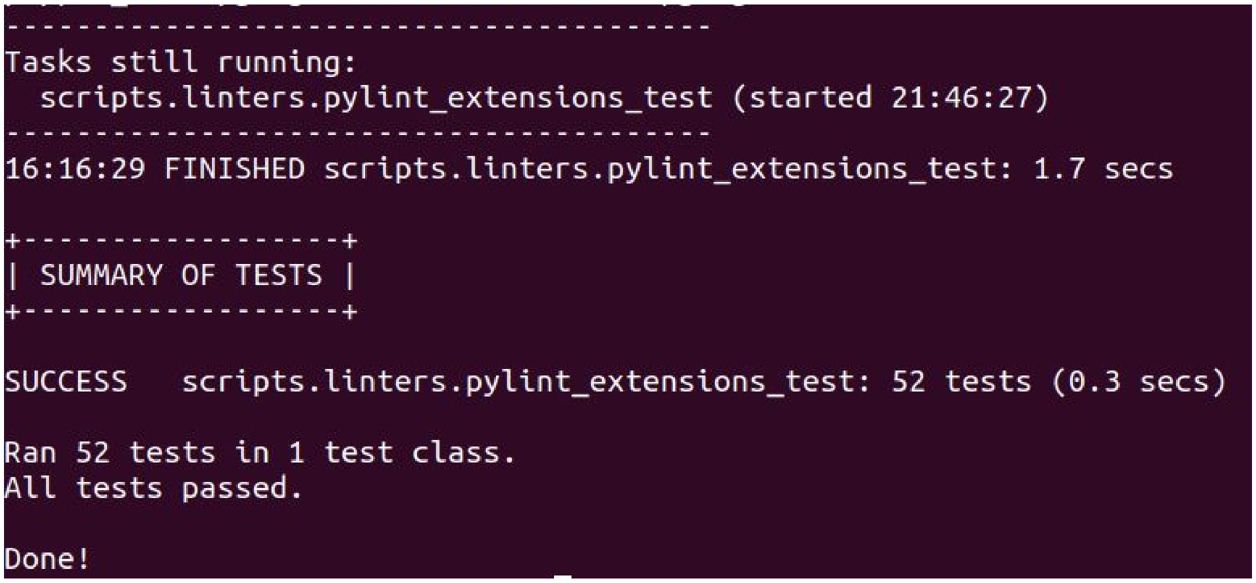
```
    self.run()
  File "/usr/lib/python2.7/multiprocessing/process.py", line 114, in run
    self._target(*self._args, **self._kwargs)
  File "/home/jaat/opensource/oppia/scripts/pre_commit_linter.py", line 1588, in
 _lint_py_files
    exit=False).linter
  File "/home/jaat/opensource/oppia_tools/pylint-quotes-0.1.8/pylint/lint.py", l
ine 1353, in __init__
    linter.check(args)
  File "/home/jaat/opensource/oppia_tools/pylint-quotes-0.1.8/pylint/lint.py", l
ine 774, in check
    self._do_check(files_or_modules)
  File "/home/jaat/opensource/oppia_tools/pylint-quotes-0.1.8/pylint/lint.py", l
ine 907, in _do_check
    self.check_astroid_module(ast_node, walker, rawcheckers, tokencheckers)
  File "/home/jaat/opensource/oppia_tools/pylint-quotes-0.1.8/pylint/lint.py", l
ine 982, in check_astroid_module
    checker.process_module(ast_node)
  File "scripts/pylint_extensions.py", line 1451, in process_module
    empty_line_check_index = closing_line_index_of_fileoverview
UnboundLocalError: local variable 'closing_line_index_of_fileoverview' reference
d before assignment
Python linting for Python 3 compatibility finished.
```

These types of errors mostly remain hidden in the verbose output of the linter and difficult to catch with just simple testing. But these errors can be easily caught with exception handling. With exception handling we can catch all the unexpected errors and print them at the end of linter output, so that the developer should be able to fix them. When I am done implementing exception handling we will have this(see screenshot below) type of warning at the end of code execution.



```
+------------------------+
Please fix the errors below
+------------------------+
+---------------------------------------------------+
Error: local variable 'closing_line_index_of_fileoverview' referenced before ass
ignment
----------------------------------------------------------------------
Some of the linting functions may not run until the above errors get fixed
----------------------------------------------------------------------
jaat@Hudda:~/opensource/oppia$
```

One can say why not just write tests for catching errors. But it is not possible to write tests for all cases, a developer isn't always able to think of all cases where errors might happen. Also, tests are not able to catch unexpected errors. For example, in the above example we have tests written for that piece of code which is causing error but still the error passes through it, because the developer who wrote that piece of code did not take the case in consideration where the file is empty(__init__.py). All the tests were passing in this case(see screenshot below) but still error was there.

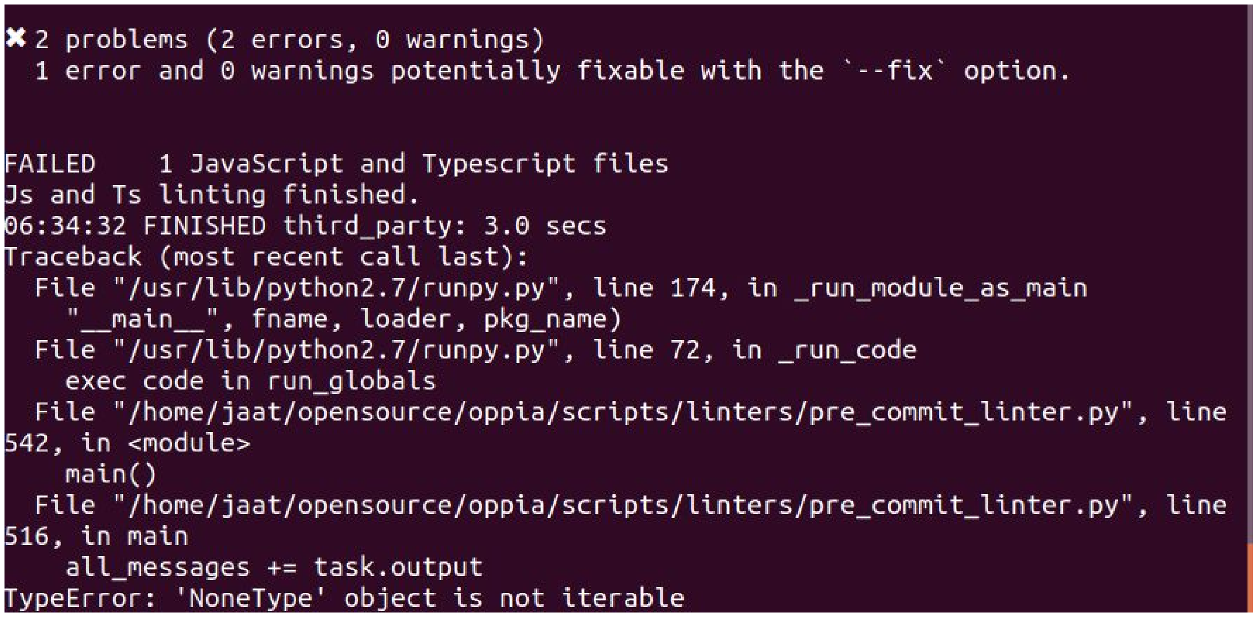


```
----------------------------------------
Tasks still running:
  scripts.linters.pylint_extensions_test (started 21:46:27)
----------------------------------------
16:16:29 FINISHED scripts.linters.pylint_extensions_test: 1.7 secs

+------------------+
| SUMMARY OF TESTS |
+------------------+

SUCCESS   scripts.linters.pylint_extensions_test: 52 tests (0.3 secs)

Ran 52 tests in 1 test class.
All tests passed.

Done!
```

For cases like this, we need exception handling because by using exception handling we have an advantage of catching unexpected errors which is not possible in the case of writing just tests. That's why I will implement exception handling along with tests for extra protection. So, if developers leave any case in tests or any unexpected error happens it will be caught by exception handling and all the error messages should be reported at the end of linter output.

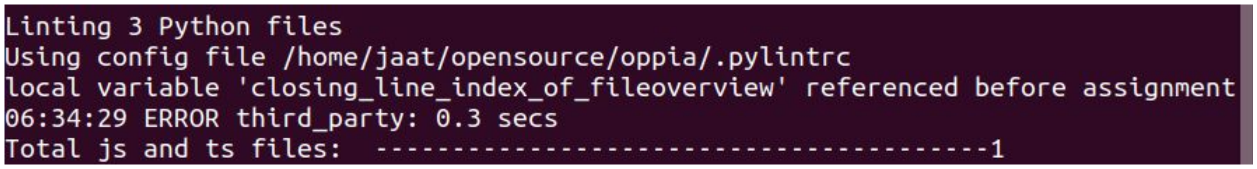
Why do we need exception Handling?

- Exception handling is not for the lint checks but for any bug in the linting script itself. The tests only check the behaviour of lint check. But there might be some logical error in the script which might not be caught by the tests.
- If we do not use exception handling, the linter still fails but the error is harder to catch as the error log is buried somewhere inside the output and not at the end of lint output. Example: Here linter fails but we are not seeing the main cause of the error because it is buried in the output.



```
✖ 2 problems (2 errors, 0 warnings)
  1 error and 0 warnings potentially fixable with the `--fix` option.


FAILED    1 JavaScript and Typescript files
Js and Ts linting finished.
06:34:32 FINISHED third_party: 3.0 secs
Traceback (most recent call last):
  File "/usr/lib/python2.7/runpy.py", line 174, in _run_module_as_main
    "__main__", fname, loader, pkg_name)
  File "/usr/lib/python2.7/runpy.py", line 72, in _run_code
    exec code in run_globals
  File "/home/jaat/opensource/oppia/scripts/linters/pre_commit_linter.py", line
542, in <module>
    main()
  File "/home/jaat/opensource/oppia/scripts/linters/pre_commit_linter.py", line
516, in main
    all_messages += task.output
TypeError: 'NoneType' object is not iterable
```

This is the error which can be found in between the linter output:



```
Linting 3 Python files
Using config file /home/jaat/opensource/oppia/.pylintrc
local variable 'closing_line_index_of_fileoverview' referenced before assignment
06:34:29 ERROR third_party: 0.3 secs
Total js and ts files:  -------------------------------------1
```

As you can see, this is hard to see and therefore, hard to debug.

- As of now even after an exception is raised the linter continues to run. The linter continues running even if an exception is raised because we have many sub linters e.g. pylint, eslint, sylint and if any exception raised, it only stops that particular linter and other linter continues running. This is the reason we need exception handling.

Why do we need custom exceptions?

- The reason we need custom exceptions is the same as the above point. Only a sublinter is stopped running and the other linter continues running. The exception remains buried in the output and we may never know if something has happened at all.
- By using custom exception handling we can catch all the errors and show them to the user at the end of the linter output with a warning message.

