A GENERIC PYTHON AUTO-GRADER

Albert Chan, Fayetteville State University, Fayetteville, NC, USA, achan@uncfsu.edu

ABSTRACT

Automated grading ("auto-grading") is an active area of research in education, particularly computer science education. Programming assignments can be difficult to grade since instructors often cannot merely look at the program to assign a grade. Rather, programs submitted by students generally need to be executed to determine whether the assignment's requirements have been satisfied. As a result, in order to grade one submission, instructors often have to set up an execution environment, run the submitted program, and tear down the environment afterwards. Auto-graders can assist the instructor in this situation.

In this paper, we present an open-source auto-tester/auto-grader that enables students to test their work before submission and facilitates the automated grading of Python assignments for instructors. Our approach to developing an auto-grader allows instructors to specify additional assignment requirements and obtain relevant statistics about the submitted code, in order to facilitate the grading process. The auto-grader is made publicly available to all instructors from a GitHub repository.

1. Introduction

Entry-level programming classes tend to be large, with a regular classroom easily containing 30 to 50 students, larger classes consisting of 200 to 500 students, and online classes accommodating thousands of students. Although instructors often have teaching assistants to help grading student assignments, this is not true in all cases, especially in teaching universities. Without teaching assistants, instructors have to grade a large number of assignments, usually within a short turnaround time in order to give timely feedback to students.

In this paper, we present an open-source, easy-to-adopt automated grader that instructors can use to auto-grade Python programming assignments. Our approach does not specify a pass-or-fail system since manual inspection of student work is unavoidable, even with the use of an auto-grader. However, our auto-grader also allows instructors to easily collect statistics about student submissions, which should make the manual inspection process less time-consuming and stressful for instructors.

The organization of this article is as follows: in Section 2, we briefly review some recent work related to auto-graders. Section 3 outlines our auto-grader and explains how to use and/or modify it to suit each instructor's individual needs. In Section 4, we describe some potential future improvements to the auto-grader. Finally, Section 5 concludes the discussion.

2. Related Work

There are many commercial auto-grading systems, including Codio, GradeScope, Mimir, Turing's Craft, Vocareum, and Zybooks. (The author has not used, and does not endorse, any of these products.) Most of these commercial products provide free access for instructors, while students have to pay for usage. Student access fees are typically not expensive, but it can still create an unnecessary burden for low-income students.

The development of auto-graders is also an active area of research. (Douce et al., 2005) presented a detailed review of early work in this field. (Nordquist, 2007) described a general auto-grader to fully automate the grading of Java assignments, and (Haldeman et al., 2018) discussed the importance of meaningful feedback in auto-grading programming assignments.

Several groups have developed specialized auto-graders tailored to specific tasks. For instance, (Carbunescu et al., 2014) presented an auto-grader for parallel programs. To grade distributed algorithms, (Maicus et al., 2019) used an auto-grader that runs in a container. (Liu et al., 2019) described an auto-grader based on formal semantics, and (Moghadam et al., 2015) described one focused on coding style. (Peveler et al., 2003) compared the approaches of using either a virtual machine or a container for auto-grading. To auto-grade untrusted code, (Malan, 2013) used a secure sandbox. (Helmick, 2017) presented a system for auto-grading interface-based Java assignments. (Johnson, 2012) described a new approach for checking the conformance of student-submitted work to the specification provided by the instructor, thereby improving the efficiency of auto-grading.

In this work, we develop and implement a novel auto-grader that is easy-to-adopt and generalizable. Compared to previously described auto-graders, our approach enables instructors to easily add additional input or information to the auto-grader, which will further relieve the burden on instructors.

3. Details

3.1. Overview

At the author's institution, Python is the entry-level programming language. Students take a two-semester Python programming course sequence to learn how to program in Python and learn about the object-oriented paradigm. Students continue by taking a data structures course, in which Python is used as a vehicle to demonstrate the implementation of data structures and algorithms. Because the Python language is heavily used in all of these entry-level courses, an auto-grader to grade Python assignments would be extremely helpful to course instructors.

