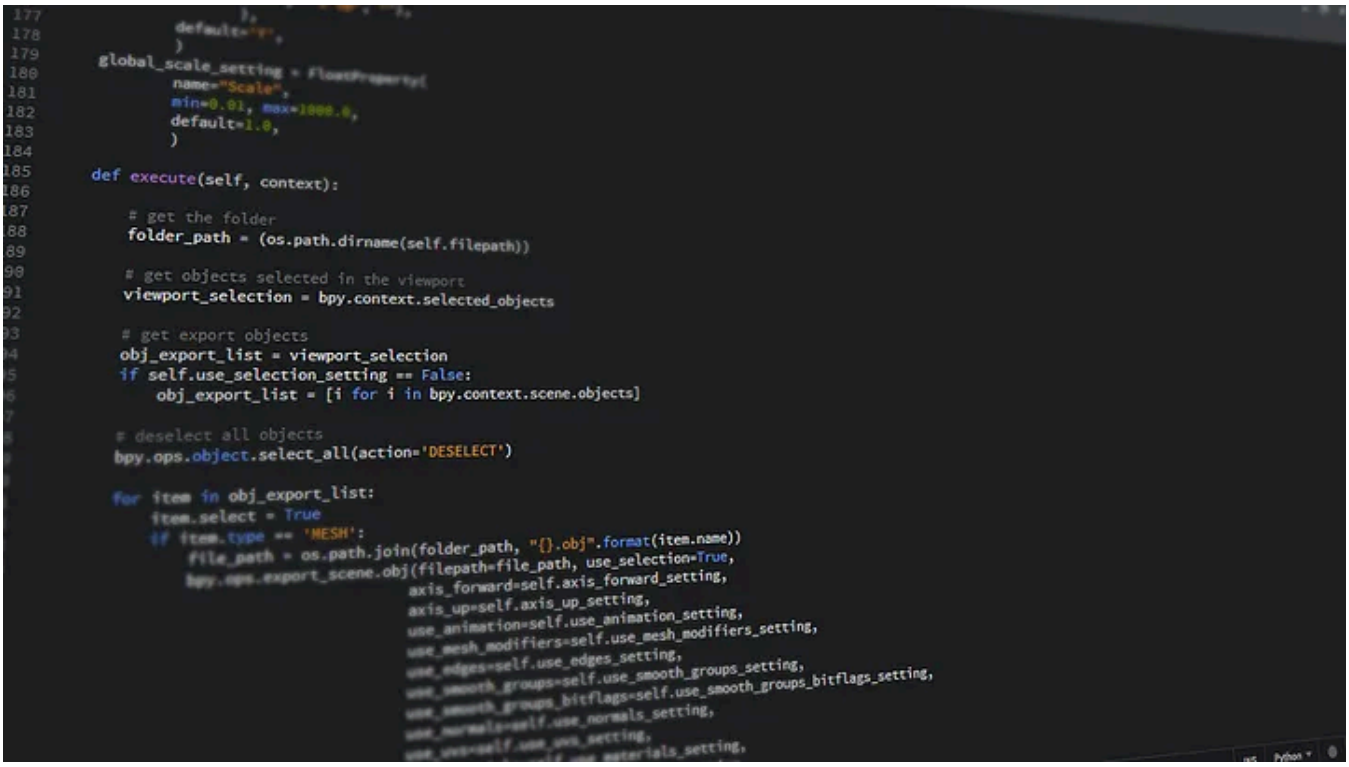
How To Get Started With Pylint

16 min read · Feb 23, 2023

Oliyadk Follow



⭐ Star your python code.

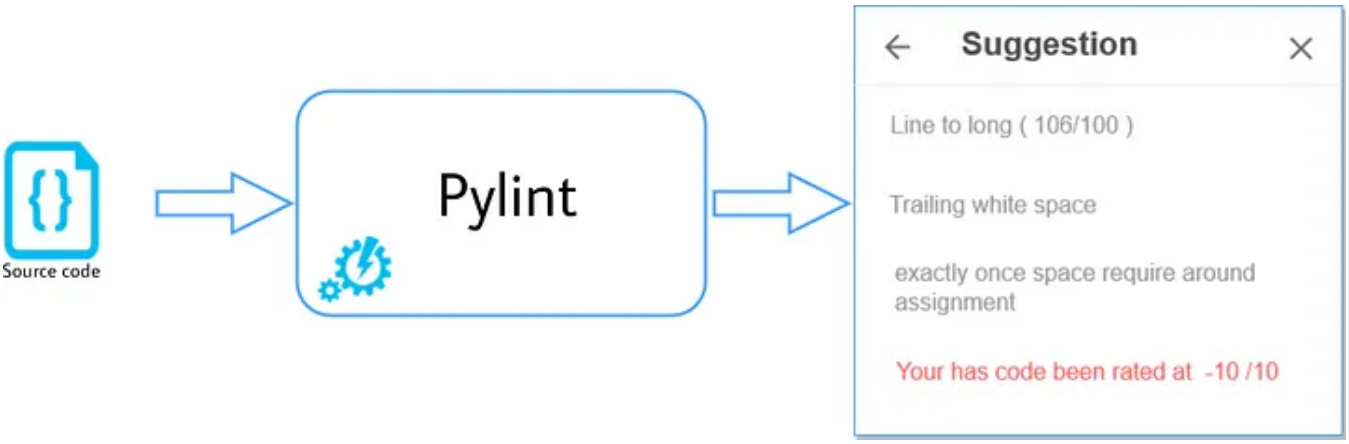
Source code analysis tools like Pylint help you to easily improve quality and comply with coding standards — without sacrificing speed. In this tutorial, I'll teach you about Pylint and I will walk you through the process of learning how to write additional checkers for pylint! Along the way, you'll be introduced to some of how static code analysis works under the hood.

Now Your journey has just begun... 🚀

👋 Welcome to Pylint

Pylint is an open-source python project that widely used a static code checker, that inspects your codebase, looks for potential errors or inconsistencies, enforces a coding standard, and looks for code smells in source code without executing the program. hence the term *"static"* code analysis.

