A Comparative Analysis of Python Code-Line Bug-Finding Methods

Article · August 2023

CITATIONS
0

READS
280

4 authors, including:

Yasmin Makki Mohialden
Mustansiriyah University
102 PUBLICATIONS **227** CITATIONS
SEE PROFILE

Nadia Mahmood Hussien
Al-Mustansiriya University
80 PUBLICATIONS **105** CITATIONS
SEE PROFILE

Kapil Joshi
Uttaranchal University
215 PUBLICATIONS **972** CITATIONS
SEE PROFILE

Scientific Research Journal of Engineering and Computer Science
Abbreviated Key Title: *Sci Res Jr Eng Comp Sci. Sci;*
ISSN 2788-9394 (Print)
ISSN 2788-9408 (Online)

OPEN ACCESS

Volume-3 | Issue-4 | Jul-Aug-2023 |

Review Article

A Comparative Analysis of Python Code-Line Bug-Finding Methods

Yasmin Makki Mohialdenq[1], Nadia Mahmood Hussien[2], Esraa Jaffar Baker[3], Kapil Joshi[4]
[1]*Department of Computer Science, College of Science, Mustansiriyah University, Baghdad, Iraq*
[2]*Department of Computer Science, College of Science, Mustansiriyah University, Baghdad, Iraq*
[3]*Department of Computer Science, College of Science, Mustansiriyah University, Baghdad, Iraq*
[4]*Computer Science and Engineering Department, Uttaranchal University, Dehradun, India*
****Corresponding Author:***
Yasmin Makki Mohialdenq

Abstract: Python is a well-known programming language in many fields, including building websites, technical computing, machine learning, and data analysis. However, Python does have faults that may prevent it from operating as intended. Defects must be found and fixed for Python code to function at its best. This essay offers a comparative examination of several Python code bug-finding methods. This study aims to assess these approaches' efficacy and efficiency in locating defects at the level of specific code lines. The relevance of bug identification and its effect on Python programs are briefly discussed in the first section of the article. Then it examines several methods for finding bugs, like testing procedures, static analysis, and dynamic analysis. There is a detailed discussion of each approach's benefits, constraints, and use cases.

Keywords: bug detection-static analysis, syntax errors, unused variables, unreachable code, effective bug-finding methods, Python programming.

1. INTRODUCTION

A key component of software engineering is ensuring the integrity and dependability of the program, and bug identification is a critical component of this process. The research community has paid much attention to finding faults in Python code. Static analysis, rule-based approaches, and machine learning-based techniques have all been suggested as ways to find bugs; each has its benefits and drawbacks [1].

This study includes the source code for a Python code-line bug detection system that aims to improve the quality of Python code by locating and disclosing mistakes at the line level. Static analysis, rule-based inspections, and machine learning-based approaches are all used in the methodology to accomplish accurate and thorough bug discovery. The source code may be used as a stand-alone program or a plugin in well-known integrated development environments.

We evaluate the Python code-line bug detection source code's efficacy by contrasting it with well-known programs like PyLint [2] and Flake8 [3]. PyLint is a powerful static Python code analyzer, and Flake8 scans code for possible defects and problems in style and formatting using pre-established criteria [4] [5] [6] [7]. We show the benefits of our method by demonstrating how our source code finds mistakes that other tools could miss using code snippets.

II. LITERATURE REVIEW

According to research [8], the authors created Hyperstyle, a flexible build tool that builds on several scripts and expert code checkers to satisfy various programming needs while preserving flexibility. The programming languages Python, Java, Kotlin, and Javascript are all supported by Hyperstyle, which may be used alone or as a component of a Massive Open Online Course (MOOC) platform. Notably, the writers combined Hyperstyle with two well-known educational media, JetBrains Academy and Stepik, and by 2021, they were handling close to 1 million entries every week. Integrating Hyperstyle into these instructional platforms allows learners to receive real-time feedback and help them develop their programming abilities.