Here, we assume that the students are asked to implement a single Python function. The behavior of the function is precisely defined. We also assume that this function has no visible side effects (that is, the auto-grader will ignore any visible side effects such as outputs by the print function), and the return value can be completely determined by the inputs (function arguments).

We divide the framework into five parts: test data, model solution, auto-tester, auto-grader, and test generation.

The purpose of separating the auto-tester and auto-grader is to allow students to use the auto-tester to test their work, while the functionality can be reused by instructors in the auto-grader component. This allows students to receive instant feedback on their work. Research has shown that instant or early feedback can contribute positively to students' learning by removing the elements of surprise (Haldeman et al., 2018).

The reason for separating the test data from the auto-tester is to allow easy replacement of the test data. In other words, students will use one set of data for their own testing, while the instructor can use another set of (similar) data for grading, if desired.

The test generation component enables the automatic generation of a large number of test cases.

3.2. Test Data

The test suite is encapsulated in a Python list of test cases named `tests`. Each test case is a Python tuple. This tuple contains 3 or 4 elements:

- A test ID
- Input parameters (in the form of a single value, or a tuple of single or multiple values)
- Return value (in the form of a single value, or a tuple of multiple values), or `None` (in the case that an exception is expected)
- Expected exception (optional, in the form of a single exception class), or `None` (in the case that no exception is expected). If missing, `None` is assumed.

Since the test suite is collected under a variable (`tests`), it can be easily imported and made available to the auto-tester and auto-grader.

3.3. Auto-Tester

We first implement a function to import a function from an external module. Since all students will implement the required function with the same name (and possibly also store the function in files with the same filename), Python may be confused when importing them. One reason for a potential error is that Python may choose to compile and cache a successfully imported function of the same name for future use. Another reason is that when a student's work fails to load, then the previous student's submitted work may be used without the instructor's knowledge. To minimize this risk, we encapsulate the loading of the student's implementation in a single function:

```
def loadFunction (module, location, function):
```

This function takes 3 arguments, all strings. The first is the name of the module (usually the name of the student-submitted file without the "`.py`" extension); the second is the location of the module (so it does not have to be mixed with the tester code, and ideally, it can be located in a zip file – see later for more details); and the last is the name of the function to load. In this function, we first insert the location at the beginning of the search path, so it will always be found. We then use the Python built-in `__import__` function to locate the module and then use the `getattr` function

to retrieve and return the (student-defined) function. This will raise an exception if the required function cannot be found (or cannot be loaded, potentially due to a syntax error).

We note that this framework is not absolutely required for the auto-tester. As the auto-tester is used by the student, there will be no multiple imports of the same function, and a simple static import will be sufficient. For example, the following function call:

```
fun = loadFunction ('homework', '.', 'function')
```

is roughly equivalent to:

```
from homework import function as fun
```

However, since the auto-tester functionality will be reused in the auto-grader, using the `loadFunction` function is safer.

Next, we define a function to test for a single test case:

```
def runSingleTest (function, testcase, verbose = False, record = printMessage):
```

This function takes the function to be tested as its first parameter, and a test case as its second parameter. It also accepts two additional optional parameters to control verbosity and output media. The `runSingleTest` function runs the test case through the imported student-defined function and compares the return value (and/or resulting exception, in case of failure) to the expected return value (and/or expected exceptions, if appropriate). The function returns a Boolean value to indicate the success or failure of this test case.

By default, verbosity is turned off so the tester will not generate an excessive amount of output when running a large number of test cases.

Another function is needed to run the entire test suite in one step:

```
def runAllTests (testSuite, module = 'homework', path = '.', record = printMessage):
```

This function is conceptually simple. It first loads the student-defined function. It then runs each test case sequentially and records the success or failure for each test case. Finally, it outputs the results. This function also returns the computed score that can be further processed in the auto-grader component.

