OPINION

Automated Assessment of Python Code

Sam Morley, School of Mathematics, University of East Anglia, Norwich, UK. Email: Sam.Morley@uea.ac.uk. https://orcid.org/0000-0001-5971-7418.

Keywords: python, assessment, code quality.

1. Introduction

Python is a high-level programming language that is interpreted and executed by a runtime, most commonly CPython. It is fast becoming one of the most popular languages due to its flexibility and interoperability with optimised tooling and libraries. Moreover, Python is syntactically simple, which makes it ideal as a programming language for teaching and learning as a first language. The fact that Python is interpreted and executed together at execution time gives Python the ability to perform deep introspection, which also makes it an excellent language for automated assessment.

The marking criteria for code can usually be categorised as either *style* or *operation*. For beginner programmers, operation – whether the code performs the intended task correctly – is arguably more important. As experience grows, it becomes increasingly important to write well-structured and idiomatic code that is easy to read (style). More advanced programmers might also consider how robust their code is; how it handles expected or unexpected errors. For introductory programming courses, assessments should place greater emphasis on operation and less on style. However, including marking criteria for coding style will help encourage good coding practice, so it might be beneficial to include some style criteria.

Automated assessment offers various advantages over manual assessment, where an assessor would run the students' code and check the output by hand. Running a test suite over a large collection of student's code by hand can be extremely time consuming, whereas an automatic tool can run tests in the background. Automated test suites can also run in parallel, which can lead to a greatly decreased total testing time. However, automated tests are more time consuming to set up.

The author is currently, at time of writing, teaching Python programming as part of a first year undergraduate module, and is designing the assessment of Python code for this course. This assessment is intended to be marked automatically. In this article, we give a quick overview of some techniques, packages, and tools that may be useful for automatically assessing Python source code. At the end of the paper, we give a short example of a function that could be used to mark a script-based submission.

2. Assessing operation

Arguably the most important aspect of assessing the operation of source code is checking correctness. First and foremost, code should always produce predictable and repeatable results, given the same environment and input, which can be tested by a suite of individual *unit tests*. A unit test is usually a small function that tests a specific aspect of a program using assertions. The Python Standard Library contains the unittest module, that provides support for writing and running unit tests on a Python module (program). There are third party alternatives that offer a much more flexibility and features, the most popular of which is Pytest (2020).

Automated suites of tests provide a much more consistent approach to checking correctness than manual tests – that is, a user (assessor) manually running each module/item with given input and

comparing output – and is usually much faster. On the other hand, test suites can be difficult to construct properly and require a larger investment of time to write. It might also be possible to link the running of the test suite automatically upon submission via a virtual learning environment (VLE), to provide almost real-time feedback to students, which is useful for formative work. (Integration of automated testing into deployment of code is widely used in industry, via *continuous integration* services, as a fail-safe against errors that could impact the function of the program.)

There are, however, some practical concerns about using automated testing suites to test a cohort of student source code. First, the testing utilities mentioned above rely on predictable names and locations of source files. They rely on Python's standard import mechanism to bring objects into the testing suite, and this mechanism requires hard coded names of packages/modules relative to a path recognised by Python. This is difficult to accommodate if each student's code is stored in a separate directory or named according to some identifying information. (Many VLEs will automatically *mangle* the filename of submitted files to contain key information such as the submission identifier, student ID, submission time, and original file name.) Second, these tools are aimed towards testing specific elements from a large body of code, such as single function or class. It can be difficult to adapt them to test "script-like" Python files that execute immediately at the top-level.

These practical difficulties can be overcome without too much trouble. The first problem can be solved by loading the student's code in a more controllable way. For example, we could write a small script that modifies the Python path variable to cause the import statement to look for code in a different directory for each submission. The test suite can then be executed once for each submission, controlled by the script, and we can be sure that every student's submission is tested in the same way. The second problem can be mitigated by setting questions very carefully, or writing tests that execute submission files individually and check output by inspecting the stdout/stderr streams. This can cause some problems, because of the lack of standardisation for returning information from a script. (Printing to the terminal works fine in theory, but in practice string matching can be difficult, even if the form of the printed string is known; in general, this will not be the case.)

2.1. Assessing robustness and good design

A second aspect of testing operation of a source file is testing the robustness of code. For example, we might test how a student's code reacts when an error occurs (while opening a file, for example). Code that handles such errors gracefully and potentially recover, if appropriate, might be described as *robust.* A more typical example would be to test if the student has used any type and value validation in their code to raise an appropriate exception (or assertion) if an argument is provided that is not valid.