Quick Response Code

Journal homepage:
https://isrpgroup.org/srjecs/

Article History
Received: 30.07.2023
Accepted: 10.08.2023
Published: 19.08.2023

In another study, PyTA was created as a wrapper for pylint, a popular static code analysis tool for Python [9]. PyTA delivers better notifications to help students properly address the highlighted problems and enables custom tests to identify frequent rookie mistakes precisely. By incorporating PyTA into an already-existing online platform used to provide programming assignments to CS1 (Computer Science 1) students, the authors assessed PyTA's effectiveness. Exercise submissions obtained via the integrated system were compared to previously acquired data as part of the assessment. The findings revealed that students who viewed and used the PyTA output performed better at completing programming tasks, took less time to fix problems, and made fewer recurrent errors. This shows that PyTA and static analyses, in general, may help students identify functional issues in their code that may not be immediately apparent from compiler input. Static analysis output may also help students discover and debug their code more rapidly, improving learning results.

Additionally, research by [10] presented PyDFix, a specific tool made to identify and fix Python builds that are unreplicable as a result of dependency problems. The authors used two extensive bug datasets, BugSwarm and BugsInPy, drawn from open-source projects to assess PyDFix. Two thousand seven hundred-two builds were analyzed for the assessment, and 1,921 (71.1%) were irreproducible due to dependency mistakes. PyDFix generated complete repairs for 859 (44.7%) of the detected non-reproducible builds and partial solutions for another 632 (32.9%) builds. The results demonstrate PyDFix's effectiveness in resolving a significant fraction of the reproducibility problems, highlighting its potential to improve the dependability and reproducibility of Python projects.

Our contribution to this paper: to evaluate the usefulness and efficiency of several bug-finding techniques for Python programs, this article compares such approaches. The introductory part emphasizes the importance of bug detection and its effect on Python scripts. Testing processes, static analysis, and dynamic analysis are bug-finding techniques we examine. We also thoroughly overview each technique's benefits, drawbacks, and appropriate use cases.

In this paper, we significantly contribute to Python development and software engineering through a thorough review of various bug-finding methods. This study may be used by programmers, researchers, and practitioners to make educated choices on the best bug-finding way for their Python projects, eventually leading to better code quality, performance, and dependability. This paper tackles the dearth of thorough comparisons and advances bug detection techniques in Python programming by outlining the advantages and disadvantages of each strategy.

III. PYTHON CODE-LINE BUG DETECTION METHODS

Numerous methods with specific advantages and uses may be used to find bugs in Python code. Typical techniques include:

1. Static Code Analysis: This method focuses on finding possible flaws by scrutinizing the syntax and structure of the code while studying it inactively. PyLint, PyChecker, and Flake8 are well-known Python tools for static code analysis. These programs may need help with syntax, unnecessary variables, and other potential issues. Table I provides a comparison of these three tools. To sum up, PyLint, PyChecker, and Flake8 are all valuable tools for finding and fixing problems in Python code. They vary in performance, flexibility, integration possibilities, and functionality. The developer's or development team's unique demands and tastes will determine the best tool.

2. Unit testing is a technique for verifying the functioning of brief code sections by creating test cases specifically for them. It guarantees that individual code units function correctly and effectively and aids in the bug detection process before release. The well-known Python unit testing frameworks include unit test, pytest, and nose. Numerous Python projects, such as web applications, libraries, scientific computing, and game development, might benefit from unit testing. It helps bug discovery and ensures that every code works as intended[17]. Table II contains examples of Python applications for unit testing.

3. Code Reviews: A group of developers examines the code to increase its quality and find errors. Manual code reviews are possible, as are automated ones using programs like CodeClimate and ReviewBoard. They are essential for ensuring the code's readability, maintainability, and quality. Developers may improve the quality of the codebase and lower the probability of defects and mistakes by performing rigorous code reviews. Table III provides examples of how to perform code reviews for Python programs.

4. Debugging Running the code while using a debugger to go through it to detect flaws is called debugging. It is often a last option if other bug detection techniques fail. Python has a built-in debugger that enables programmers to walk through code and find problems. Python code may also be debugged with debuggers like PDB and PyCharm. Table IV lists several examples of Python code debugging approaches.

5. Code Coverage Analysis: This technique assesses how thoroughly the source code has been checked and identifies untested regions that could have flaws. For this analysis, you may use programs like PyCharm and Coverage.py, which highlight code that does not have test coverage and flag possible problems.

These bug detection techniques allow programmers to enhance the Python code's quality, dependability, and maintainability, ensuring its maximum performance and reducing the likelihood of defects.