Splitting of linter

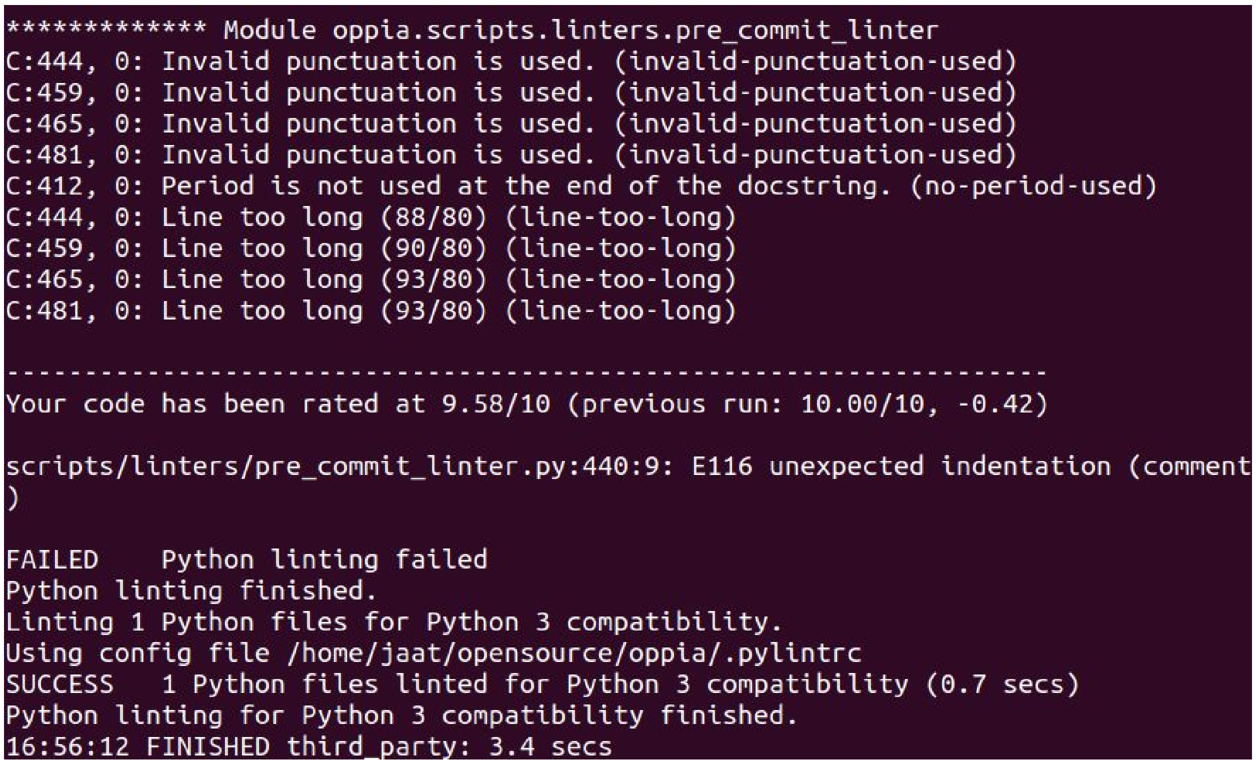
As of now, the linter is linting all the file types at once, but in #8656 I am working to separate linters according to their file extension type. So that we can have sub linters to the main linter and we can run a specific linter by using a command-line argument. This way we do not have to lint all the file types if we only want to lint files for a specific file type.

Adding tests for lint checks in ‘pre_commit_linter.py’

As of now almost all of the linter code remains untested, only pylint has tests written for its custom checks. I will add tests for the lint checks in *pre_commit_linter*. Adding tests will ensure that the lint checks are working as they meant to be. This will also ensure that any future lint checks do not remain untested and any new error will not be introduced. A small amount of work is already done in #8947.

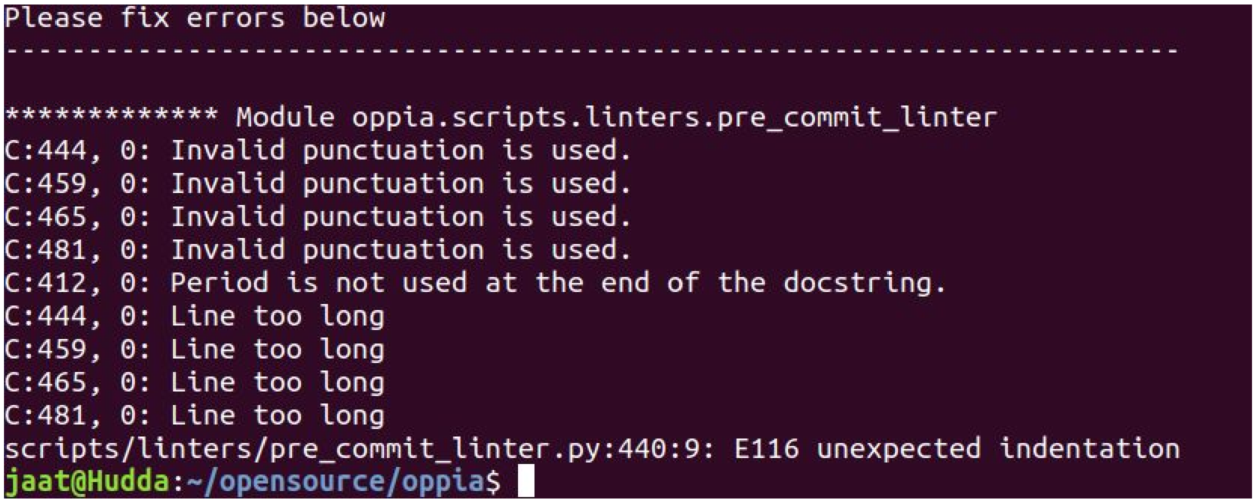
Make lint output less verbose

The verbose mode is outputting too much code right now and a large part of the output is generally are of no particular use(see screenshot below).



```
************* Module oppia.scripts.linters.pre_commit_linter
C:444, 0: Invalid punctuation is used. (invalid-punctuation-used)
C:459, 0: Invalid punctuation is used. (invalid-punctuation-used)
C:465, 0: Invalid punctuation is used. (invalid-punctuation-used)
C:481, 0: Invalid punctuation is used. (invalid-punctuation-used)
C:412, 0: Period is not used at the end of the docstring. (no-period-used)
C:444, 0: Line too long (88/80) (line-too-long)
C:459, 0: Line too long (90/80) (line-too-long)
C:465, 0: Line too long (93/80) (line-too-long)
C:481, 0: Line too long (93/80) (line-too-long)

------------------------------------------------------------------
Your code has been rated at 9.58/10 (previous run: 10.00/10, -0.42)

scripts/linters/pre_commit_linter.py:440:9: E116 unexpected indentation (comment
)

FAILED    Python linting failed
Python linting finished.
Linting 1 Python files for Python 3 compatibility.
Using config file /home/jaat/opensource/oppia/.pylintrc
SUCCESS   1 Python files linted for Python 3 compatibility (0.7 secs)
Python linting for Python 3 compatibility finished.
16:56:12 FINISHED third_party: 3.4 secs
```

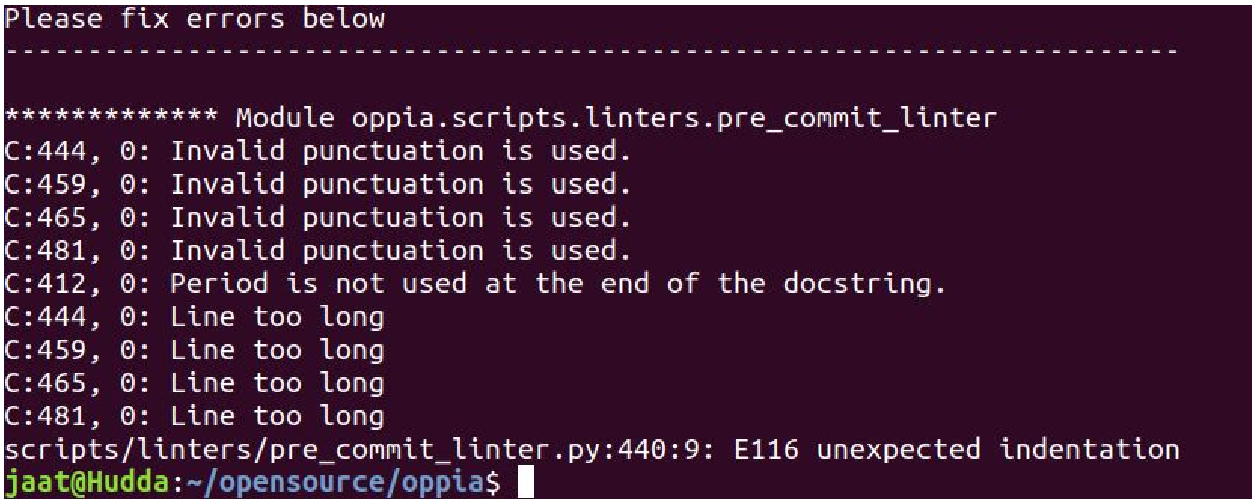
There is much extra information which is of no use and makes it hard to see the actual error. I will change the code so that only essential bits are shown by the linter. And summary of error messages will be shown at the end of the linter output under a line saying ‘*Please fix errors below*’. After I am done with implementing this feature the linter output will look like this(see screenshot below).



```
Please fix errors below
-----------------------------------------------------------------------

************* Module oppia.scripts.linters.pre_commit_linter
C:444, 0: Invalid punctuation is used.
C:459, 0: Invalid punctuation is used.
C:465, 0: Invalid punctuation is used.
C:481, 0: Invalid punctuation is used.
C:412, 0: Period is not used at the end of the docstring.
C:444, 0: Line too long
C:459, 0: Line too long
C:465, 0: Line too long
C:481, 0: Line too long
scripts/linters/pre_commit_linter.py:440:9: E116 unexpected indentation
jaat@Hudda:~/opensource/oppia$
```

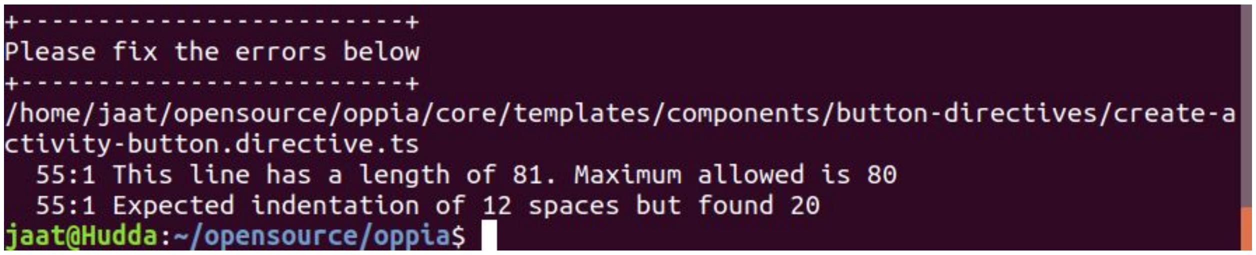
Mocks for the making lint output less verbose

Python



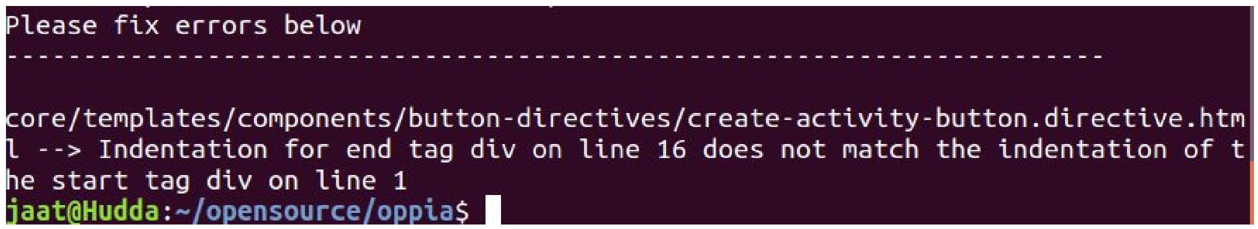
```
Please fix errors below
-----------------------------------------------------------------------

************* Module oppia.scripts.linters.pre_commit_linter
C:444, 0: Invalid punctuation is used.
C:459, 0: Invalid punctuation is used.
C:465, 0: Invalid punctuation is used.
C:481, 0: Invalid punctuation is used.
C:412, 0: Period is not used at the end of the docstring.
C:444, 0: Line too long
C:459, 0: Line too long
C:465, 0: Line too long
C:481, 0: Line too long
scripts/linters/pre_commit_linter.py:440:9: E116 unexpected indentation
jaat@Hudda:~/opensource/oppia$
```