As previously mentioned, verbosity is turned off by default to keep the output of the auto-tester concise. However, we found it helpful to be able to easily run a single test case and produce more verbose output. This enables students to get valuable insight on the cause the failure for the test case and improve on their implementation. We create a function named "`runTestCase`" for this purpose. We note that `runTestCase` must be run in interactive mode with the batch test successfully executed first. The following demonstrates sample uses of `runTestCase`:

```
$ python -i Tester.py
<<<<< output of batch test omitted >>>>>
>>> runTestCase (5)
TC 5 failed - expecting an exception, none raised
>>> runTestCase (10)
TC 10 passed
```

One very important requirement of the auto-tester/auto-grader is that the testing of student-submitted work should not be interrupted by any (intended or unintended) exceptions thrown by the students' work during the testing. This can be achieved by surrounding the execution with a `try` … `except` … block.

3.4. Auto-Grader

The auto-grader builds on the existing framework of the auto-tester, with some additional functionality added. The actual testing of student works is relatively and intuitively simple, since calling the tester's `runAllTests` function for each student should do the job. There are, however, some complications.

The first problem is passing each of all the student-submitted works to the grader. Most Learning Management Systems (e.g. Blackboard and Canvas) allow the instructors to download all student-submitted assignments as a zip-file (for example, Canvas will download all student-submitted work for an assignment with a default name `submissions.zip` – note that the name can be changed as part of the download process or through the operating system). The files can then be unzipped into a folder and some Python code can be used to go through each file and

send it to the tester. However, Python has built-in support to handle zip-files. This capability allows us to avoid the extra steps of unzipping the file, running the test, and finally deleting the files. More conveniently, the `__import__` and `getattr` functions mentioned in the previous section can directly extract a function defined in a file located inside a zip-file, without having to first manually extract the file. All we need to do is to pass the zip-file as the path to the desired module.

```
def gradeOneStudent (testSuite, filename, path, zip, log):
```

Here `testSuite` is the test suite of test cases as defined previously, `filename` is the name of the student file inside the zip-file, `path` is the complete path of the zip-file (for example: "`/path/to/zipfile/submissions.zip`"), `zip` is the zip-file object (opened by `ZipFile (path)` – it is needed later to collect statistics about the submitted program – see next section), and `log` is the opened log file.

`gradeOneStudent` will delegate the testing to the tester's `runAllTests` function. After it returns the score, some additional adjustments can be performed (see next section).

Another consideration in developing the auto-grader is that LMSs will usually augment the submitted filename with some additional attributes. For example, Canvas will add a student login ID, an indication of whether the submission is late or not, and some seemingly arbitrary numbers. These additional attributes should not pose any problems for the `__import__` function. However, we would like the ability to identify the student for each submitted assignment. This is not as important for the auto-tester, which is designed to be used by the student, but for the auto-grader, this information needs to be available in order to assign credit to the appropriate students. Since this identifying information is embedded in the augmented filename, it is theoretically easy to write a short function to extract this information. Unfortunately, different LMSs use different approaches for augmenting the filenames, so there is no one-size-fits-all solution to this problem. The user must first note how the LMS augments the filenames and then modify the routine accordingly.

Of course, our goal is to grade all students in a single step. To do this, we create another function `gradeAllStudents`:

```
def gradeAllStudents ():
```

This function is the entry point for the auto-grader. It opens the zip-file, then loops through it and invokes `gradeOneStudent` for each file inside.

3.5. Collecting Program Statistics

Instructors cannot solely depend on the auto-grader to grade students' work. For example, it is possible that a student may almost implement the function correctly, but a slight syntax error prevents the function from being loaded, causing the entire test suite to fail. Another situation is that the student may import a library and delegate all (or most) of the work to the library. This may not be permitted by the assignment, and it cannot be detected by the auto-grader. Consequently, manual inspection of student-submitted work is still required.

Although manual inspection is unavoidable, the ability to automatically obtain some statistics about the submissions would assist the instructor in performing such inspection. For this reason, we add to the auto-grader the capacity to collect some measurements about the submitted work. We will collect the following statistics:

0. File size in number of characters (after normalizing all newline characters to the Linux `LF` style for a fair comparison)
1. Number of words
2. Number of lines (by counting the newline characters, taking into account that a newline character may or may not appear at the end of the program)

We then go through a second phase of normalization to remove comment contents (but leaving the "`#`" character to signal a comment on a line) and string contents (but leaving the quote characters to signal a string and also normalizing all quote characters in one style – here we choose single quotes). This is required so that we will not count the "`#`" character within strings as comments; or count quoted strings within comments as strings. It will also prevent keywords appeared in comments and inside strings to be counted.