IV. RESULTS AND DISCUSSION

Static code analysis may be a more rapid and effective technique to find possible problems in code than other approaches since it does not require the code to be run. False positives might be found, and it might only catch some problems. On the other hand, developing test cases for each code unit may be time-consuming. Unit testing, however, is a solid strategy for finding errors. When other techniques are ineffective at finding the fault, debugging is often utilized as a last option but is time-consuming.

Combining these techniques may ensure that the code is error-free and operates as intended. PyLint, PyChecker, and Flake8 are three well-known tools for static code analysis, while unit tests, pytest, and nose are three well-known tools for unit testing. The material mainly concentrates on these two techniques. The text compares and summarizes the attributes of various tools and frameworks in two tables, including their customization possibilities, integration possibilities, and performance.

PyChecker, PyLint, and Flake8 are contrasted in Table I. Description of PyLint describe it as a thorough tool that examines code for possible flaws, coding style problems, and other things. It contains a sizable configuration file that may be used to change its behavior and provides several customization options for code analysis. Editors and integrated development environments (IDEs) like PyCharm, Sublime Text, and Visual Studio Code may all be integrated with PyLint. However, compared to the other two tools, it is slower. PyChecker, on the other hand, is referred to as a more straightforward tool that focuses on spotting typical code faults and probable blunders. It produces a comprehensive report that includes the issue's description, line number, and file name. PyChecker may be launched from the command line using Python programming environments and editors. Despite no longer being actively developed, its functionalities have been incorporated into other Python tools like PyLint and Flake8. A combination of PyFlakes, pycodestyle, and McCabe called Flake8 provides a thorough analysis of Python code that may spot issues, including inappropriate indentation, undefined variables, and syntax mistakes. It may be run using a command-line interface or integrated with several Python programming environments and editors. There are some customization possibilities, although they are not as many as PyLint's. Comparing Flake8 to PyLint and PyChecker, the latter gives a quicker analysis, while the former offers a more thorough code analysis.

Comparing unit tests, pytest, and nose is shown in Table II. A unit test, a standard library component, supports Python's unit testing. As opposed to Pytest and Nose, it has a more verbose syntax. Unlike unit tests, Pytest offers a more streamlined and succinct syntax and is lauded for its robust and adaptable testing infrastructure. It includes cutting-edge functions like fixtures, parameterized testing, and test discovery, making it flexible and able to work with a wide range of testing plugins and tools. The nose is a minimal testing framework that maintains compatibility with unit tests and supports a plugin design. The syntax is Straightforward to understand. Despite this, it is not advised for new projects because of its deprecated state.

Overall, the paper thoroughly examines the many frameworks and techniques that may be used to identify flaws in Python programs. Developers can make educated decisions based on their unique requirements and preferences by highlighting the critical characteristics and distinctions between various tools and frameworks.

TABLE I. COMPARISON OF PYLINT, PYCHECKER, AND FLAKE8.

Factor	PyLint[11]	PyChecker[12]	Flake8[13]
Features	a complete tool that tests for code style flaws, bugs, and more.	A simplified tool that identifies typical code problems and possible faults.	Combines PyFlakes, pycodestyle, and McCabe to analyze Python code thoroughly
Integration	integrates with PyCharm, Sublime Text, and Visual Studio Code.	Runs from the command line or in Python programming environments.	Executable from the command line or incorporated into Python programming environments and editors.

Factor	**PyLint[11]**	**PyChecker[12]**	**Flake8[13]**
Features	a complete tool that tests for code style flaws, bugs, and more.	A simplified tool that identifies typical code problems and possible faults.	Combines PyFlakes, pycodestyle, and McCabe to analyze Python code thoroughly
Customization	provides comprehensive code analysis customization and a large configuration file. Behavior.	An essential tool with minimal modification possibilities.	Provides fewer customizing possibilities than PyLint.
Performance	albeit slower than the other two, it analyzes code more thoroughly.	Quicker yet restricted code analysis.	Analyzes code quicker than PyLint and more thoroughly than PyChecker.

TABLE II. SUMMARIZES THE COMPARISON OF UNIT TEST, PYTEST, AND NOSE.