JS/TS



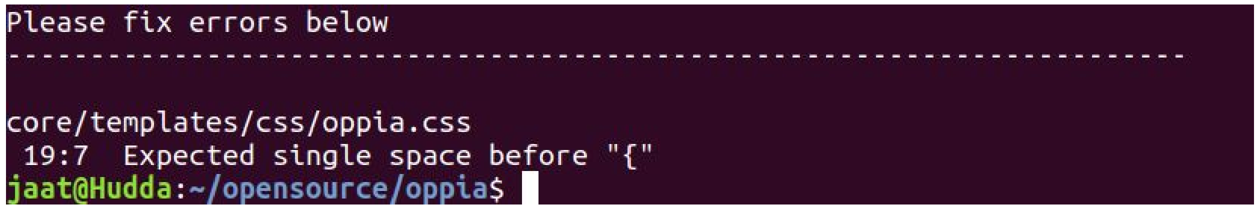
```
+------------------------+
Please fix the errors below
+------------------------+
/home/jaat/opensource/oppia/core/templates/components/button-directives/create-a
ctivity-button.directive.ts
  55:1 This line has a length of 81. Maximum allowed is 80
  55:1 Expected indentation of 12 spaces but found 20
jaat@Hudda:~/opensource/oppia$
```

HTML



```
Please fix errors below
-----------------------------------------------------------------------

core/templates/components/button-directives/create-activity-button.directive.htm
l --> Indentation for end tag div on line 16 does not match the indentation of t
he start tag div on line 1
jaat@Hudda:~/opensource/oppia$
```

CSS



```
Please fix errors below
-----------------------------------------------------------------------

core/templates/css/oppia.css
 19:7  Expected single space before "{"
jaat@Hudda:~/opensource/oppia$
```

In the screenshots above, all the extra bits which are of no use have been trimmed. This way the user sees only important information and extra clutter has been removed. Also, all the errors are at the end of lint output and hence it will be easy for developers to easily spot the lint errors.

Requirements of verbose and non-verbose mode

Verbose mode	Non-verbose mode
Need --verbose flag to enable verbose mode and will be mainly used in a CI environment.	Linter will run in non-verbose mode by default.
Will log error twice once when they are being caught and once at the end of linter output	Will log error only once at the end of linter output.
All the information will be shown.	Only important parts of the error message will be shown and extra bits will be removed.

Implementation of new lint checks

There are a number of lint checks which are not implemented yet(see issue #8423). I will implement these checks and write their test in their respective files. Also, I am going to fix all new lint errors which are going to pop up after the implementation of these new lint checks.

List of all lint checks with deadlines

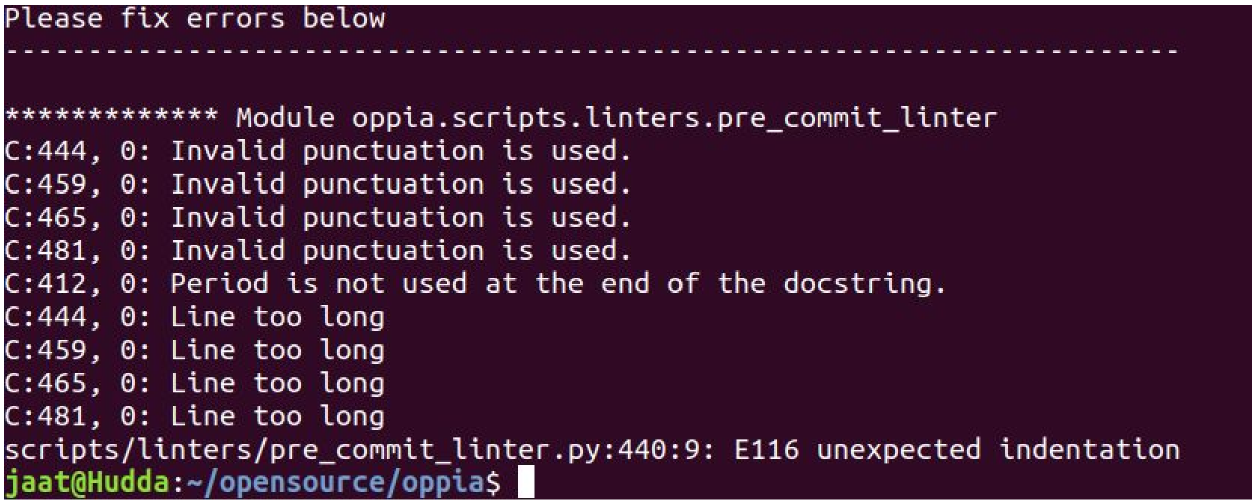
S.No.	Lint Checks	Deadline
General Lint Checks		
1.	Check to ensure that all lines in skip_files in app.yaml reference valid files in the repository	7/6/2020
2.	Newline check at end of file	9/6/2020
3.	Lint checks to ensure that there are valid spaces and newlines	11/6/2020
4.	Lint checks for webpack config	13/6/2020
5.	Check that all TODO comments start with capital letter	15/6/2020
6.	Check for proper comment style for Python and JS files	18/6/2020
7.	Check to ensure that every file (of any type) ends with exactly one newline character.	20/6/2020
CSS lint checks		
8.	Check for alphabetized list in CSS	23/6/2020
9.	Prohibit inline styling	25/6/2020
Python lint checks		
10.	Enforce multiples of 4 indentation in docstring	7/7/2020
11.	Check for proper comment indentation	8/7/2020
12.	Check for proper Args and Returns style	10/7/2020
13.	Check to multiline expressions ensuring line break after '('	12/7/2020
14.	Check for args-name for a non-keyword argument	14/7/2020
15.	Check to ensure that pylint pragmas are used to disable any rule for a single line	16/7/2020
16.	Lint check to ensure that there is one blank newline below each class docstring.	18/7/2020
17.	Check to detect variables that are declared but never used.	20/7/2020

18.	Forbid usage of assertRaises; require assertRaisesRegexp instead.	22/7/2020
19.	Enforce that PEP8 naming convention is followed.	24/7/2020
JS/TS lint checks		
20.	Forbid use of innerHTML due to security issues	24/7/2020
21.	Check for unused imports	25/7/2020
22.	Add a check to ensure that multi-line expressions in parenthesis are broken down after "(" in JavaScript files	2/7/2020
23.	Check for unused directive/service dependencies	4/8/2020
24.	Improve reachability of sorted imports	6/8/2020
25.	Check that filename is similar to service or directive name	8/8/2020
26.	Check for eslint-disable statements	10/8/2020
27.	Lint check to ensure require(…) statements are alphabetized	12/8/2020
28.	Lint check for test files	16/8/2020
29.	Detect variables that are defined but never used.	16/8/2020
30.	Add lint check to make sure there's a space before the opening braces of a function start.	18/8/2020
31.	There should be no space before "function"	20/8/2020
32.	Catch missing space after semicolon	22/8/2020
"All $http calls should be in backend-api files and prohibited elsewhere (#8039)" will be implemented after GSoC once the issue #8016 and #8038 are resolved.		

Final Product

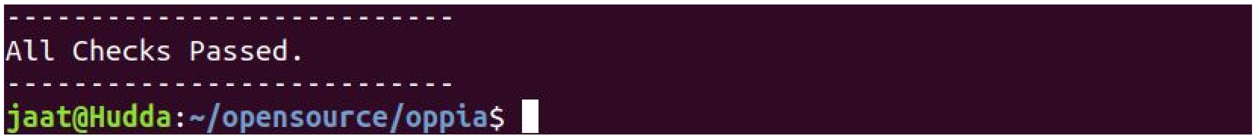
After I am done implementing these features we have a more robust linter with more lint checks. This will make the developer's life easy by identifying any syntax errors before the code reviewing phase.

In the end we have these(see screenshot below) types of lint errors if we made any style error and pushing will be blocked until we fix all the lint errors.



```
Please fix errors below
----------------------------------------------------------------------

************* Module oppia.scripts.linters.pre_commit_linter
C:444, 0: Invalid punctuation is used.
C:459, 0: Invalid punctuation is used.
C:465, 0: Invalid punctuation is used.
C:481, 0: Invalid punctuation is used.
C:412, 0: Period is not used at the end of the docstring.
C:444, 0: Line too long
C:459, 0: Line too long
C:465, 0: Line too long
C:481, 0: Line too long
scripts/linters/pre_commit_linter.py:440:9: E116 unexpected indentation
jaat@Hudda:~/opensource/oppia$
```

If all tests are passing we have these types of success messages(see screenshot below).



```
---------------------------
All Checks Passed.
---------------------------
jaat@Hudda:~/opensource/oppia$
```

Technical Design

Architectural Overview

Refactoring of code will be done in **oppia/scripts/linters/pre_commit_linter.py**. All the code for catching exceptions and making output less verbose will go in this file.

New lint checks for the different file types will go in following files under **oppia/scripts/linters** directory:

1. **Python**
   1.1. *pylint_extensions.py*
   1.2. *python_linter.py* (If a third party check like isort needed to be used)
   1.3. *general_purpose_linter.py* (If need to check for bad patterns in files)

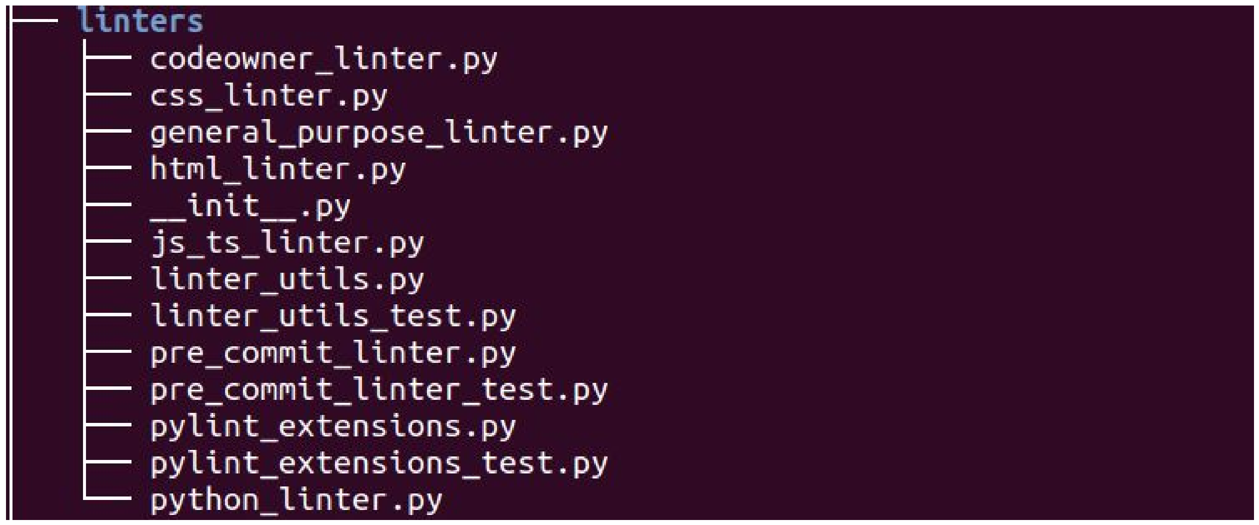
2. **Javascript and Typescript**
   2.1. *Js_ts_linter.py*
   2.2. *general_purpose_linter.py* (If need to check for bad patterns in files)

3. **HTML**

    3.1. *html_linter.py*
    3.2. *general_purpose_linter.py* (If need to check for bad patterns in files)

4. **CSS**
    4.1. *css_linter.py*
    4.2. *general_purpose_linter.py* (If need to check for bad patterns in files)

For tree structure of linter files, see screenshot below.



```
├── linters
    ├── codeowner_linter.py
    ├── css_linter.py
    ├── general_purpose_linter.py
    ├── html_linter.py
    ├── __init__.py
    ├── js_ts_linter.py
    ├── linter_utils.py
    ├── linter_utils_test.py
    ├── pre_commit_linter.py
    ├── pre_commit_linter_test.py
    ├── pylint_extensions.py
    ├── pylint_extensions_test.py
    └── python_linter.py
```