After this second phrase of normalization, we collect the following:

3. Number of blank lines
4. Number of lines with comments

5. Number of lines with the "import" keyword
6. Number of lines with the "global" keyword
7. Number of classes defined
8. Number of top-level classes defined
9. Number of functions defined
10. Number of top-level functions defined

The function to collect these statistics is:

```
def getStats (program):
```

Here `program` is a string representing the student program. We put this function (and its supporting functions) in a separate file "`Utilities.py`" so it can be reused by other projects.

The collected statistics will be returned as a tuple for easy access. The numbers in the above list also match the indices of the items in the tuple. For example, if we want to calculate the percentage of lines that have comments, the following sample code could be used:

```
stats = getStats (program)
percentage = stats [4] / (float) stats [2]
```

We also structure the code in such a way that it is easy for instructors to add their own custom statistics. The code provides examples of various ways to do this.

3.6. Test Data Generation

One advantage of using a tester and grader is that the student implementation can be tested with many test cases to cover more ground. For example, if the function is assessed with only 10 testcases, there is a relatively large probability that some edge cases may be missed. On the other hand, this will be less likely to happen if 100 or more test cases are executed. Additionally, it may be easy for students to write a solution that only passes the given test cases, if the number of test cases is small. The number of test cases that are needed depends highly on the specific problem and the input-output requirements.

However, it is always difficult and time-consuming to manually create a large number of test cases. As a result, it is helpful to have a mechanism for automatic test data generation, to make the auto-tester and auto-grader efficient.

The first requirement for such a mechanism is the ability to deduce the output from the input. For this, the instructor needs to be able to correctly implement the solution, following the instructions given to the students.

We also need to be able to generate random input data. For this task, the instructor needs to provide a function "`getRandomInput`". This function takes no parameter and returns a single value or a tuple representing the possible input parameter(s). This should include valid and invalid parameters (if appropriate). An example in the next section demonstrates this process.

There may be a situation in which some special input data needs to be tested but is not (or is hard to be) generated by the automated generator. To overcome this problem, we let the generator output to a text file (e.g. "`TestData.py`"), so that hand-crafted test cases can be easily added afterwards, if desired.

3.7. Example

Let us assume that students are asked to implement a function named `double` (note that the name of this function can actually be anything that makes sense in the context). The function takes one input parameter. If the input is a positive integer, then the function returns twice the input. If the input is an integer but not positive, then the function raises a `ValueError`. If the input is not an integer at all, then the function raises a `TypeError`. Furthermore, the students are not allowed to use any `import` statement in their implementation (violation results in a deduction of 50% of the original score). Also, students are not allowed to implement any helper functions (violation results in a deduction of 20% of the original score).

The model solution is first implemented in a file named "`ModelSolution.py`". Again, the function can be appropriately named:

```
def double (x):
    if isinstance (x, int):
        if x > 0:
```

```
            return x + x
        else:
            raise ValueError ()
    else:
        raise TypeError ()
```

We also need to implement a function named `getRandomInput` (as described in previous section) to obtain random inputs. Here, we assume that we want 75% of the generated inputs to be valid inputs (positive integers), 15% of the inputs to be non-positive integers, and 10% of the inputs to be floating-point values.

```
def getRandomInput ():
    choice = random ()
    if choice < 0.75:
        return randrange (1, 1001) # valid input
    elif choice < 0.85:
        return -randrange (1001) # invalid input
    else:
        return random () * 1000 # invalid type
```

Finally, we also need to set a few variables, including the name of the model solution function (this is not necessary if we name the solution function "`function`"), the number of test cases that should be generated, and the name of the output data file (without "`.py`" extension):

```
function = double
nTestCases = 100
dataFileName = 'TestData'
```

Now we can run the test data generator to generate the test data file. In this example, it will generate 100 testcases and store the generated data in a file named "`TestData.py`":

```
tests = [(1, -733, None, ValueError),
         (2, 361.550717122, None, TypeError),
         (3, 623, 1246, None),
         (4, 967, 1934, None),
         (5, -702, None, ValueError),
         …
         (96, -979, None, ValueError),
         (97, 966.237307706, None, TypeError),
         (98, -70, None, ValueError),
         (99, -27, None, ValueError),
         (100, 897, 1794, None)]
```

This file can now be distributed to the students (along with the tester) or used for grading.