How does this happen?



The process from code to pylint suggestion

✍️ The diagram represents, the process from code to pylint suggestion.

1. ➡ Initially, you input source code into the Pylint program.
2. ⬇ Pylint handles most of the complexity like code conversion to AST, parsing of the tree, and calling the correct checker by identifying specific coding patterns that have the same sort of warning or information associated with them,
3. ➡ finally, makes suggestions or infers how the code could be refactored.

Pylint can be seen as another PyChecker, Pyflakes, flake8, and mypy since nearly all tests you can do with them can also be done with pylint.

🤔 *So what makes Pylint so special?*

Okay, let see the most exciting parts!

1. **Coding Standard**

Pylint offers some more features, like checking the length of lines of code, checking if variable names are well-formed according to your coding standard, checking if declared interfaces are truly implemented, and much more.

2. **Customizable**

As we know that, Existing is not always enough. The big advantage of Pylint is that it is highly configurable and customizable. the config file and once you are familiar enough, you can create powerful configurations that enforce standards throughout your codebase.

3. **Extendable**

If customization not fulfilling for what you are looking for pylint allows you easily write a plugin to add a personal feature. This help developer follows a specific rule set by their company or team.

4. **UML diagrams**

Pylint is shipped with “pylint-gui”, and “pyreverse” (UML diagram generator).

👉 *Let’s take a look at How this tutorial is organized.*

We’ve broken up this tutorial into two major sections:

1. Run pylint on python code
2. Write a custom Pylint plugin

After finishing this tutorial You will be able to :

- ✔ Understand what pylint is, How to run it on python code, and the fundamental concepts of its systems.
- ✔ Learn how static analysis work.
- ✔ Know, how to transform your code into an Abstract Syntax Tree(AST) and how to process it.
- ✔ Build a simple custom plug-in for pylint.

😊Prerequisites

This tutorial assumes that you have some Python programming experience. If you don’t know Python yet, http://learnpython.org/ is a great place to start.

📄 Official documentation

Pylint's official documentation walks you through most of the important concepts required to use Pylint and also write checkers. you can download it here.

Code Examples

All of the code examples we use in this tutorial are available at https://github.com/OliyadKebede/pylint-tutorial .

Let's start with installation.

Installation

Installing Pylint is quite easy. One of the easiest ways to download Pylint is by using "pip".

NB: Pip is a package management system used to install and manage software packages written in Python. Your package manager will find a version that works with your interpreter. I recommend pip.

For command line use, pylint is installed with:

```
pip install pylint
```

🧐 Bear in your mind, Pylint can be used as a stand-alone program, but can also be integrated with most editors or IDES. more information can be found in the documentation.

To verify the pylint installation, type the following command

```
Pylint --version
```

output *:*

NB: you may get a different result.

In another way, you can simply run Pylint with no arguments and verify the pylint installation as well as it will also invoke the help dialogue and give you an idea of the arguments available to you.

Now, Let's start the first section of this tutorial

1️⃣ Run pylint on python code

Pylint run to look over the code. It will check against defined coding rules from hundreds of pre-configured checkers. For a full list of built-in checkers, see the Pylint documentation.

👌 Once the code is run through the static code analyzer, the analyzer will have identified whether or not the code complies with the set rules.

For this section, there are a few steps you need to go through —

1. Write python source code
2. Parse code and transform it into an Abstract Syntax Tree(AST) (parse tree)
3. Walk through the Abstract Syntax Tree (run a static code analyzer)
4. Run and review the results
5. Fix what needs to be fixed

😊 Intentionally, I added two steps in between, to help you catch the idea of, how static analysis tools work, which is very important for you to get an overall picture for designing a custom checker for pylint.

So, let's get to work

Step 1. write your python source code

Your first step is to write the code. The starting code we will use is called *simpl_circle_class.py* and is here in its entirety:

```
#!/user/bin/env python3

import string

PI = 3.14
```

```
class Circle:

    def __init__(self,radius:int) -> None:
        assert radius > 0 , \
            "circle radius must be a positive number"
        self.radius = radius

    def area(self) -> str:
        return PI * self.radius**2

    def perimeter(self) -> str:
        return 2 * PI * self.radius

    def __repr__(self):
        return f"{self.__class__.__name__}(radius={self.radius})"
```

Let's get started. Run the command pylint with the file name as shown below:

```
pylint simple_circle_class.py
```

📌Assume this step as input of the source code to pylint, and in the next steps, we will unlock the complex process behind pylint without further digging. After that, we look at the final output and refactor the code according to the suggestion.

Step 2. Parsing code and transforming it into an Abstract Syntax Tree

Pylint comes with a parser called asteroid which it uses to convert the code to ASTs (stands for Abstract Syntax Tree).

📌 **astroid** — is similar to the built-in **ast** module in the python standard library with extra functionality to make AST navigation and code introspection easier.

An Abstract Syntax Tree (AST) is simply a tree-based representation of source code. The transformation is no easy task as it sounds, thanks python for making this heavy process much easier.

Ironically, a similar concept is applied to make a compiler, a code generator, or, for example, a source code syntax checker.

✍️ There are three kinds of checkers in pylint. There are

1. AST checkers — operate on an abstract syntax tree (AST) of the code that Pylint generates.
2. filestreams checker — operate on the raw code without any pre-processing by Pylint and
3. tokens checkers — operate on a list of the tokens that represent the code being linted.

But most of the pylint checkers are AST based, meaning they operate on the abstract syntax tree of the source code.

For example, the following picture represents the AST of the source code we use in an established parser library called astroid to generate one.