Factor	**Unittest[14]**	**Pytest[15]**	**Nose[16]**
Syntax	is part of the Python standard library and more verbose than pytest and nose.	The syntax is shorter and simpler.	It is similar to Unittest but has more features.
Assertions	contains limited built-in assertion methods. AssertEqual, assertTrue, and assertIn check test code conditions in it	expands built-in claims and permits new ones. It provides comprehensive assertion introspection, rewriting, and assisting for a subset of unittest assertions.	Provides similar assertion features to unit test
Test discovery	using a predefined naming convention to find tests is less flexible than pytest's method.	Strong test discovery technique that automatically detects and runs tests without a naming convention.	Similar to the unit test
Fixtures	includes rudimentary fixture support in setUp and tearDown.	Enables more sophisticated setup and disassembly with its robust fixture system.	A unit is test-like short.
Plugins and extensions	It limits plugins and extensions but permits custom test runs and loaders.	Offers a large ecosystem of plugins and extensions for test coverage, code profiling, and test parallelization.	Test coverage and parallelization with fewer plugins and extensions than pytest.

Factor	Unittest[14]	Pytest[15]	Nose[16]
Syntax	is part of the Python standard library and more verbose than pytest and nose.	The syntax is shorter and simpler.	It is similar to Unittest but has more features.
Test organization	Fixtures (setup and teardown methods) and parameterized tests are available. However, tests must be in classes and methods.		

TABLE I I I. SOURCE CODE EXAMPLES FOR RESIZED CODE REVIEWS

debug Python code name	Definition	Source code example

Printing statements	One of the easiest methods to debug Python code is to use print statements to show variable, expression, and other values at precise moments.	`def calculate_average(numbers):`
		`  total = 0`
		`  for number in numbers:`
		`  total += number`
		`  print(f"Number: {number}, Total: {total}")`
		`  average = total / len(numbers)`
		`  return average`

Debuggers	Several Python debugging tools let you walk through code, create breakpoints, and analyze variables and data structures in real time.	`import pdb`
		`def calculate_factorial(n):`
		`  result = 1`
		`  for i in range(1, n+1):`
		`  result = result * i`
		`  pdb.set_trace()  # Set breakpoint`
		`  return result`
		`calculate_factorial(5)`

Error messages	Python interpreters produce error messages that describe the kind of issue, its location, and other characteristics. Reading and interpreting error messages helps you find and repair code issues.	Traceback (most recent call last): File "example.py", line 7, in <module> result = 1 / 0 ZeroDivisionError: division by zero

TABLE 1V. EXAMPLES OF PYTHON CODE DEBUGGING.

Code review name	Definition	Source code example

Style and formatting	PEP 8 style guide and project code standards. Indentation, variable naming, and line length are usually stylistic considerations.	# Bad code
		def myFunction(param1, param2):
		if param1 == 1 or param2 == 2:
		print("Hello World")
		else:
		print("Goodbye World")
		# Good code
		def my_function(param1, param2):
		if param1 == 1 or param2 == 2:
		print("Hello, World!")
		else:
		print("Goodbye, World!")

Code review name	**Definition**	**Source code example**

Style and formatting	PEP 8 style guide and project code standards. Indentation, variable naming, and line length are usually stylistic considerations.	# Bad code
		def myFunction(param1, param2):
		  if param1 == 1 or param2 == 2:
		  print("Hello World")
		  else:
		  print("Goodbye World")
		# Good code
		def my_function(param1, param2):
		  if param1 == 1 or param2 == 2:
		  print("Hello, World!")
		  else:
		  print("Goodbye, World!")

functionality and correctness	Examine the code's edge case functionality. Verify that the code works as expected.	# Bad code
		def calculate_factorial(n):
		  result = 1
		  for i in range(n):
		  result = result * i
		  return result
		# Good code
		def calculate_factorial(n):
		  result = 1
		  for i in range (1, n+1):
		  result = result * i
		  return result

Comments, documentation	Review code and documentation. Simple code and comments explain purpose, operation, and use.	# Bad code
		def calculate_average(numbers):
		  """Calculates the average of a list of numbers"""
		  total = sum(numbers)
		  length = len(numbers)
		  return total / length
		# Good code
		def calculate_average(numbers):
		  """Calculate the average of a list of numbers.
		  Args:
		  numbers (list): A list of numbers.
		  Returns:
		  float: The average of the list of numbers.
		  """
		  total = sum(numbers)
		  length = len(numbers)
		  return total / length

Performance and optimization	Remove unnecessary operations and improve the code for speed and efficiency. Use the proper algorithms and data structures in the code.	# Bad code
		def find_largest_number(numbers):
		  largest = 0
		  for number in numbers:
		  if number > largest:
		  largest = number
		  return largest
		# Good code
		def find_largest_number(numbers):
		  return max(numbers)

V. CONCLUSION

This study compared line-level Python code fault detection methods. Debugging, unit testing, and static code analysis were discussed. Each method has pros and cons.

