Python syntax, call graph and variable type analysis tool based on Pylint

Zhimao Lin
Department of Computing Science
University of Alberta
Edmonton Canada
zhimao@ualberta.ca

Yi Zhang
Department of Computing Science
University of Alberta
Edmonton Canada
yi16@ualberta.ca

Wang Dong
Department of Computing Science
University of Alberta
Edmonton Canada
wdong2@ualberta.ca

ABSTRACT

Numerous static analysis techniques have recently been proposed for Python programming language. Most of them are aiming at the industry development. Yet, the static analysis configurations used for the industry are not always applicable for educational Python programmers such as university students. There is a specific software quality requirement list for CMPUT174 students at the University of Alberta. Currently, all of the requirements are checked manually by teaching assistants. The key idea of our work is to implement one or a series of program checkers to fit the software requirement at our university so that it could automatically check the software quality and give an overall software quality grade to a student's project. As an example of our approach, we built a variable naming checker to determine whether all the variables are named by "snake case" style. We have applied test-driven development and have written our own test cases to discover all the syntax problems in a Python program.

In order to gain more experience in making the program analysis tool, we also discussed on how to implement a call graph generator by static analysis using Abstract Syntax Tree (AST) and checker framework from Pylint. Based on the idea of over-approximations, we can perform a sound and fast static analysis to generate call graphs.

CCS CONCEPTS

• Software and its engineering → Automated static analysis;

KEYWORDS

Static analysis, Pylint, Syntax Checker, Call graph

ACM Reference format:

Wang Dong, Zhimao Lin and Yi Zhang. 2019. Python Syntax, and Call Graph Analysis Tool Based on Pylint.

1 Introduction

Static program analyses are the fundamental for software development. They form the basis of the program optimization, enable refactoring in IDEs, and detect errors [1]. In order to make the program analysis useful in practice, they must follow the requirements for the specific user. In this work, we research on how to implement our own program checkers for University of Alberta 100-level Python program courses. After discovering the Pylint structure, we implemented some new checkers. In this way, a more complex program checker with enhanced functionality could be implemented based on the current checkers.

Programing syntax refers to the spelling and grammar which is describable by any algorithmic process whatsoever [2]. Syntax errors are the elementary type of error which is fatal. There is no way to execute a piece of code with a syntax error. A good static program checker should always have efficient syntax error checking functionality.

Pylint is a source-code, bug and quality checker for the Python programming language. It follows the style recommended by PEP 8, the Python style guide [3]. It includes many features like Coding Standard Check. In this paper, we focus on the syntax checking and call graph generation based on the existing Pylint structure.

The call graph generator is another feature we implemented into Pylint. The call graph is a useful data representation for control and data flow programs which investigates inter-procedural communication [4]. Our work shows how to generate the Python call graph statically within Pylint. Based on our research, currently, there is no open source static call graph generator for Python. A static call graph generator remains unsolved for many programming languages due to its complexity and high level of difficulty. Therefore, we have made a working static call graph generator for Python based on traversing Abstract Syntax Tree (AST).

As a result, our work is successful. We have passed all of our own test cases for python syntax checkers. For the call graph generator, we are able to find all the possible function calls including nested calls.

Our work makes the following contributions:

1. We implemented our own checkers to match CMPUT174 software quality test based on Pylint.
2. We made a static Python call graph generator based on Pylint analysis flow.

The remainder of this paper is structured as follows: Section 2 provides the background of our project. Section 3 presents our implementations for the project. Sections 4 outlines the results and

The artifact is available at https://github.com/ZhimaoLin/pylint.

Section 5 discusses our project limitations. In section 6, we list the related work. We finalize the paper by section 7 and section 8 which are the conclusion and future work.

2 Background

2.1 Pylint

Pylint is a Python program checker that tries enforcing coding standards [5], which establishes the foundation of our project. As one of our main objectives, Pylint can help us to check the coding styles of students' assignment. Pylint has many predefined standards. It reports coding style violations of a Python code. With the existing features of Pylint, we can easily add or remove checking rules as needed.