Tests for the different linter files will be written in following files under **oppia/core/tests/linter_tests** directory:

1. **Python**
    1.1. *pylint_extensions_test.py*
    1.2. *valid.py*
    1.3. *invalid.py*

2. **Typescript**
    2.1. *valid.ts*
    2.2. *Invalid.ts*

3. **Javascript**
    3.1. *valid.js*
    3.2. *invalid.ts*

4. **HTML**
    4.1. *valid.html*

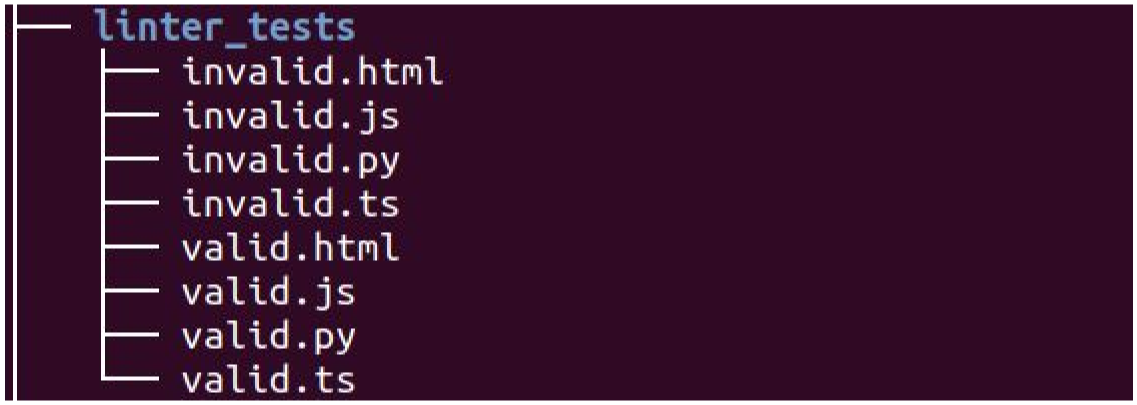
4.2. *invalid.html*

5. CSS

5.1. *valid.css*

5.2. *invalid.css*

For tree structure of directory, see screenshot below.



```
├── linter_tests
│   ├── invalid.html
│   ├── invalid.js
│   ├── invalid.py
│   ├── invalid.ts
│   ├── valid.html
│   ├── valid.js
│   ├── valid.py
│   └── valid.ts
```

We may need to add more test files depending on the requirement.

Implementation Approach

The project is divided into 2 major parts. Below are the detailed explanations of each of these.

1. Refactoring of linter

1.1. Addition of exception handling

Task execution and output catching will be covered under try/except block.

The code will look something like this:

try/except block at task execution ensure that it will catch any exception during execution of task

```
custom_task_execution_failed = False
# Concurrency limit: 25.
try:
    concurrent_task_utils.execute_tasks(tasks_custom, custom_semaphore)
except Exception:
    custom_task_execution_failed = True

third_party_task_execution_failed = False
# Concurrency limit: 2.
try:
    concurrent_task_utils.execute_tasks(
        tasks_third_party, third_party_semaphore)
except Exception:
    third_party_task_execution_failed = True
```

try/except block at catching output ensures that it will catch any error during the execution of a particular linter execution.

```
for task in tasks_custom:
    try:
        all_messages += task.output
    except Exception:
        concurrent_task_utils.log(
            python_utils.convert_to_bytes(task.exception.args[0]))

for task in tasks_third_party:
    try:
        all_messages += task.output
    except Exception:
        concurrent_task_utils.log(
            python_utils.convert_to_bytes(task.exception.args[0]))
```

This part is just an example, more code will be added when I will implement this fully.

1.2. Splitting of linter

This part is already completed in #8656.

1.3. Making output less verbose

This is the basic strategy of making output less verbose:

1. Error messages will be stored in *summary_message* and then appended to a list *summary_messages* and print the *summary_message.* Then this list will be returned to be used in main linting script

```
        summary_messages.append(summary_message)
        python_utils.PRINT(summary_message)
        python_utils.PRINT('')
    return summary_messages
```

2. Then, Individual linter output will be added to list *all_messages.*

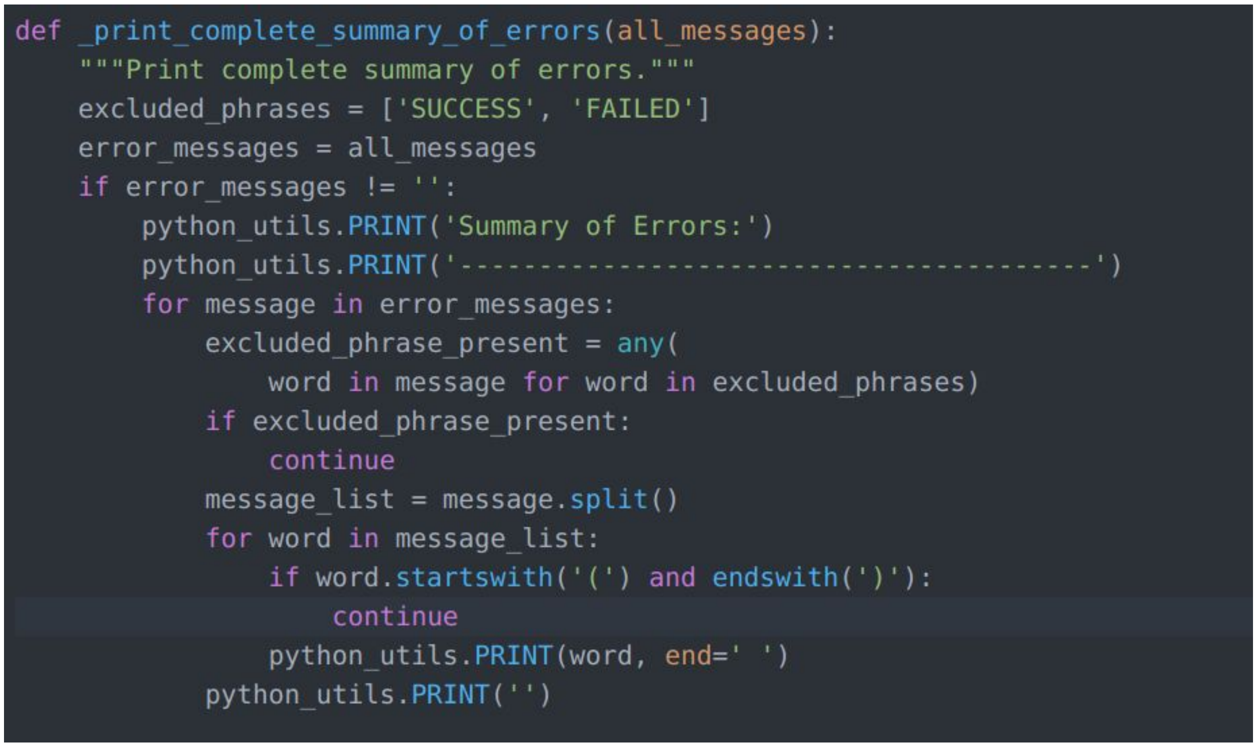
```
for task in tasks_third_party:
    all_messages += task.output
```

3. Then the list containing all the linter messages will be passed to a method to trim down the extra bits in linter output.

```
_print_complete_summary_of_errors(all_messages)
```

4. Script for removal of extra bits will be written in '*_print_complete_summary_of errors'* method*.*

5. I am going to ignore all the extra bits of the linter output by processing the linter output in a function and only an important part of the linter output will be shown. The function will look something like this:



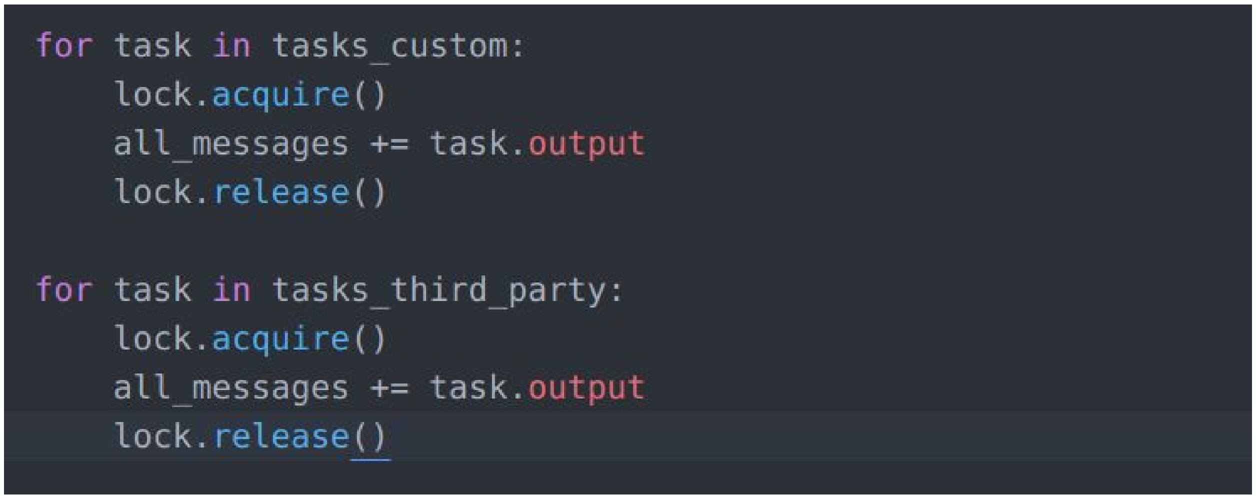
```python
def _print_complete_summary_of_errors(all_messages):
    """Print complete summary of errors."""
    excluded_phrases = ['SUCCESS', 'FAILED']
    error_messages = all_messages
    if error_messages != '':
        python_utils.PRINT('Summary of Errors:')
        python_utils.PRINT('----------------------------------------')
        for message in error_messages:
            excluded_phrase_present = any(
                word in message for word in excluded_phrases)
            if excluded_phrase_present:
                continue
            message_list = message.split()
            for word in message_list:
                if word.startswith('(') and endswith(')'):
                    continue
                python_utils.PRINT(word, end=' ')
            python_utils.PRINT('')
```

This part is just an example, more code will be added when I will implement this fully.

2. Locking mechanism for summary_messages

As of now, we do not have a locking mechanism for *summary_messages* because of that sometimes we have lint errors of one file under the name of different files. To solve this I am going to implement a locking mechanism so that *the summary_messages* variable can be accessed by one resource at a time.

The code will look something like this:



```python
for task in tasks_custom:
    lock.acquire()
    all_messages += task.output
    lock.release()

for task in tasks_third_party:
    lock.acquire()
    all_messages += task.output
    lock.release()
```

3. Implementation of new lint checks.

This is the basic strategy for writing lint checks.