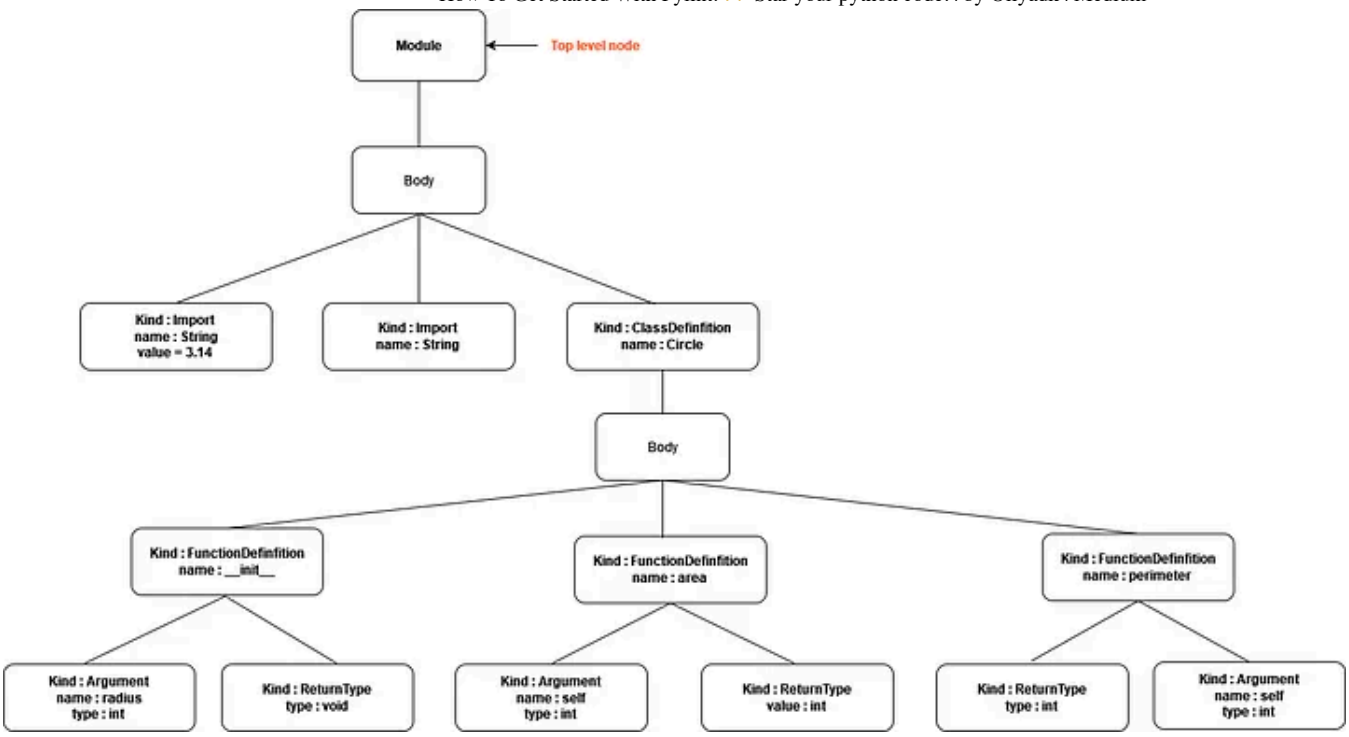
python code

To understand the code that does this, we need to use the parser functions from astroid and pass the ***simple_circle_class.py*** source string as an argument. Run this:

```
import astroid
source_code =astroid.parse("""source string """) #Parses a source string of sim
source_code.repr_tree() #Get a string representation of the AST from this node
```

The output of this program is a mess 😨. You can found ***here***

Corresponding AST



This diagram Corresponding the Abstract Syntax Tree of the ***simple_circle_class.py*** source code. as you can see everything is a node. A node has only one parent (module) and zero or multiple leaves.

👌 This recursive structure of trees gives rise to an elegant way of processing trees. Now, we have a complete AST, on the next step, we see how we can analyze and process it.

Step 3. Pylint Walking through the Abstract Syntax Tree

Many pylint's checkers work by visiting nodes in a given AST. Visiting typically starts at the root node(module) and traverses the tree in a depth-first manner.

Pylint lets you define two special callback methods for collecting information about the context of specific statements or patterns that we wish to detect and process it using their node functions. these are

1. *visit_<node_type>* and
2. *leave_<node_type>*

where *node_type* is the lowercase name of the node class (as defined by astroid). The complete list of all node types can be found here.

All of the magic happens inside such methods. They are executed automatically when the parser iterates over nodes of the respective type. As expected, the visit method is called when the pylints parser visits a node and the leave method is called

when it leaves the node. we'll see this in action when defining a custom checker for pylint.

Step 4. Review the results

We can now get an idea of what a pylint handles behind the scene. The output should be something like this :

```
************* Module simple_circle_class
simple_circle_class.py:3:9: C0303: Trailing whitespace (trailing-whitespace)
simple_circle_class.py:13:34: C0303: Trailing whitespace (trailing-whitespace)
simple_circle_class.py:17:0: C0303: Trailing whitespace (trailing-whitespace)
simple_circle_class.py:19:0: C0304: Final newline missing (missing-final-newline)
simple_circle_class.py:1:0: C0114: Missing module docstring (missing-module-docstring)
simple_circle_class.py:5:0: C0115: Missing class docstring (missing-class-docstring)
simple_circle_class.py:12:4: C0116: Missing function or method docstring (missing-function-docstring)
simple_circle_class.py:15:4: C0116: Missing function or method docstring (missing-function-docstring)
simple_circle_class.py:1:0: W0611: Unused import string (unused-import)

-----------------------------------
Your code has been rated at 2.50/10
```

Now, considering the suggestions given in the report from Pylint. Each one focuses on a particular aspect of the source code, such as the number of messages by categories, and document string.

👉 Each message suggestion/point in the report will be given in a message format that consists of a 5-digit number, prefixed with a message category.

There are multiple message categories, these being

```
+----------+------------+-------------------------------------------------------
| category |    name    |                                 description
+----------+------------+-------------------------------------------------------
| C        | Convention | It is displayed when the program is not following the
| R        | Refactor   | It is displayed for bad code smell
| W        | Warning    | It is displayed for python specific problems
| E        | Error      | It is displayed when that particular line execution r
| F        | Fatal      | It is displayed when pylint has no access to further
+----------+------------+-------------------------------------------------------
```

👉 when we dive into designing a custom checker we'll see how to define a pylint message. See Defining a Message for the details about defining a new message or there is a list of available pylint messages id here.