Pylint is powered by a Python library called Astroid. Astroid parses Python code into AST, so that we can analyze the code by traversing the AST [6]. After the code is parsed, each line of the code is transformed to a node with its corresponding type. For example, a function definition, "def f()," is transformed into a "functiondef" node. A function definition code usually contains many lines of code, so a "functiondef" node has an attribute body which contains all the codes in this function. The codes inside a node's body are also nodes with their corresponding types. Overall, after a python program is parsed by the Astroid, it is translated into different kinds of nodes, which form a forest.
As Pylint is open sourced, we are able to study and observe the way Pylint checks through each code with its checker. Each checker has four parts which are priority, message dictionary, options, and code node visitors [7]. A checker's priority will determine the order of checkers run. Higher priority checkers would run first. The message dictionary stores the error message for each coding style. The error message is displayed only when that coding style is violated. Options store the parameters of this checker. For instance, if a checker checks whether the maximum number of parameters of a function exceeds the limit, the limit would be declared in the options. Code node visitors is the most essential part in the checker. It is constituted by many functions defined by the Pylint that visit each type of code node parsed by the Astroid. During the visiting, if a checker finds a violation, it reports the error message. Suppose the checker has a node visitor called "visit_functiondef(node)". If the input code has two function definitions which means it has two functiondef type nodes. When that checker runs, "visit_functiondef(node)" will be called twice to visit each "functiondef" node, and the "functiondef" node will be assigned to visitor's parameter. By understanding the data structure inside Pylint, we are able to implement the new functionalities of each checker. Some visitor functions will be discussed in section 3 with their meanings. One example of the functions would be the "close" function in which it will be called after the checker has visited every node.

2.2 Call Graph

Call graph represents the call relations between functions. Call graphs can be either dynamic or static [4]. A dynamic call graph can be exact, but only describes one run of the program. A static call graph is a call graph intended to represent every possible run of the program. The exact static call graph is an undecidable problem, so that static call graph algorithms are generally over approximations. That is, every call relationship that occurs is represented in the graph, and possibly also some call relationships that would never occur in actual runs of the program.

Call graph is a useful tool for program analysis, it can be used for finding never called functions, dead code, program documentation, and tracking the flow between functions [8]. For beginners, call graph is also great tool for them to understand the structure of python program since it is made easy for them to debug codes.

The call graph we implemented is a static call graph. Our analyzer will generate the call graph without running the program. With the tradeoff of precision and sound, we have chosen precision. At each branch of the program flow, we present call graph for all flows. Calls defined in the input Python program will be reported by our call graph generator. We could get precise call graph even when a function call involves many layers of other functions. Our call graph implementation is still based on Pylint. Since Pylint's checker can traverse the AST of Python program, we have taken the advantage of Pylint to get the definition of each defined function. Then we generate call graph by looking at function node body to know what functions are called.

3 Implementation of the Syntax Checker and Call Graphs

3.1 Setup Working Environment

When we started our implementation, there was a problem that the part of our modified Pylint code was not found by Pylint. It only ran the original Pylint installed on the computer. Then, we wrote a shell script called "run.sh," which installs our Pylint code first before running Pylint. In this way, we can finally run and test our Pylint code.

3.2 Syntax Checker

Once we figured out how to run our Pylint code, we started to implement the syntax checker. We used test-driven development (TDD) method to implement the syntax checker [9]. Firstly, we created some test cases for the problems that are listed in the requirements. Then, we created a Pylint checker class called "UaCmput174Checker" followed by a register function, which registers our checker to Pylint. After that, we defined the checker name and priority, which are "ua-cmput174" and the lowest priority. The reason for setting a lowest priority is to prevent disturbing the messages generated by original Pylint checkers. Secondly, we have defined an error message dictionary and an options tuple. Each message in the message dictionary uses a message ID as its key, and its value would contain the display message, message name, and help message of the problems detected [7]. The message ID consists of a letter and 4 numeric digits. The display message is the message displayed on Terminal

once there is such a problem in the input Python file. The options tuple contains some configuration value of the checker. At last, we have implemented some node visitor functions and close function. All the check conditions are in those functions. Once a problem is detected, “add_message()” function will be called to show the corresponding error message after the analysis.

3.3 Call Graph Generator

We have created the call graph generator by leveraging the Abstract Syntax Tree (AST) and checker framework from Pylint. Pylint can generate an AST of the input Python program using the Astroid library, and therefore we are able to access the AST in the checker framework. The call graph generator visits all function definitions using “visit_functiondef()” function in the checker framework, and it stores the information into a dictionary. After visiting each function definition, it starts from the main function and recursively finds all functions stem from the main function. If a function is a Python built-in function, it would just show the function call. On the other hand, if a function is a user-defined function, it would show the call graph of that function based on the function definition. However, we have decided to not show the call graph of a user-defined function in the assignment statements due to the readability problem of the output result. Eventually, the call graph is printed after the linter message.