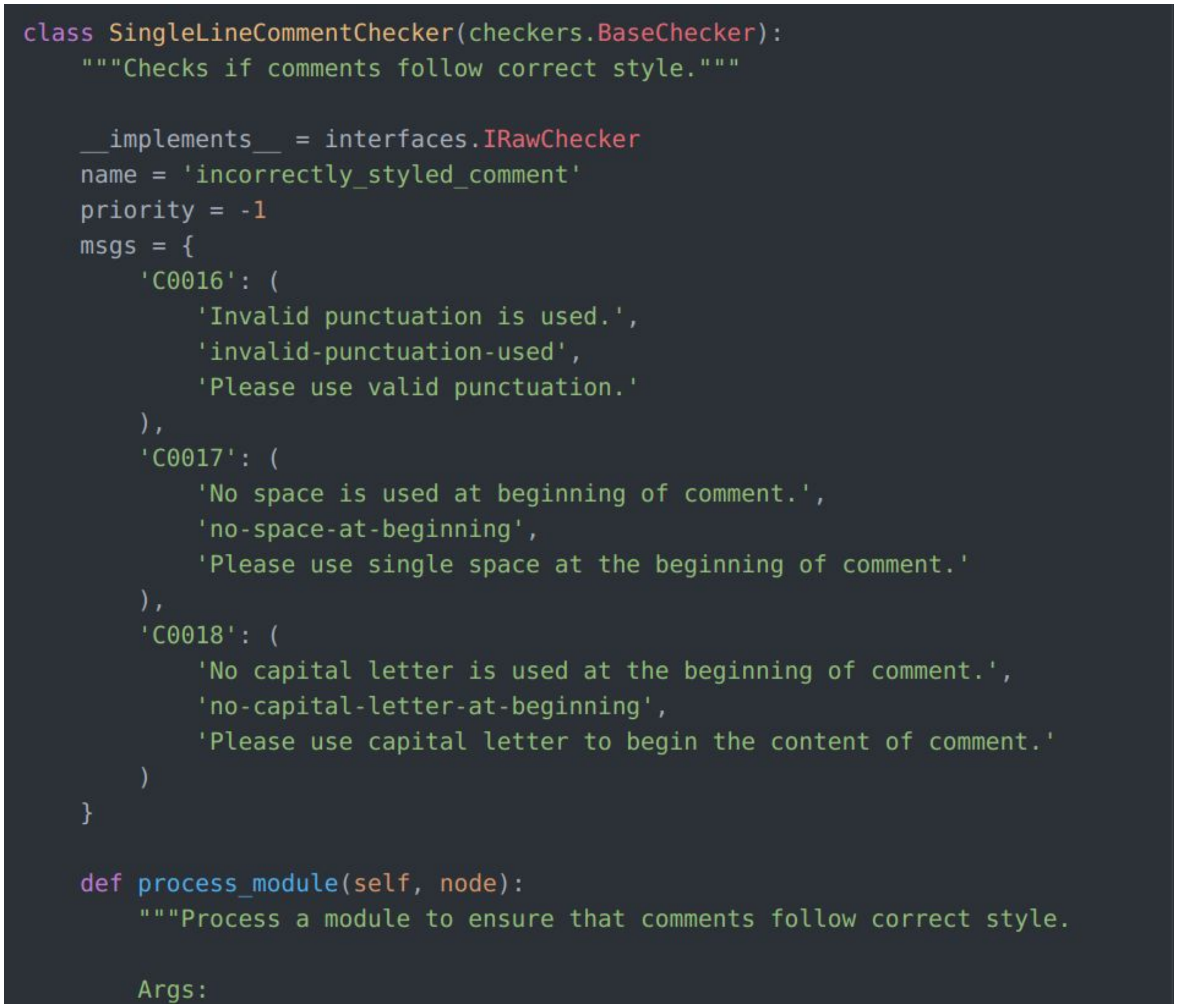
Custom Python Lint Checks

1. We need to create a **Class** with the desired name with parent class **checkers.BaseChecker**. Inside this class we have the following parts.

   1.1. **A name.** The name is used to generate a special configuration section for the checker, when options have been provided.

   1.2. **A priority.** This must be to be lower than 0. The checkers are ordered by the priority when run, from the most negative to the most positive.

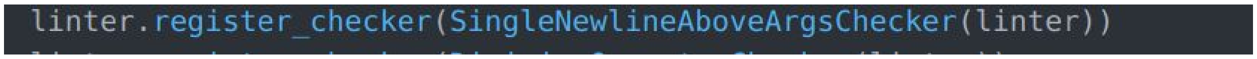
   1.3. **A message dictionary.** Each checker is being used for finding problems in code, the problems being displayed to the user through messages.



```python
class SingleLineCommentChecker(checkers.BaseChecker):
    """Checks if comments follow correct style."""

    __implements__ = interfaces.IRawChecker
    name = 'incorrectly_styled_comment'
    priority = -1
    msgs = {
        'C0016': (
            'Invalid punctuation is used.',
            'invalid-punctuation-used',
            'Please use valid punctuation.'
        ),
        'C0017': (
            'No space is used at beginning of comment.',
            'no-space-at-beginning',
            'Please use single space at the beginning of comment.'
        ),
        'C0018': (
            'No capital letter is used at the beginning of comment.',
            'no-capital-letter-at-beginning',
            'Please use capital letter to begin the content of comment.'
        )
    }

    def process_module(self, node):
        """Process a module to ensure that comments follow correct style.

        Args:
```

2. *Process_module* contains the logic for the lint test.

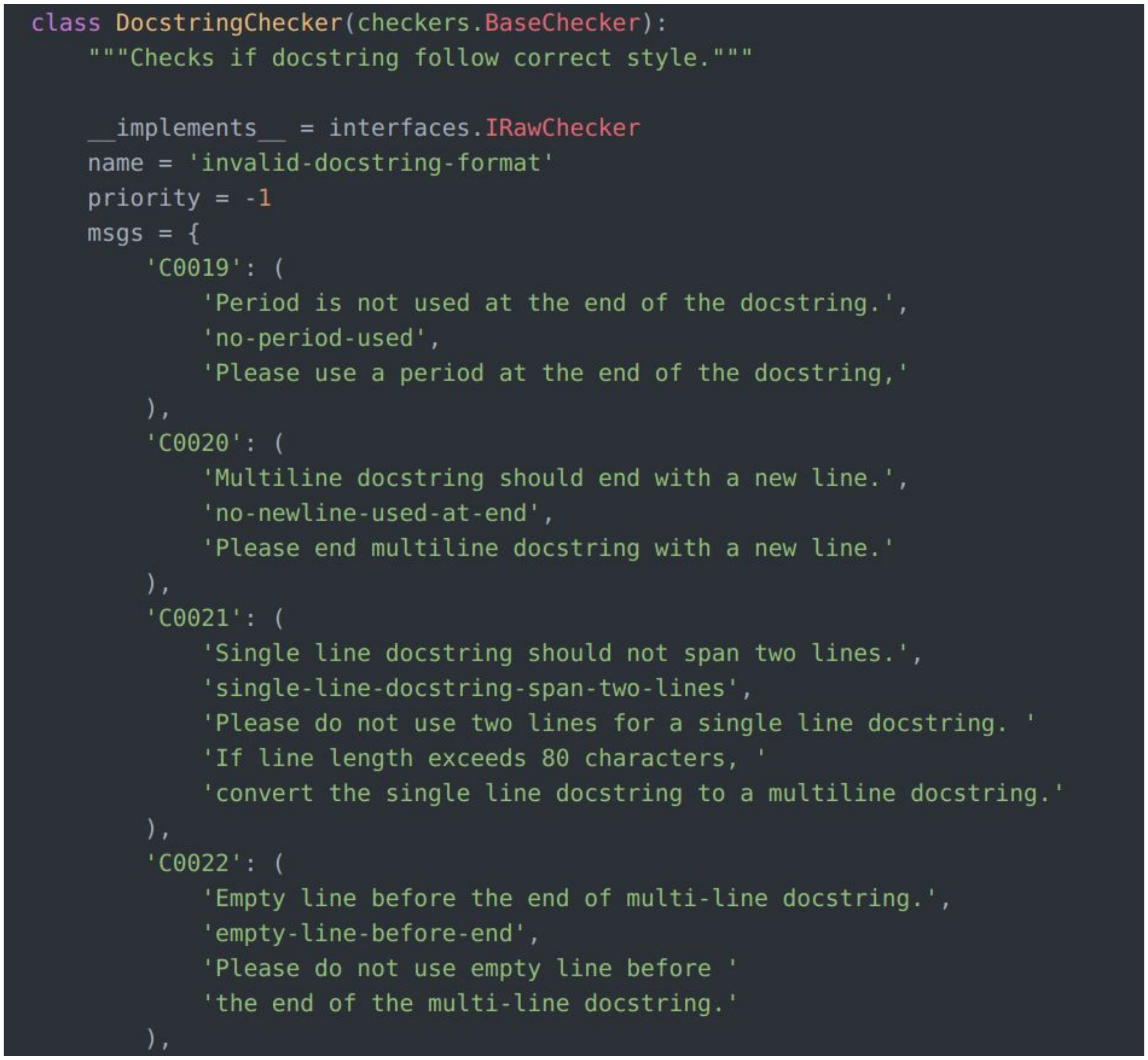
3. At the end we need to register this class with the pylint so that we can use the new custom pylint.



```python
 linter.register_checker(SingleNewlineAboveArgsChecker(linter))
```

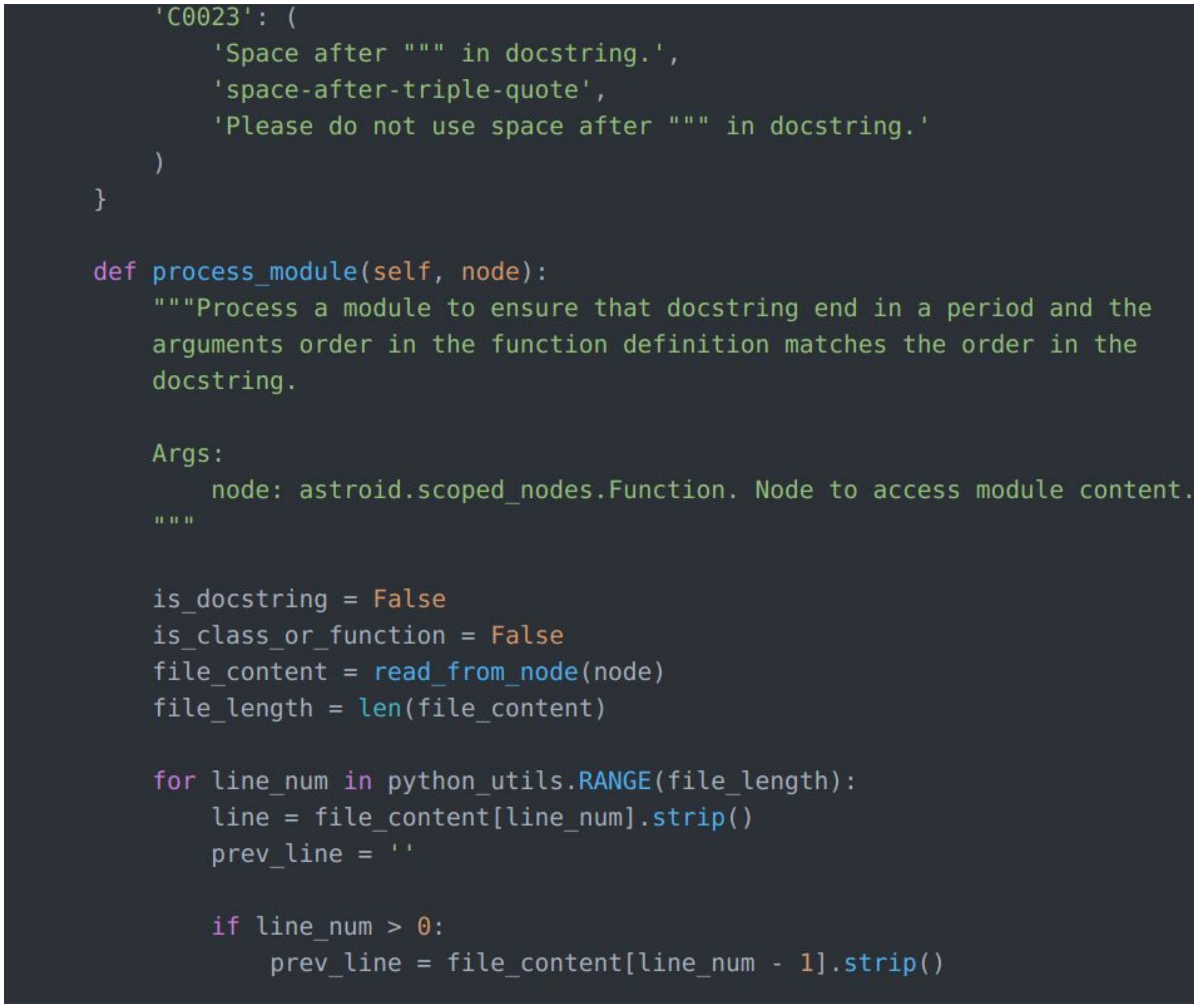
#7881, #7929 is the link to two PRs in which I had added lint checks.