⚠️ From the suggestions report, We've got 9 hints. as you can see all report messages are self-explanatory. for instance,

In line 1, column 0, the pylint checker name *missing-module-docstring* whose message id *C0114*, comes under the missing-module-docstring suggestion which means *we need to add a docstring at the top which refers to the use of the program written below that.*

😬 ***The score for the code given above is 2.50/10.0 (Very low)***

The score represents how good/bad your code is understandable by another programmer. We need to improve our code by considering the suggestions given in the report.

After, Pylint starts analyzing the source code and identify code that doesn't comply with the coding rules. You can then review the results. There may be false positives to dismiss. And there will be some issues that are more important to fix than others.

Step 5. Fix what needs to be fixed

So let's quickly refactor the code to solve these 9 problems

Next, you fix the issues that need to be fixed. Start with the most critical fixes. And go down the list from there.

```
#!/user/bin/env python3
'''
This program create Circle object, which is find area and perimeter and display
'''

PI = 3.14 # global constant PI to replace the magic 3.14 number.

class Circle:
    '''A class to represent a Circle.

    ..........
    Attributes
    ----------
    radius: str
        the radius of the Circle

    Methods
    -------
    area():
        Prints the Circle's area.
```

```
perimeter():
    Prints the Circle's perimeter.

'''
def __init__(self,radius:int) -> None:
    '''
    Constructs all the necessary attributes for the Circle object.

    Parameters
    ----------
        radius: str
            the radius of the Circle
    '''
    assert radius > 0 , \
        "circle radius must be a positive number"
    self.radius = radius

def area(self) -> str:
    '''calculate the area of the circle, return the result'''
    return PI * self.radius**2

def perimeter(self) -> str:
    '''calculate the perimeter of the circle, return the result'''
    return 2 * PI * self.radius

def __repr__(self):
    return f"{self.__class__.__name__}(radius={self.radius})"
```

The output should be something like this:

```
--------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

😊 Awesome! there are no errors in the code and the score is 10/10!

Now, let's write a simple pylint plugin using the basics discussed in the above concept.

2️⃣ Write a custom Pylint plugin

In this section, I'm going to teach you how to design and code a custom checker using the basics discussed in the first section of this tutorial.

We follow this specific guideline for our code. Our new checker will check to all Docstrings in a module, function, and class nodes must triple quoted with `"""`. Once we are done with this checker, Pylint should indicate all the places a `"""` has been used.

Now, we are going to write an AST + token Checker to achieve our objective. As mentioned earlier, Pylint can also handle raw filestreams .

We're going to do this in three stages:

1. Start by creating a checker class
2. Register the custom checker for pylint
3. finally, invoke the plugin

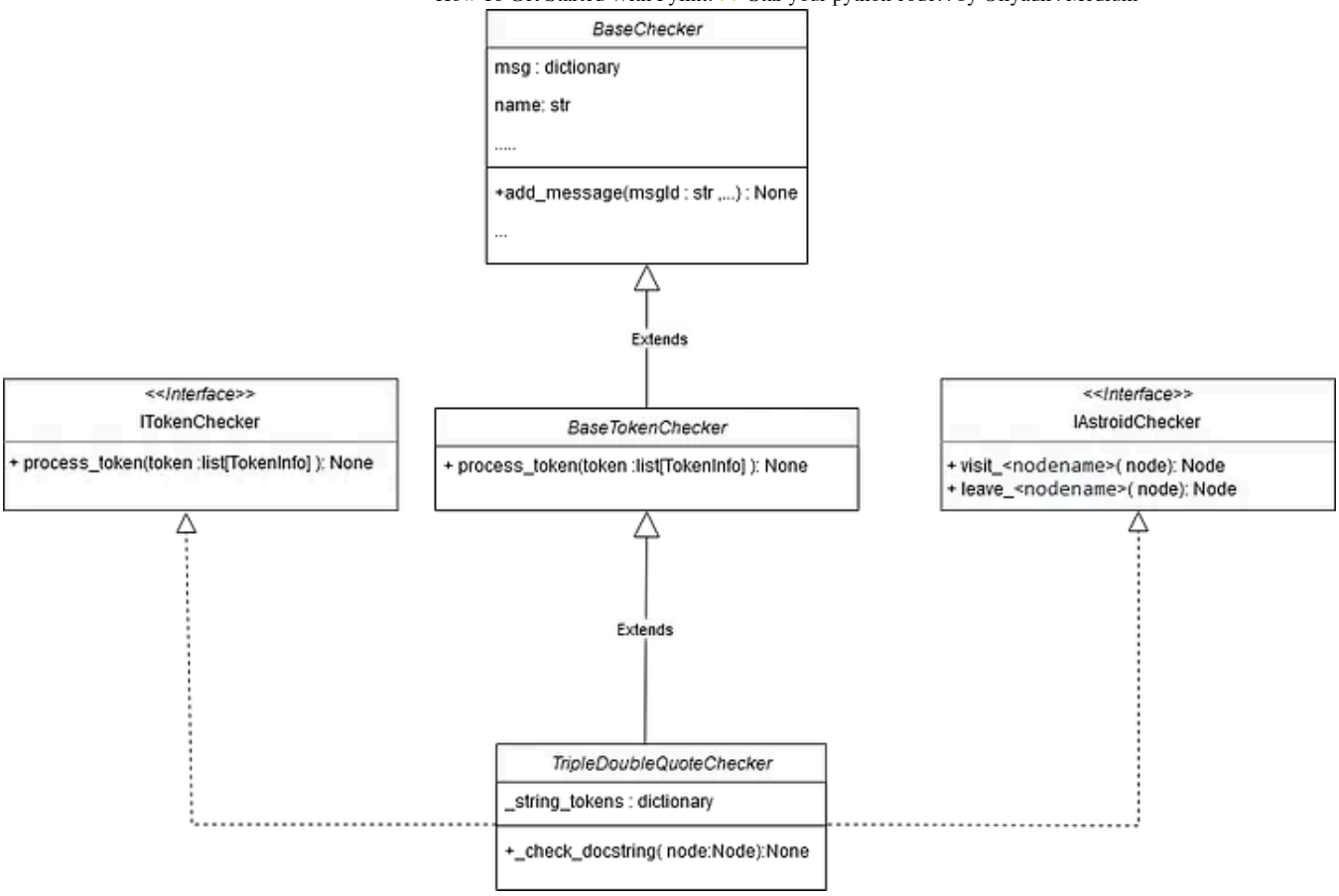
Step 1. Start by creating a checker class

Let's start with a new class called TripleDoubleStringChecker. The Pylint plugins need to be written in specific Checker classes that follow a certain format. This class will inherit from the BaseTokenChecker class of Pylint.

Firstly we will need to create a class and fill in some required boilerplate before we write the actual check.:

```
class TripleDoubleStringChecker(BaseTokenChecker):
    __implements__ =  ( IAstroidChecker , ITokenChecker )
    name = 'invalid-module-string-quote'
    msgs = {}
    _string_tokens = {}
```

Right. Let's break it down. The most interesting ones are



UML class diagram

1. BaseChecker — as the name implies, provides certain functionalities required by all checkers. This primarily includes the validation of the checker class that we implement.
2. BaseTokenChecker — inherits functionalities from BaseChecker in addition to this it include `process_tokens()` method that we use to collect a list of tokens in the source code.
3. IAstroidChecker — is an interface for checkers which prefers to receive events according to statement type.
4. ITokenChecker — is an interface for checkers that need access to the token list.
5. TripleDoubleStringChecker — class inherits and implements the properties and functions from the Pylint plugin architecture that allows you to extend its functionality easily. which is required to create our custom checker. your classes should only inherit BaseTokenChecker base class.

So far we have defined the following required components of our checker:

1. `__implements__` : Specifies what kind of checker we are creating. In this case, we are creating a combination of AST and token checker.
2. `name` : Short name(ID) of the checker that will be used in the configuration file.
3. `msgs` is the dictionary to which we add our error message. The dictionary has the following form:

```
{
 'message id': ('displayed message', 'message symbol', 'message help')
}

#displayed message = BRIEF_DESCRIPTION_OF_ERROR
#message symbol = CHECK_ID
#message help = A MORE DETAILED EXPLANATION OF ERROR
```

✔ we have seen a Pylint Error report before, you will instantly recognize this. The ERROR_CODE, CHECK_ID and the BRIEF_DESCRIPTION will show up in the report in the below format:

```
path_to_file:line_no:col_no: ERROR_CODE: BRIEF_DESCRIPTION_OF_ERROR (CHECK_ID)
```

Let's add our custom linter message, like below:

```
msgs = {
            'C1111' : (
                'Invalid string quote ',
                'invalid-module-string-quote',
                'Used when the string quote character does not match the
                 value triple-double string'
                ),
            }
```

4. `_string_tokens` : we initialize a stack to keep a list of return nodes for each function. we need to check quote usage via tokenization. as the AST walk will only

tell us what the doc is, but not how it is quoted. we need to store any triple quotes found during tokenization and check against these when performing the walk.

Next, you need to write the checker code.

👉 Pylint lets you define methods that get called as it traverses the AST. Methods are visit_module, visit_classdef, and visit_functiondef, and the visiting node is passed as an argument.

```
    def visit_module(self, node):
        """ Visit module and check for docstring quote consistency.
        Args:
            node: the module node being visited.
        """
        self._check_docstring(node)

    def visit_classdef(self, node):
        """ Visit class and check for docstring quote consistency.
        Args:
            node: the class node being visited.
        """
        self._check_docstring(node)

    def visit_functiondef(self, node):
        """Visit function and check for docstring quote consistency.
        Args:
            node: the function node being visited.
        """
        self._check_docstring(node)
```

As I mentioned before, the pylint leave method is called when it leaves the node. Here implementing leave<node_type> methods is not necessary. we override `process_tokens()` method from BaseTokenChecker derive class. what does this method do for us? to get a list of tokens from the source code and we filter and catch them in `_string_tokens` for further processing in each visiting node.

In Python tokenization, splitting up code into tokens. Each token is described by a named 5-tuple as documented in the tokenize library. The tuple takes the form:

```
( type, string, start, end, line)
```

```
#type -> The token type, for example, tokenize.STRING.
#string -> A string with the code that the token represents.
#start -> A 2-tuple (row, col) of the row and column where the token starts in
#end -> A 2-tuple (row, col) of the row and column where the token ends in the
#line -> A string with the line of code where the token was found.
```

You can use `tokenize.tokenize()` to see what these tuples look like. for example

```
from tokenize import tokenize
from io import BytesIO
txt = "print('hello world')"
## tokenize the helo world string
tokens = list(tokenize.tokenize(BytesIO(txt.encode('utf-8')).readline))

#output

[
 TokenInfo(type=62 (ENCODING), string='utf-8', start=(0, 0), end=(0, 0), line='
 TokenInfo(type=1 (NAME), string='print', start=(1, 0), end=(1, 5), line="print
 TokenInfo(type=54 (OP), string='(', start=(1, 5), end=(1, 6), line="print('hel
 TokenInfo(type=3 (STRING), string="'hello world'", start=(1, 6), end=(1, 19),
 TokenInfo(type=54 (OP), string=')', start=(1, 19), end=(1, 20), line="print('h
 ...
]
```

Based on this output, we can see that we need to check that the token type is `tokenize.STRING` and that the token string is stripped of any quotes. Then we keep a list of string tokens in the `_string_tokens` dictionary.

👉 `process_tokens()` method that implements this check:

```
    def process_tokens( self , tokens):
        """Process the token stream.
        This is required to override the parent class' implementation.
        Args:
            tokens: the tokens from the token stream to process.
        """
        for (tok_type, token, (start_row , start_col), _, _) in tokens:
            if tok_type == tokenize.STRING:
                token_text = token.strip('"').strip("'")
                self._string_tokens[token_text] = ( token , start_row , start_c
```

Let's write the checker itself. This is the simplest part. The code is very simple.