4 Result

4.1 Syntax Checker

Our syntax checker can check the following issues:

1. Main function
   - 1.1. No main function
   - 1.2. Main function is not the first function in the file
   - 1.3. Main function is not called
   - 1.4. Main function is called multiple times
   - 1.5. Main function
2. Comments
   - 2.1. No block comments before a function
3. Naming convention
   - 3.1. Class names do not follow the camel-case style
   - 3.2. Other names do not follow the snake-case style
4. Duplicate code
   - 4.1. Two adjacent lines contain duplicate code
5. Literal
   - 5.1. Some literals are repeated without storing in constant variables except for 0, 1, 2, -1, 0.0 and "
   - 5.2. Literal assignment statements in a function are not within 5 lines of the function name
6. Functions
   - 6.1. Functions have more than 5 arguments
   - 6.2. Functions have more than 12 statements
7. Methods in a Class
   - 7.1. Methods have more than 5 arguments
   - 7.2. Methods have more than 12 statements

4.2 Call Graph Generator

The output of the call graph generator starts with the main function call. The indentation shows the relationship between the caller and callee. Functions with less indentation level are callers, and those with more indentation level are callees. Figure 1 shows an example of the output of the call graph generator.

```
============================
|         Call Graph         |
============================
main()
    print('Call function f1.')
    f1()
        print('Function f1 is called.')
        f2()
            print('Function f2 is called.')
```

Figure 1: An Example of Call Graph Output

The main function calls function “print” and “f1”. Then, function “f1” calls function “print” and “f2.” After that, function “f2” calls function “print.” The call graph generator just displays the “print” call since it is Python built-in function. However, it shows the call graph of function “f1” and “f2” because they are user-defined functions.

In addition to generating call graphs of user-defined functions, the call graph generator can also generate call graphs of “if” clause, for loop, while loop, and “try” clause.

As for “if” clause, it shows both true branch and false branch. For example, Figure 2 and 3 show an “if” clause and the generated call graph.

```
if apple <= 3:
    print("True branch")
elif apple > 3 and apple <= 7:
    print("Else if branch")
else:
    print("False branch")
```

Figure 2: Python If Clause

```
if True
    print('True branch')
else False
    if True
        print('Else if branch')
    else False
        print('False branch')
```

Figure 3: Call Graph Output of Figure 2

“If True” shows the true branch, and “else False” shows the false branch. If there is an “else if” branch, it is decomposed into an inner true and false clause.

As for the for loop and while loop, it only shows the body of the loop. For example, Figure 4 and 5 show two loops and its call graph.

```
for i in range(0, 10):
    print("For loop body")

i = 0
while i < 10:
    print("While loop body")
```

Figure 4: Python Loop

```
for loop:
    print('For loop body')
while loop:
    print('While loop body')
```

Figure 5: Call Graph Output of Figure 4

The call graph can show the name of the loop whether it is for loop or while loop. Then, it shows the body of the loop with 1 deeper indentation.

As for the "try" clause, it shows the "try" branch, "except" branch, and "finally" branch if there is "finally" branch. For example, Figure 6 and 7 show a "try" clause and its call graph.

```
try:
    print("Try clause")
except:
    print("Except clause")
finally:
    print("Finally clause")
```

Figure 6: Python Try Clause

```
try:
    print('Try clause')
except:
    print('Except clause')
finally:
    print('Finally clause')
```

Figure 7: Call Graph Output of Figure 6

Besides flattering structure, the call graph generator can also show a call graph of a nested structure, which is a big achievement of our call graph generator. The call graph generator can display all nested components. The call graph of the inner component is showed with one deeper indentation level. For example, Figure 8 and 9 show a for loop with a nested "if" clause and its call graph.

```
for i in range(0, 10):
    if apple <= 5:
        f1()
    else:
        f2()

def f1():
    print("Function f1 is called.")

def f2():
    print("Function f2 is called.")
```

Figure 8: Python Nest Structure

```
for loop:
    if True
        f1()
            print('Function f1 is called.')
    else False
        f2()
            print('Function f2 is called.')
```

Figure 9: Call Graph Output of Figure 8

5 Limitations and Threats to Validity

There are limitations to our syntax checker. It cannot check some of the software quality requirements which require context analysis. For example, our checker is unable to answer these two questions: "is there a comment that completes a logical task?" and "does the main function in the code correspond to the main algorithm?". We have decided to ignore those features for now and may work on them later.