Example of docstring checker
Below is the code of docstring checker merged in #8410

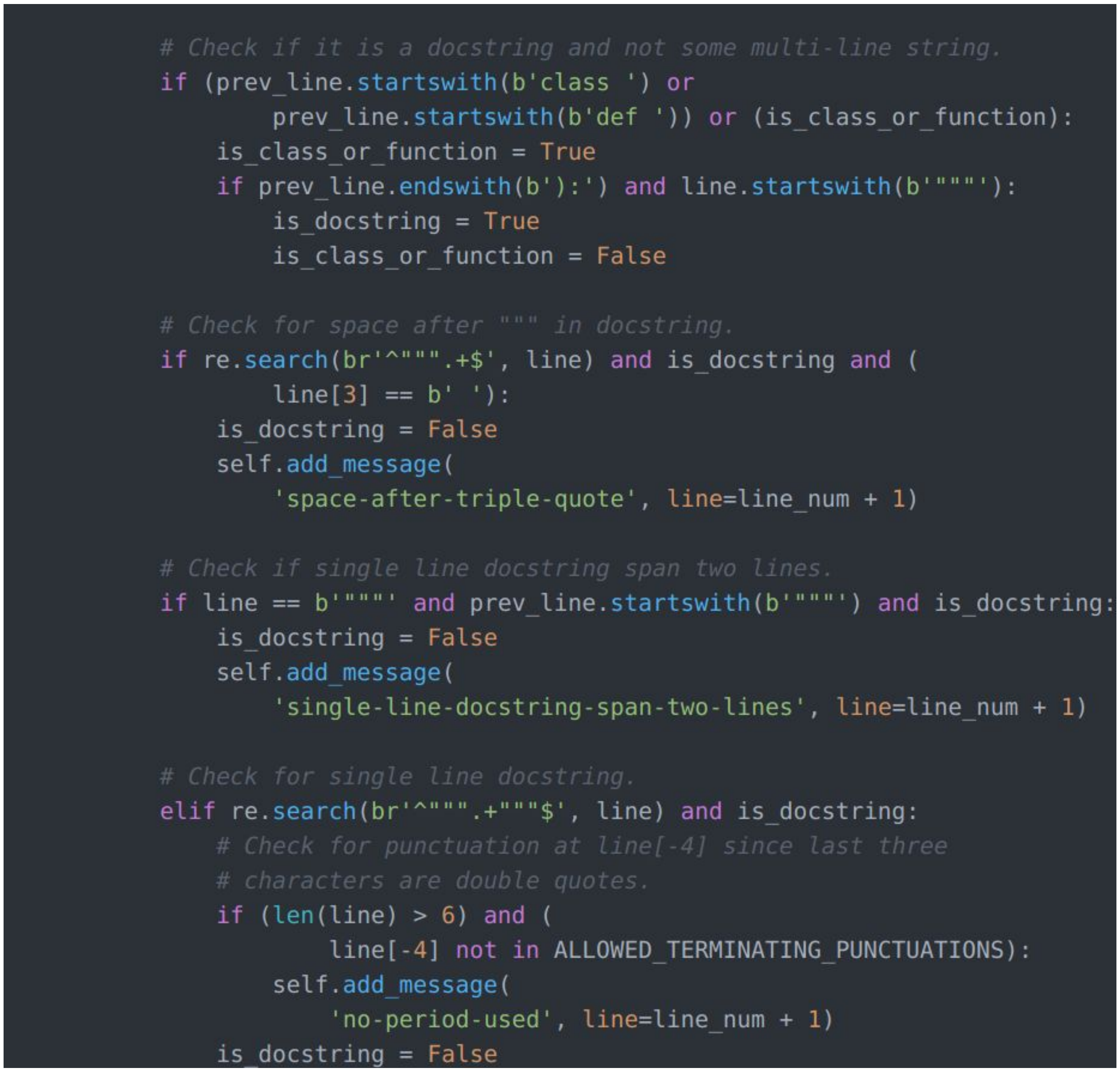


```
class DocstringChecker(checkers.BaseChecker):
    """Checks if docstring follow correct style."""

    __implements__ = interfaces.IRawChecker
    name = 'invalid-docstring-format'
    priority = -1
    msgs = {
        'C0019': (
            'Period is not used at the end of the docstring.',
            'no-period-used',
            'Please use a period at the end of the docstring,'
        ),
        'C0020': (
            'Multiline docstring should end with a new line.',
            'no-newline-used-at-end',
            'Please end multiline docstring with a new line.'
        ),
        'C0021': (
            'Single line docstring should not span two lines.',
            'single-line-docstring-span-two-lines',
            'Please do not use two lines for a single line docstring. '
            'If line length exceeds 80 characters, '
            'convert the single line docstring to a multiline docstring.'
        ),
        'C0022': (
            'Empty line before the end of multi-line docstring.',
            'empty-line-before-end',
            'Please do not use empty line before '
            'the end of the multi-line docstring.'
        ),
```

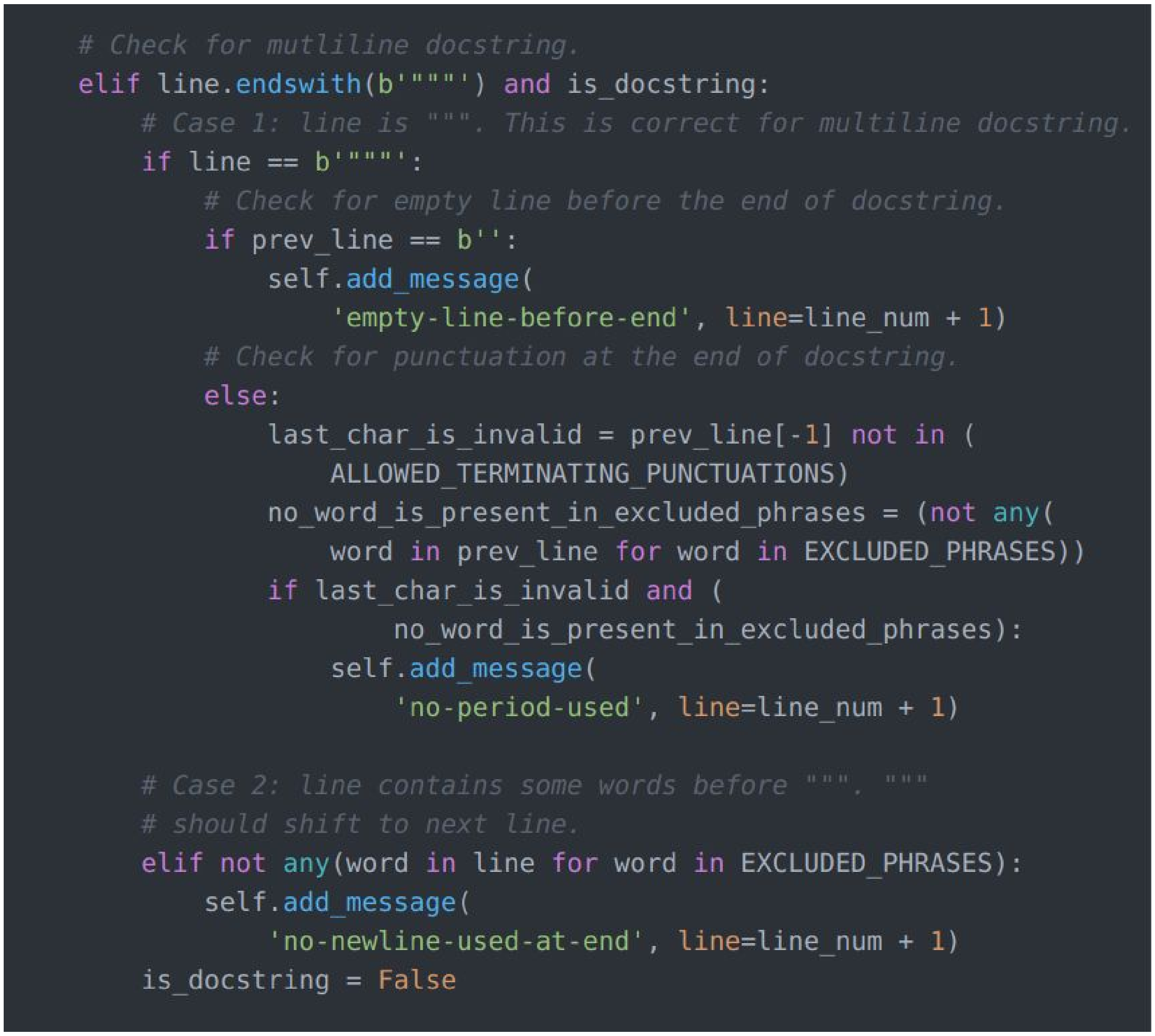
The Challenge in writing this function is to check if the given docsting is really a docstring or a multiline comment for writing test in pylint_extension_test.py. This is checked by creating a variable is_class_or_function initialized with ‘FALSE’ value. This is only TRUE if the docstring is found below the class or function definition.



```
        'C0023': (
            'Space after """ in docstring.',
            'space-after-triple-quote',
            'Please do not use space after """ in docstring.'
        )
    }

    def process_module(self, node):
        """Process a module to ensure that docstring end in a period and the
        arguments order in the function definition matches the order in the
        docstring.

        Args:
            node: astroid.scoped_nodes.Function. Node to access module content.
        """

        is_docstring = False
        is_class_or_function = False
        file_content = read_from_node(node)
        file_length = len(file_content)

        for line_num in python_utils.RANGE(file_length):
            line = file_content[line_num].strip()
            prev_line = ''

            if line_num > 0:
                prev_line = file_content[line_num - 1].strip()
```



```python
# Check if it is a docstring and not some multi-line string.
if (prev_line.startswith(b'class ') or
        prev_line.startswith(b'def ')) or (is_class_or_function):
    is_class_or_function = True
    if prev_line.endswith(b'):') and line.startswith(b'"""'):
        is_docstring = True
        is_class_or_function = False

# Check for space after """ in docstring.
if re.search(br'^""".+$', line) and is_docstring and (
        line[3] == b' '):
    is_docstring = False
    self.add_message(
        'space-after-triple-quote', line=line_num + 1)

# Check if single line docstring span two lines.
if line == b'"""' and prev_line.startswith(b'"""') and is_docstring:
    is_docstring = False
    self.add_message(
        'single-line-docstring-span-two-lines', line=line_num + 1)

# Check for single line docstring.
elif re.search(br'^""".+"""$', line) and is_docstring:
    # Check for punctuation at line[-4] since last three
    # characters are double quotes.
    if (len(line) > 6) and (
            line[-4] not in ALLOWED_TERMINATING_PUNCTUATIONS):
        self.add_message(
            'no-period-used', line=line_num + 1)
    is_docstring = False
```



```
# Check for mutliline docstring.
elif line.endswith(b'"""') and is_docstring:
    # Case 1: line is """. This is correct for multiline docstring.
    if line == b'"""':
        # Check for empty line before the end of docstring.
        if prev_line == b'':
            self.add_message(
                'empty-line-before-end', line=line_num + 1)
        # Check for punctuation at the end of docstring.
        else:
            last_char_is_invalid = prev_line[-1] not in (
                ALLOWED_TERMINATING_PUNCTUATIONS)
            no_word_is_present_in_excluded_phrases = (not any(
                word in prev_line for word in EXCLUDED_PHRASES))
            if last_char_is_invalid and (
                    no_word_is_present_in_excluded_phrases):
                self.add_message(
                    'no-period-used', line=line_num + 1)

    # Case 2: line contains some words before """. """
    # should shift to next line.
    elif not any(word in line for word in EXCLUDED_PHRASES):
        self.add_message(
            'no-newline-used-at-end', line=line_num + 1)
    is_docstring = False
```

Lint Checks for bad pattern matching.