```
    def _check_docstring( self , node):
        """Check for docstring quote consistency.
        Args:
            node: the AST node being visited.
        """
        if node.doc in self._string_tokens:
            token , row , col  = self._string_tokens[node.doc]
            if not token.startswith('"""'):
                self.add_message('invalid-module-string-quote',line = row)
```

What's happening here? the method name _check_docstring does the actual check every time when the AST node is visited. The actual check is then, very simple. If a doc node is quoted by a single or double quote, Now we call the inherited `add_message` function from BaseChecker and pass in the error ID, and the line row itself. Pylint handles the rest.

Alright! That wasn't difficult now, was it?

So, our checker is done. the next step is How do we actually have pylint use this? we will see this in the next step.

Step 2. Register your custom checker for pylint

At the bottom of `pylint_plugin.py` , you need to register your new checker to pylint by calling `linter.register_checker()` . For example, here's how to register the checkers we wrote for hello world strings:

```
def register(linter):
    """ This required method auto-registers the checker during initialization.
    : param linter: The linter to register the checker to.
    """
    linter.register_checker(TripleDoubleStringChecker(linter))
```

Step 3. invoking the plugin

This step is valid only for this terminal session. A more permanent solution would be to add this export to your environment config file.

Now add the path of the python modules for which you want to generate the documentation to PYTHONPATH. If you don't know how to use PYTHONPATH? you can find out a helpful introduction from Psych 214 course.

✔ **If you successfully** add this file to your PYTHONPATH then you can check the current setting of environment variables, using the `os.environ` dictionary. It contains all the defined environment variables of the shell that started Python.

For example, you can check the value of the PYTHONPATH environment variable, if it is defined as:

```
import os
os.environ['PYTHONPATH']
'/home/my_user/code'
```

We'll use our previous source code ***simpl_circle_class.py*** for the test case and ran it through pylint. This is my output:

```
PYTHONPATH=. pylint --load-plugins=myplugin simple_circle_class.py
```

Now we can debug our checker!

```
************* Module simple_circle_class
simple_circle_class.py:10:4: C1111: Invalid string quote  (invalid-module-string-quote)
simple_circle_class.py:28:8: C1111: Invalid string quote  (invalid-module-string-quote)
simple_circle_class.py:41:8: C1111: Invalid string quote  (invalid-module-string-quote)
simple_circle_class.py:45:8: C1111: Invalid string quote  (invalid-module-string-quote)

-------------------------------------------------------------------
Your code has been rated at 6.92/10 (previous run: 10.00/10, -3.08)
```

🛠 Fixing this issue is very simple, replace all the source code doc strings with `"""` . and run again