Since static call graph generators are always optimistic, the generator must be unspecific or unsound for some cases. For our generator, we need an assumption before getting the correct result as the program would always have the main function. Since the assumption is too specific, many simple Python programs which do not contain the main function would not be able to get any results by running our generator. Even when a main function is present in the program, the generator is unable to check the class method call and recursion functions.

6 Related Work

There are some related works done in the field such as PythonBuddy [10] which is modified from Pylint, and PEP8 online check [11] which is not modified from Pylint. The result generated by PythonBuddy is similar as one would run Pylint locally, and its modifications are only related to result representation instead of analyzation. PEP8 online check only examine Python coding convention PEP8. PythonBuddy is also equipped with Python sandbox and it highlights the code with error in a more visual-appealing way.

There is an existing tool which generates python call graph named Python Call Graph [12]. Instead of generating the textual result, Python Call Graph generates the image result that shows the relationship between function calls. It generates the call graph by running the program. The execution is based on the input Python program, such that the output might be different every time it runs. It only reveals function calls between functions and records how many times a function calls another function. The disadvantage of this package is that it would be inaccurate if the execution skips some parts of the program. Our call graph generator is managed to avoid this pitfall by performing a static analysis and the result contains all possible execution paths of the program.

7 Conclusion

A new approach to generate the call graph statically using Pylint is done. Its core idea is to find out all the possible function calls

by using the Pylint built-in AST structure. Additionally, we have implemented new checkers which match the University of Alberta 100-level course software quality requirement. The checker is well documented in which it can be easily modified to make further development.

8 Future Work

Our current work has focused on simple syntax check. There are some future works to be done such as in order to check the comments efficiency and quality for our syntax checker, we need some ways to check the keywords used in the comments. As an instance, synonym checking tools or machine learning technology would be capable to perform the tasks. For the call graph generator, it would be better to get the correct call graph without having the main function. Besides, our generator could be applicable for class method calls and recursion functions in the future developments. That being said, our configurations and results are made public and it is available on GitHub to allow reproducibility.

ACKNOWLEDGMENTS

We thank for the University of Alberta instructor Sadaf Ahmed. She has provided us guidance with the software quality requirement for CMPUT174 at the University of Alberta.

REFERENCES

[1] Dmitry Filippov. 2014. Write Clean, Professional, Maintainable, Quality Code in Python. (June 2014). Retrieved April 14, 2019 from https://blog.jetbrains.com/pycharm/2014/06/write-clean-professional-maintainable-quality-code-in-python/

[2] P.T. Geach. 1972. A Program for Syntax. Semantics of Natural Language (1972), 483–497. DOI:http://dx.doi.org/10.1007/978-94-010-2557-7_17

[3] Anon. PEP 8 -- Style Guide for Python Code. Retrieved April 14, 2019 from https://www.python.org/dev/peps/pep-0008/

[4] B.g. Ryder. 1979. Constructing the Call Graph of a Program. IEEE Transactions on Software Engineering SE-5, 3 (1979), 216–226. DOI:http://dx.doi.org/10.1109/tse.1979.234183

[5] Anon. Introduction¶. Retrieved April 14, 2019 from http://pylint.pycqa.org/en/stable/intro.html

[6] Anon. Welcome to astroid's documentation!¶. Retrieved April 14, 2019 from https://astroid.readthedocs.io/en/latest/

[7] Anon. How to Write a Checker¶. Retrieved April 14, 2019 from http://pylint.pycqa.org/en/stable/how_tos/custom_checkers.html

[8] Anon. 2018. Call graph. (December 2018). Retrieved April 14, 2019 from https://en.wikipedia.org/wiki/Call_graph

[9] Anon. 2019. Test-driven development. (March 2019). Retrieved April 14, 2019 from https://en.wikipedia.org/wiki/Test-driven_development

[10] Ethan Chiu. Python Linter Online. Retrieved April 14, 2019 from https://pythonbuddy.com/

[11] Anon. PEP8 online check. Retrieved April 14, 2019 from http://pep8online.com/

[12] Anon. Python Call Graph¶. Retrieved April 14, 2019 from http://pycallgraph.slowchop.com/en/master/