This is the general approach of writing lint checks for bad pattern matching.:

1. If the lint check is meant to just check bad patterns. Then the *regex* for that pattern will go in the bad pattern matching lists in the **general_purpose_linter.py** file. The list has dictionaries in it to store regexes and corresponding messages. List of bad pattern matching lists is as follows:

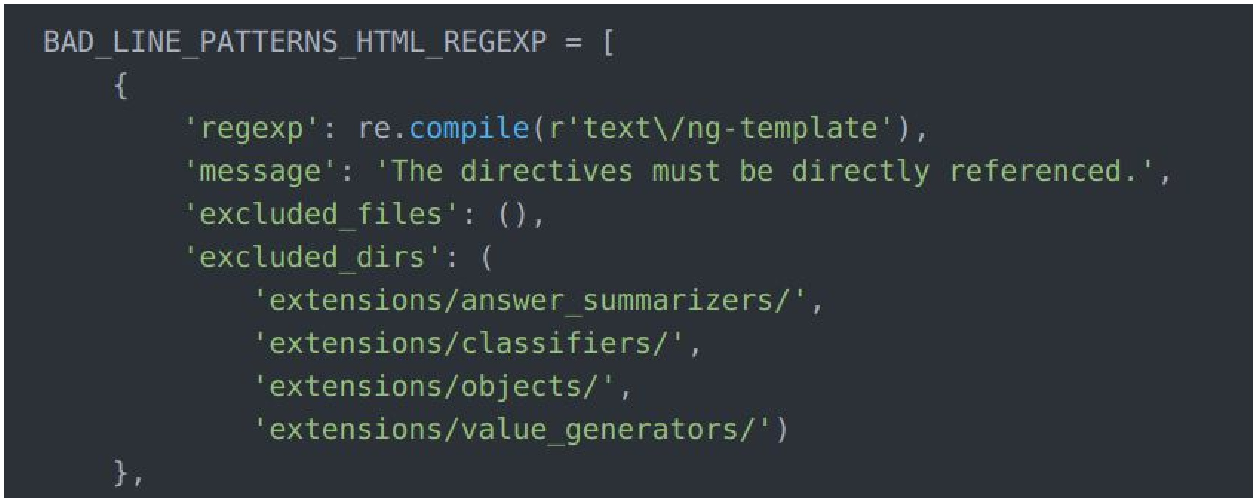
    1.1. **BAD_PATTERNS ->** For general bad pattern matching for every file type.

1.2. **BAD_PATTERNS_JS_AND_TS_REGEXP ->** For bad pattern matching in js and ts files.
1.3. BAD_LINE_PATTERNS_HTML_REGEXP -> For bad pattern matching in HTML files.
1.4. BAD_PATTERNS_PYTHON_REGEXP -> For bad pattern matching in Python files.

Below is the general format to define a bad pattern check:

1. **A regexp:** regexp is used to define the regex pattern which is used to catch the unwanted patterns.
2. **A message:** Message is used to display the problem to the user.
3. **excluded_files:** Files to be excluded from the pattern matching.
4. **excluded_dirs:** Directories to be excluded from the pattern matching

   Below is an example for writing a bad pattern matching lint check.



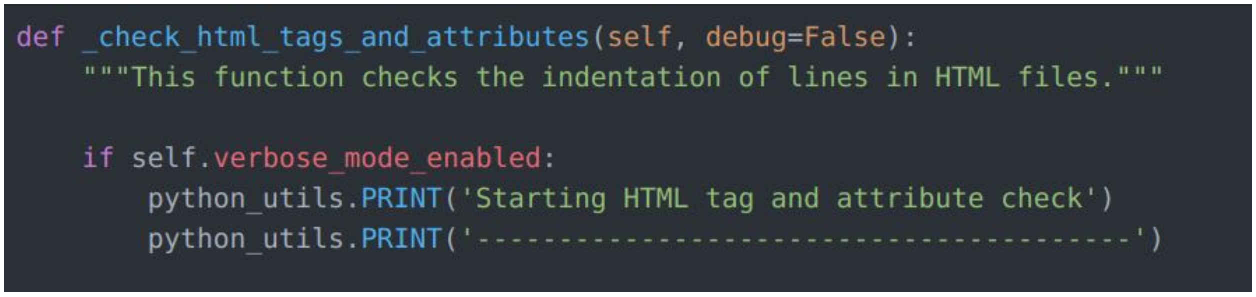
```
BAD_LINE_PATTERNS_HTML_REGEXP = [
    {
        'regexp': re.compile(r'text\/ng-template'),
        'message': 'The directives must be directly referenced.',
        'excluded_files': (),
        'excluded_dirs': (
            'extensions/answer_summarizers/',
            'extensions/classifiers/',
            'extensions/objects/',
            'extensions/value_generators/')
    },
```

Writing Custom HTML, CSS, JsTs Lint Checks

Custom Lint checks for these file types will be written in their respective files i.e. **html_linter.py, css_linter.py, python_linter.py, js_ts_linter.py**.

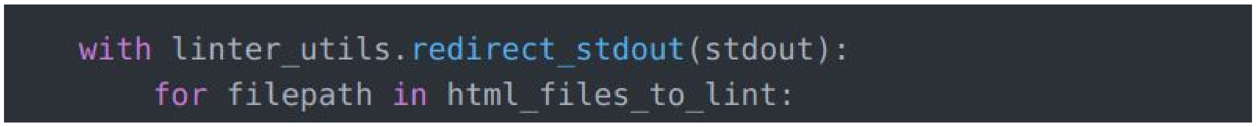
The general strategy for writing custom HTML, CSS, JsTs Lint checks is as follows:

1. Define a method under a class(HTMLLintChecks, PythonLintChecks, JsTsLintChecks).



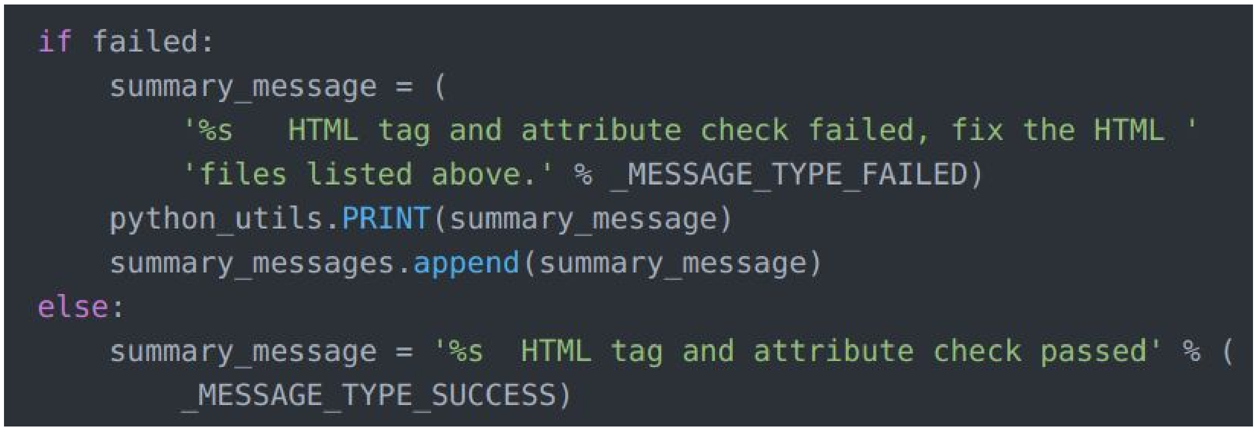
```python
def _check_html_tags_and_attributes(self, debug=False):
    """This function checks the indentation of lines in HTML files."""

    if self.verbose_mode_enabled:
        python_utils.PRINT('Starting HTML tag and attribute check')
        python_utils.PRINT('----------------------------------------')
```

2. Redirect output to terminal.



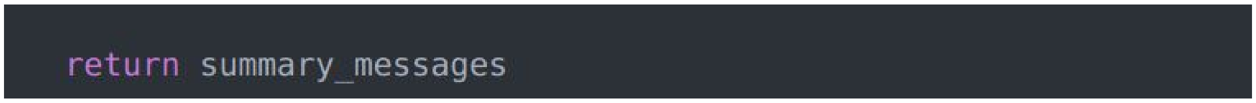
```python
    with linter_utils.redirect_stdout(stdout):
        for filepath in html_files_to_lint:
```

3. Catch output and append in a list.



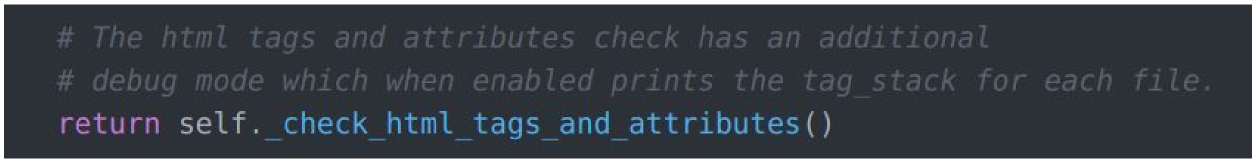
```python
if failed:
    summary_message = (
        '%s   HTML tag and attribute check failed, fix the HTML '
        'files listed above.' % _MESSAGE_TYPE_FAILED)
    python_utils.PRINT(summary_message)
    summary_messages.append(summary_message)
else:
    summary_message = '%s  HTML tag and attribute check passed' % (
        _MESSAGE_TYPE_SUCCESS)
```

4. Return the ‘*summary_messages’* list.



```python
    return summary_messages
```

5. Call the method defined above from *‘perform_all_lint_checks’* method and return it.



```python
# The html tags and attributes check has an additional
# debug mode which when enabled prints the tag_stack for each file.
return self._check_html_tags_and_attributes()
```

Testing Approach

Custom Pylint checks

This is the general strategy for writing custom pylint checks:

1. We need to create a class with the parent **unittest.TestCase** class.

```
class SingleLineCommentCheckerTests(unittest.TestCase):
```

2. Then we need to create the object of the checker class we want to test.

```
def setUp(self):
    super(SingleLineCommentCheckerTests, self).setUp()
    self.checker_test_object = testutils.CheckerTestCase()
    self.checker_test_object.CHECKER_CLASS = (
        pylint_extensions.SingleLineCommentChecker)
    self.checker_test_object.setup_method()
```

3. Then there will be different test cases to test the above created test class object.

```
def test_invalid_punctuation(self):
    node_invalid_punctuation = astroid.scoped_nodes.Module(
        name='test',
        doc='Custom test')
    temp_file = tempfile.NamedTemporaryFile()
    filename = temp_file.name

    with python_utils.open_file(filename, 'w') as tmp:
        tmp.write(
            u"""# Something/
            """)
```

4. Check if it outputs the right message.

```
message = testutils.Message(
    msg_id='invalid-punctuation-used',
    line=1)

with self.checker_test_object.assertAddsMessages(message):
    temp_file.close()
```

Custom Python, CSS, HTML, JsTs lint checks

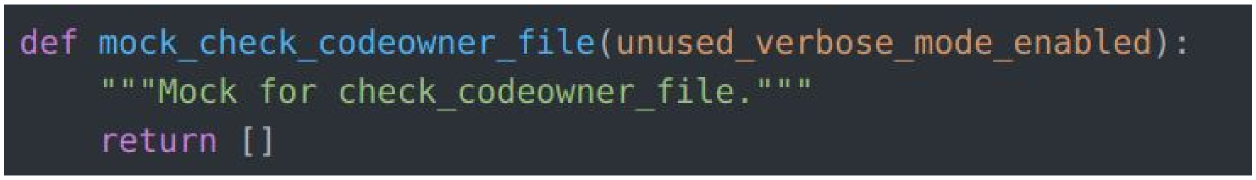
Currently, we do not have a mechanism to test the custom Python, CSS, HTML, JsTs lint checks. I will create the test mechanism in a new file named **pre_commit_linter_test.py** under **oppia/scripts/linters/** directory.