```
-------------------------------------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
```

💥 Boom, Now everything is clean. Our checker worked like charm.

Anyway enough talk. In this tutorial, most of our attention is on the cleanup of source code. Once you are a fluent Pylint user, you are likely to turn your focus to intermediate-level Pylint topics that are beyond the scope of this introduction.

Some of the skills that you can learn:

- How to integrate Pylint in CI and Continuous Testing
- How to configure Pylint to co-operate with other Python tools such as *Black*, *Flake8*, *isort*, mypy, and so on.
- What should be in `pylintrc` to tune Pylint for large projects.
- How to configure Pylint to check for spelling errors.
- How to accommodate actual errors in Pylint.

For now, concentrate on getting to and staying at a quality score of 10, and enjoy the increasingly problem-free source code that Pylint helps you create.

Final word

Do you think you are smart enough to catch common errors in your source code without a code checker?

Probably, if your answer is yes, let's tell you about a fascinating movie famous phrase '***Fight in the shade!***' about *300* spartan soldiers fought nearly 4 million Persian soldiers and died at the Battle of Thermopylae.

Figure . King Leonidas at Thermopylae.

Remember that movie 300? Well, that actually originates from a true story. As you can see spartans have no chance because arrows will blot out the sun. The Spartan General is said to have laughed and replied *" then we shall have our battle in the shade! "* Here is where I was inspired. It makes no difference if you never make those mistakes or if you always catch them during code reviews. Any codebase has multiple developers working on it, and mistakes can occur anywhere. Think of all static code checkers as a developer shield🛡️ . You can get rid of extremely typical, detectable mistakes by enforcing them, And that gives the power to turn every obstacle into an aid towards victory.

Python Code Quality Software Pylint Static Code Analysis

Follow

Written by Oliyadk

2 followers · 6 following

I'm software developer , inspired by the power of Algorithms that change the world.

Responses (1)

Dragan Nikolic

What are your thoughts?