Using static code analysis, finding bugs without executing the code is rapid and effective. However, it may miss specific issues and provide false positives. However, unit testing has successfully discovered faults by building test cases for code units. Despite its efficacy, unit testing may take time to generate detailed test cases. Debugging is the last resort when other approaches fail. It may take time, particularly with complex issues.

The research examined PyLint, PyChecker, and Flake8 is one of three popular static code analyzers. Each tool offered different levels of comprehensiveness, customizability, and integration. More complicated and customizable than PyChecker and Flake8, PyLint is slower. The simpler PyChecker, which targets common code mistakes, has yet to be maintained. Flake8 analyzes code better and faster than PyLint.

Comparing unit tests, pytest, and nose for unit testing, Unittest, part of the Python standard library, has a more verbose syntax than Pytest and Nose. Pytest is popular among developers because of its simplicity, syntax, adaptability, and robustness. The nose is still functional, but its deprecation makes it unsuitable for new projects. The paper stressed the need for thorough code reviews to improve code quality and reduce errors. Numerous debugging approaches were identified as effective ways to detect and fix Python code errors, improving dependability.

Future studies may explore how machine learning and AI enhance bug discovery. New testing frameworks and static analysis tools may improve problem-finding. Real-time problem detection and automated bug remediation research may improve software development. Developers may use many bug detection methods and suitable tools to ensure their Python code is reliable, trustworthy, and efficient.

REFERENCES

1. S. Axelsson, D. Baca, R. Feldt, D. Sidlauskas, and D. Kacan, "Detecting defects with an interactive code review tool based on visualization and machine learning," in *the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009)*, 2009.
2. H. Gulabovska and Z. Porkoláb, "Survey on Static Analysis Tools of Python Programs," in *SQAMIA*, 2019.
3. T. Jenness, "Modern Python at the Large Synoptic Survey Telescope," *arXiv preprint arXiv:00461*, 2017.
4. A. Martelli, A. M. Ravenscroft, S. Holden, and P. McGuire, *Python in a Nutshell*. "O'Reilly Media, Inc.", 2023.
5. P. Alisa, "Prostředí pro testování a připojení simulátoru výhybek," České vysoké učení technické v Praze. Vypočetní a informační centrum., 2022.
6. V. Ahlström, "Improvement of simulation software for test equipment used in radio design and development," ed, 2023.
7. D. Brodmann and E. Rodner, "OpenPredict-An Open Research Dataset and Evaluation Protocol for Fine-grained Predictive Testing," *Angewandte Forschung*, p. 287.
8. A. Birillo *et al*., "Hyperstyle: A tool for assessing the code quality of solutions to programming assignments," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, 2022, pp. 307-313.
9. D. Liu and A. Petersen, "Static analyses in Python programming courses," in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 2019, pp. 666-671.
10. S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for Python build reproducibility," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 439-451.
11. "pylint · PyPI." Accessed: Aug. 05, 2023. [Online]. Available: https://pypi.org/project/pylint/
12. N. N. Inc MetaSlash, "PyChecker: Python source code checking tool." Accessed: Aug. 05, 2023. [OS Independent]. Available: http://pychecker.sourceforge.net/
13. "flake8 · PyPI." Accessed: Aug. 05, 2023. [Online]. Available: https://pypi.org/project/flake8/
14. "unit test — Unit testing framework," Python documentation. https://docs.python.org/3/library/unittest.html (accessed Aug. 05, 2023).
15. "pytest: helps you write better programs — pytest documentation." https://docs.pytest.org/en/7.4.x/ (accessed Aug. 05, 2023).
16. J. Pellerin, "nose: nose extends unit test to make testing easier." Accessed: Aug. 05, 2023. [OS Independent]. Available: http://readthedocs.org/docs/nose/
17. Y. M. Mohialden, N. M. Hussien, and S. A. Hameed, "Review of Software Testing Methods," Journal La Multiapp, vol. 3, no. 3, pp. 104-112, 2022. Available: https://doi.org/10.37899/journallamultiapp.v3i3.648