This is the general strategy to create test mechanism:

1. We need to create a general class to handle all linter function tests. This has **test_utils.GenericTestBase** as a parent class to have basic testing methods.

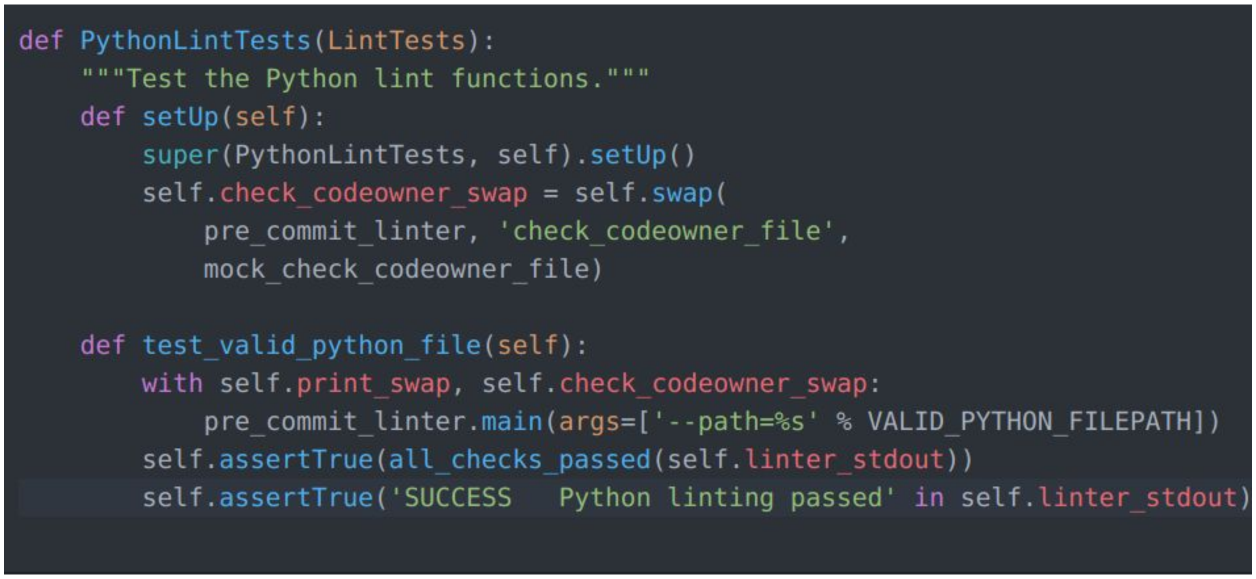
```
class LintTests(test_utils.GenericTestBase):
    """General class for all linter function tests."""
    def setUp(self):
        super(LintTests, self).setUp()
        self.linter_stdout = []
        def mock_print(val):
            """Mock for python_utils.PRINT. Append the values to print to
            linter_stdout list.

            Args:
                val: str. The value to print.
            """
            if isinstance(val, list):
                self.linter_stdout.extend(val)
            else:
                self.linter_stdout.append(val)
        self.print_swap = self.swap(python_utils, 'PRINT', mock_print)
        self.sys_swap = self.swap(sys, 'exit', mock_exit)
```

2. Also, we are going to need **mock_check_codeowner_file** method to suppress code owner file messages.



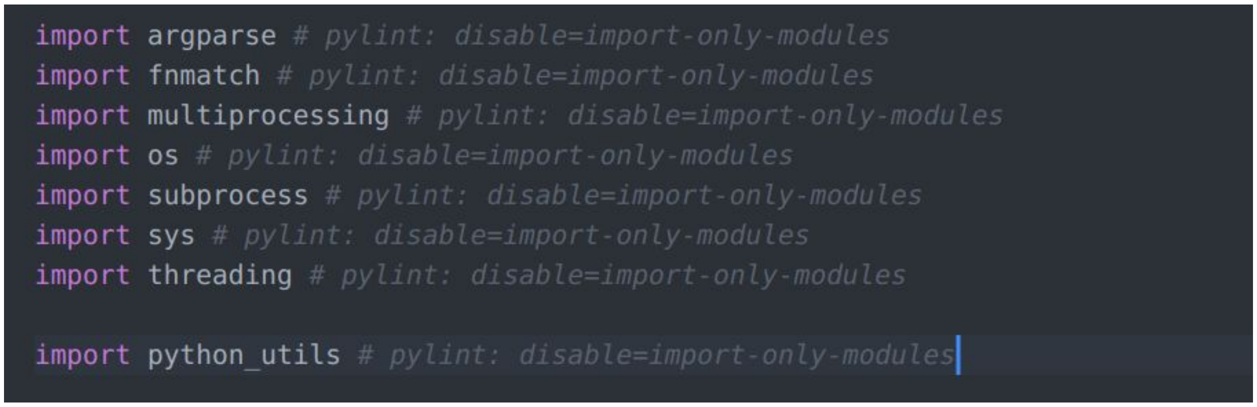
```python
def mock_check_codeowner_file(unused_verbose_mode_enabled):
    """Mock for check_codeowner_file."""
    return []
```

3. Then, we create the Test class for that filetype which we want to lint. In this case as an example I used **PythonLintTests**. In the setup method for this class.This class has a **test_valid_python_file** method to check the contents of **valid.py** test file contents. In this method we check the output of the **pre_commit_linter.py** if it is outputting what we want.



```python
def PythonLintTests(LintTests):
    """Test the Python lint functions."""
    def setUp(self):
        super(PythonLintTests, self).setUp()
        self.check_codeowner_swap = self.swap(
            pre_commit_linter, 'check_codeowner_file',
            mock_check_codeowner_file)

    def test_valid_python_file(self):
        with self.print_swap, self.check_codeowner_swap:
            pre_commit_linter.main(args=['--path=%s' % VALID_PYTHON_FILEPATH])
        self.assertTrue(all_checks_passed(self.linter_stdout))
        self.assertTrue('SUCCESS   Python linting passed' in self.linter_stdout)
```

4. **oppia/core/tests/linter_tests** contains all the tests file for **pre_commit_linter.** In this directory, files starting with **valid** contain the valid format for testing the Linting classes. While files starting with **invalid** contain the invalid format for testing desired Linting classes. Here is an example of a valid format file for the Python linting function.



```python
import argparse # pylint: disable=import-only-modules
import fnmatch # pylint: disable=import-only-modules
import multiprocessing # pylint: disable=import-only-modules
import os # pylint: disable=import-only-modules
import subprocess # pylint: disable=import-only-modules
import sys # pylint: disable=import-only-modules
import threading # pylint: disable=import-only-modules

import python_utils # pylint: disable=import-only-modules
```

Milestones

Tasks before GSoC starts (Current Period - 31 May)

- Adding locking mechanism for '*summary_messages*'
- Contributing to bug fixes and other issues
- Reviewing PRs

Milestone 1

Key Objective: In non-verbose mode, the linter has less verbose output, and raises an exception if there is any error due to the operation of the linter script. All existing lint checks have tests written for them. The linter handles all general and CSS lint errors, and the linter script has 100% test coverage.

No.	**Description of PR**	**Prereq PR numbers**	**Target date for PR submission**	**Target date for PR to be merged**
1.1	Add tests for existing checks in pre_commit_linter.py		1/6/2020	3/6/2020
1.2	Lint output is more cleaner and less verbose.		2/6/2020	4/6/2020
1.3	Exception handling is used to handle unexpected errors.		3/6/2020	5/6/2020
1.4	Add lint check to ensure that all lines in skip_files in app.yaml reference valid files in the repository		5/6/2020	7/6/2020
1.5	Add newline check at end of file		7/6/2020	9/6/2020
1.6	Add lint checks to ensure that there are valid spaces and newlines		9/6/2020	11/6/2020
1.7	Add lint checks for webpack config		11/6/2020	13/6/2020
1.8	Add lint check that all TODO comments start with capital letter		13/6/2020	15/6/2020
1.9	Add lint check for proper comment style for Python and JS files		15/6/2020	18/6/2020
1.10	Add lint check to ensure that every file (of any type) ends with exactly one newline		18/6/2020	20/6/2020

	character.			
1.11	Add lint check for alphabetized list in CSS		20/6/2020	23/6/2020
1.12	Add lint check to prohibit inline styling		23/6/2020	25/6/2020

Milestone 2

Key Objective: The linter fully handles all Python lint checks. All JS/TS lint checks are fully implemented, except for at most 11.

No.	**Description of PR**	**Prereq PR numbers**	**Target date for PR submission**	**Target date for PR to be merged**
2.1	Add lint check to enforce multiples of 4 indentation in docstring		3/7/2020	7/7/2020
2.2	Add lint check for proper comment indentation		4/7/2020	8/7/2020
2.3	Add lint check for proper Args and Returns style		6/7/2020	10/7/2020
2.4	Add lint check to multiline expressions ensuring line break after		8/7/2020	12/7/2020
2.5	Add lint check for args-name for a non-keyword argument		10/7/2020	14/7/2020
2.6	Add lint check to ensure that pylint pragmas are used to disable any rule for a single line		12/7/2020	16/7/2020
2.7	Add lint check to ensure that there is one blank newline below each class docstring.		14/7/2020	18/7/2020
2.8	Add lint check to detect variables that are declared but never used.		16/7/2020	20/7/2020
2.9	Add lint check to forbid usage of assertRaises; require assertRaisesRegexp instead.		18/7/2020	22/7/2020
2.10	Add a lint check to enforce that PEP8 naming convention is followed.		20/7/2020	24/7/2020

2.11	Add lint check to forbid use of innerHTML due to security issues		22/7/2020	24/7/2020
2.12	Add a lint check to check for unused imports		23/7/2020	25/7/2020

Milestone 3

Key Objective: The linter fully handles all JS/TS lint checks.

No.	Description of PR	Prereq PR numbers	Target date for PR submission	Target date for PR to be merged
3.1	Add a check to ensure that multiline expressions in parentheses are broken down after "(" in JavaScript files		31/7/2020	2/7/2020
3.2	Add lint check for unused directive/service dependencies		2/8/2020	4/8/2020
3.3	Add lint check to Improve reachability of sorted imports		4/8/2020	6/8/2020
3.4	Add lint check that filename is similar to service or directive name		6/8/2020	8/8/2020
3.5	Add lint check for eslint-disable statements		8/8/2020	10/8/2020
3.6	Add lint check to ensure require(…) statements are alphabetized		10/8/2020	12/8/2020
3.7	Add lint check for test files		12/8/2020	16/8/2020
3.8	Add lint check to detect variables that are defined but never used.		14/8/2020	16/8/2020
3.9	Add lint check to make sure there's a space before the opening braces of a function start.		16/8/2020	18/8/2020
3.10	Add lint check to make sure that there should be no space before "function"		18/8/2020	20/8/2020
3.11	Add lint check to catch missing space after semicolon		20/8/2020	22/8/2020

Optional Sections

Future Work

After the GSoC, I will add the codeclimate linting to replace the following:

1. Eslint
2. Pylint
3. Stylelint
4. Bad Pattern matching

A significant amount of work is already done here(#8267)

Why we can’t use Code Climate now

The problem is with the pylint, as code climate uses ‘pylint v2’ and we are using ‘pylint v1’. Also, we can’t upgrade to ‘pylint v2’ as this will need Python 3 and we are currently using Python 2. There is also another issue, we can’t use pylint custom checks with code climate’s pylint engine due to code climate security policy. So, we are going to need a separate .pylintrc file for code climate. Due to these reasons, I was thinking of adding code climate linting after the GSoC period ends.