These features might not be directly related to the execution of an element in the program, but they are certainly part of good design. Since Python is not strongly typed, we cannot rely on a compiler to check that the types provided as arguments are valid and fulfil the requirements of the function. In practice, this can lead to obscure and unexpected errors that are difficult to fix. Stressing good practice and style should be part of every course on programming, regardless of level, although it will not usually be the most critical element of assessment.

Other good design elements such as breaking repeated blocks of code into individual functions can be more difficult to test automatically, and instead will probably require somebody to read the code. Perhaps some of the tracing and debugging tools within Python (or newly introduce audit hooks, provided in CPython 3.8) could be used to check the usage of functions at runtime, but it is not obvious how to implement criteria for this task.

3. Assessing style

In addition to the assessment of operation, the code can be analysed for compliance with specific style guide, such as the official Python style guide PEP8 (van Rossum, Warsaw and Coghlan, 2013). This is usually achieved using a *linter*, which inspects the code and compares the actual code as written to some idealised version. The resulting differences are reported back to the user via the terminal, or by some other means. Popular linting programs include PyLint (Logilab, n.d.) and Flake8 (Stapleton Cordasco, 2016), although there are a large number of such programs available. These tools fall into the category of *static analysis* (analysing code without running it).

Linters can detect syntax errors and typographical errors within code before the code is executed, which makes them particularly useful as part of a build and deploy pipeline in industry (such as in a continuous integration process). Some tools can also detect code that is not *Pythonic*. This means that the programmer has not made use of Python idioms in their code. For instance, it might detect when a programmer has written code for iterating over a list using a for loop over a range and accessing each item by index, rather than leveraging the iterator protocol. While technically correct, iterating over a list by index can often lead to errors if not done correctly and is frequently slower than the iterator equivalent, particularly if the element is accessed numerous times within the body of the loop. (Each lookup incurs a function invocation rather than a simple pointer lookup. This difference is miniscule but significant in large loops.)

Generally, a linter will output a list of messages regarding issues with the code checked into the terminal, and it is up to the user to parse this list and generate a score, if necessary. The messages that are output can usually be customised to include only certain issues, such as syntax errors or convention items. It might be possible to access an application programming interface (API) for a linter and link this directly into an assessment script.

4. Example

We now give a short example function that would mark scripts submitted by students, which I supposed to plot a simple function using the Matplotlib library. We omit the code that handles finding and reading student code and the handling of edge cases for the sake of space.)

```
from unittest import mock
from io import StringIO
import matplotlib.pyplot

def mark_code(sub_code):
    patcher = mock.patch("matplotlib.pyplot", autospec=True
    mpl_mock = patcher.start()
    try:
        exec(sub_code)
    except Exception as err:
        return 0, "Your code did not execute"
    patcher.stop()
    if mpl_mock.plot.called:
       return 1, "You plotted something, well done"
    return 0, "You didn't plot anything"
```

The main step involved here are *mocking* the Matplotlib Pyplot module to test whether it was used when the submission code was executed. Once this is done, we can inspect the Mock object to see which routines have been called, in this case the plot routine. We execute the submission code inside a try block using the built in exec function. This replicates the way that Python executes code from

a command line. The function returns a mark (0 or 1) and some simple feedback. In this case, the submission would get 1 mark if the plot routine was called, and 0 otherwise.

5. Conclusions

The most important part of assessing Python code written by students is testing whether the code runs and meets the requirements of the exercise. This can be achieved using tools from the software testing facilities available for Python, such as the unittest module from the Python Standard Library, or the Pytest package. However, one should not neglect the importance of good coding style when writing assessments, even if this is only used formatively. Tools such as PyLint can be used to quickly assess the style of Python code, and provide useful feedback to students on how to improve the style of their code.

6. References

Stapleton Cordasco, I., 2016. *Flake8: Your Tool For Style Guide Enforcement*. Available at: https://flake8.pycqa.org/en/latest/ [Accessed 9 July 2020].

Logilab, n.d. *Pylint*. Available at: https://pylint.org/ [Accessed 9 July 2020].

Pytest, 2020. *pytest: helps you write better programs*. Available at: https://docs.pytest.org/en/stable/ [Accessed 9 July 2020].

van Rossum, G., Warsaw, B. and Coghlan, N., 2013. *PEP 8 -- Style Guide for Python Code*. Available at: https://www.python.org/dev/peps/pep-0008/ [Accessed 9 July 2020].