H

Hebtamu Teshome
Mar 3, 2023

nice introduction

Reply

More from Oliyadk

Oliyadk

What Happens When You Type https://www.google.com In Your Browser And Press Enter

For this topic, I will be explaining the following briefly :

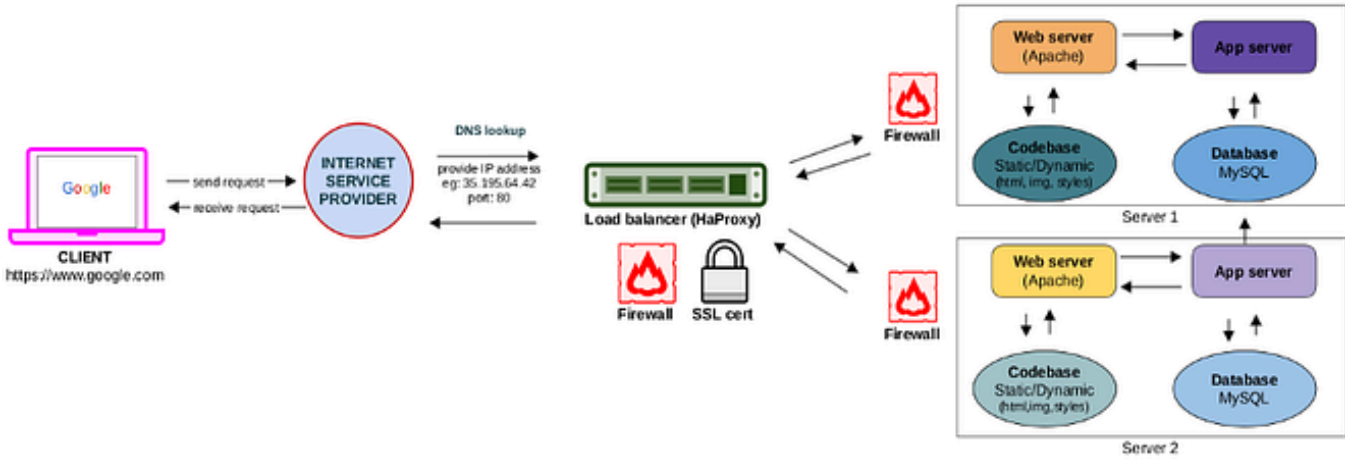
Oct 10, 2022

Oliyadk

በተመሰለ እውነታ ውስጥ የመኖር ትርክት ና የግዙፉ አጽናፈ ሰማይ (Universe) የስሌት ሂደት

The Matrix የሚለውን ፊልም አብዛኞቻችን አይተነዋል ። ከፊልሙ ማትሪክስ በአእምሮ ላይ የተሳለ ምናባዊ ዓለም ሲሆን በቀላሉ ሀልም ማለት ነው ። ከዚህ ስርአት ጋር የተገናኙ አእምሮዎች በምናባዊ...

May 27, 2022



Oliyadk

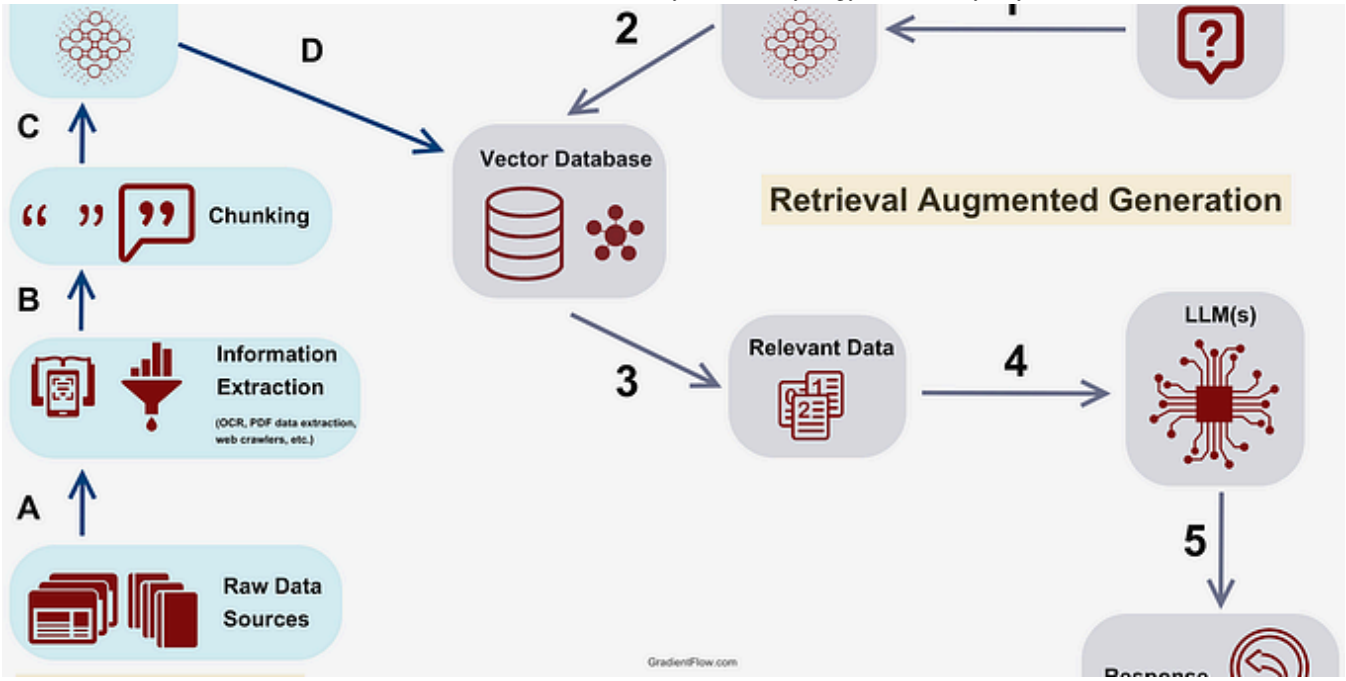
What Happens When You Type https://www.google.com In Your Browser And Press Enter

For this topic, I will be explaining the following briefly :

Oct 10, 2022 1

See all from Oliyadk

Recommended from Medium



In ITNEXT by Javier Ramos

You Don't Need RAG! Build a Q&A AI Agent in 30 Minutes 🚀

Is RAG Dead? Exploring simpler AI Agents alternatives by building tools that query the source data directly

Jun 10 1.2K 45

In Level Up Coding by Anmol Baranwal

The complete guide to building MCP Agents

MCP agents can now talk to real apps and actually get stuff done.

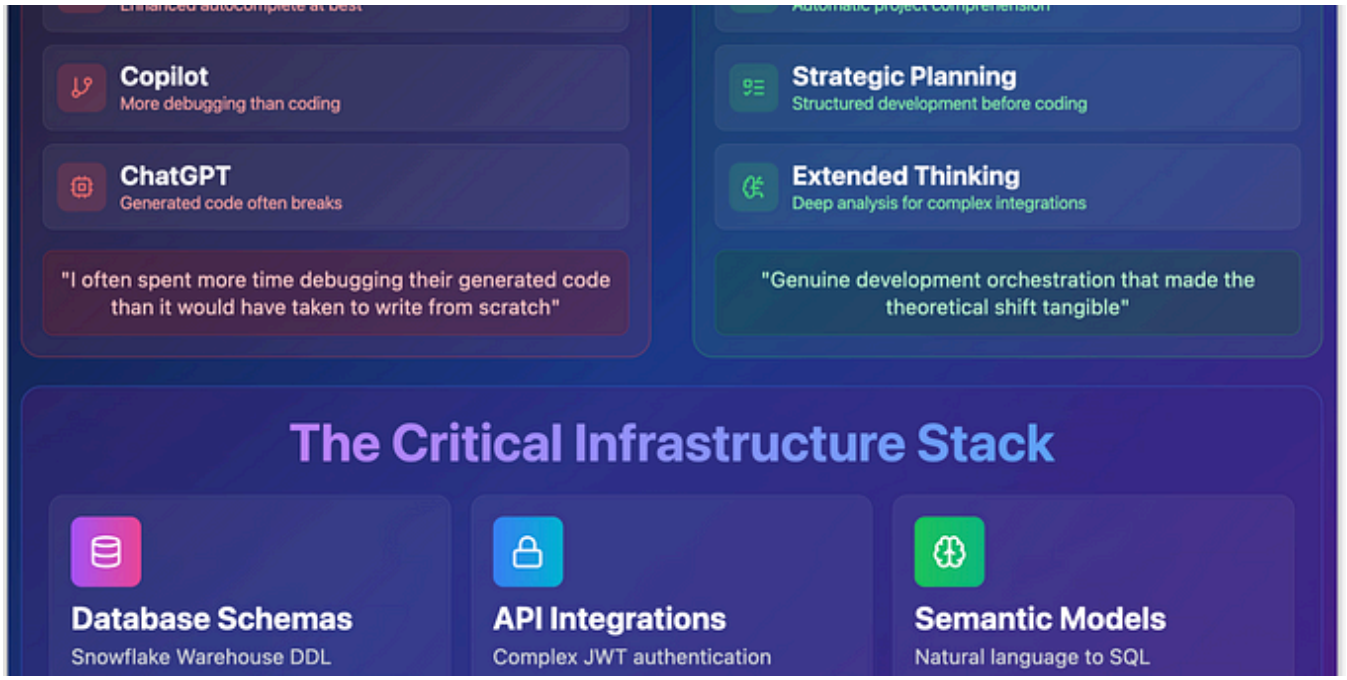
Jun 11 570 7

VerticalServe Blogs

Best Practices for Implementing Configuration Class in Python

Managing configurations for development (dev), user acceptance testing (uat), and production (prod) environments is a critical part of...

Jan 7 11

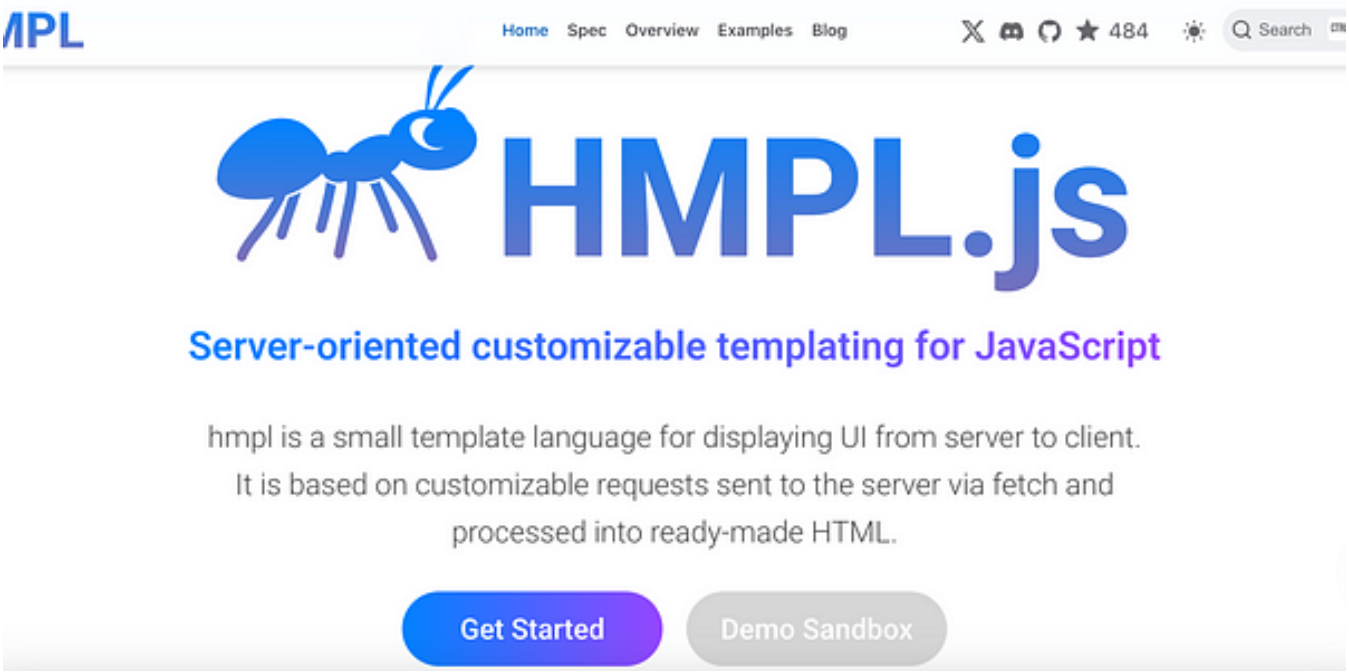


George Vetticaden

Claude Code: The Agentic Development Revolution That Made Me Cancel Cursor, Copilot, & ChatGPT —...

Watch MCP Servers, Snowflake Warehouse Schemas, and Cortex Analyst Semantic Models Come to Life in Minutes—Not Days

Jun 10 195 6



In Let's Code Future by TheMindShift

14 Open-Source Tools That Will Sharpen You Into an Ultimate Developer

Have you ever had that feeling? You spend countless hours learning new languages, frameworks, and design patterns—yet somehow, you're...

4d ago 331 2

Sohail Saifi

Kubernetes Is Dead: Why Tech Giants Are Secretly Moving to These 5 Orchestration Alternatives

I still remember that strange silence in the meeting room. Our CTO had just announced we were moving away from Kubernetes after two years...

Jun 7 2.3K 88

See more recommendations