Let's assume that a student implemented the function correctly, but failed to check for error conditions:

```
def function (x):
    return x + x
```

Running the tester on this solution gives the following output:

```
Passed: [3, 4, 6, 8, 9, 10, 11, 12, 13, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,
26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 38, 39, 40, 41, 42, 43, 45, 46, 47, 48,
52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 79, 80, 82, 84, 86, 88, 89, 91, 100] - total 75.
Failed: [1, 2, 5, 7, 14, 35, 44, 49, 50, 51, 66, 78, 81, 83, 85, 87, 90, 92, 93,
94, 95, 96, 97, 98, 99] - total 25.

75 of 100 test cases passed. Score = 75.00 of 100.00
```

The specifications of no `import` statement and no helper functions can easily be implemented in the auto-grader, after the tester has scored and returned the grade:

```
stats = getStats (readfile (zip, filename))
factor = 1.0
if (not stat [5] == 0): # imports
    factor = factor - 0.5
    # you may want to inform students here why
    # their scores are reduced and by how much
if (not stat [10] == 1): # toplevel functions
```

```
    factor = factor - 0.2
    # see comments above
score = score * factor
```

3.8. Other Options

In general, the scripting nature of Python means that we will distribute the source code of `Tester.py` and `TestData.py` to students. However, it is also possible to compile the two Python source files into bytecode files and distribute the compiled bytecode files to students (`Tester.pyc` and `TestData.pyc`). This will work exactly as if the source code files are distributed. However, note that unlike Java bytecode, Python bytecode is not universal, and it is very sensitive to the interpreter version. Therefore, this option is useful if and only if the students are required to use a unified Python environment.

3.9. Availability

The auto-grader can be accessed from the following GitHub repository: https://github.com/alchan/A-Generic-Python-Auto-Grader.

4. Future Work

There are several improvements that could be made to the auto-grader. The following list contains a few examples, ranging roughly from easier to more difficult:

1. Expanding the metrics of statistics collection. As the list of statistics is expanded, potentially becoming less readable, accessing individual values by index may become more difficult. One possible solution to this problem is to define mnemonics for constant-value variables for the indices. Another approach is to encapsulate the statistics into a class and provide access methods with mnemonics names to access each individual value.
2. Expanding the auto-tester/auto-grader to handle classes. This is more difficult for several reasons. First, a class usually has more than one method (at minimum, one should have a constructor, `__init__`, which is treated as a method in all aspects), so a mechanism is needed to determine which method to test. Second, instant methods of a class are associated with an instance of the class, and a mechanism is needed to pass an appropriately constructed object to the tester (this must be based on the assumption that the student's code can correctly construct the object in the first place, which may not be true). Third, since objects usually also encapsulate states, it is hard to guarantee that a call to a method is not affected by a previous call to the same or another method. For this reason, it is also hard to guarantee that the success or failure of a test case will not affect later test cases.
3. Limiting the runtime and/or memory usage of the function. This can prevent an ill-behaved submission from crashing the auto-grader. This feature will involve setting up a separate thread in which the student-submitted function can be executed, while the main thread will monitor the usage of resources. Once the thresholds specified by the instructor are surpassed, the main thread will terminate the spawned thread and mark the test as a failure.
4. Adopting the auto-tester/auto-grader to other programming languages. We have already found some success in adopting the auto-grader to grade Haskell assignments. First, we created an auto-tester in Haskell (this is not very difficult to do) for students to use to test their implementations. Next, we wrote a Haskell program to invoke the tester and output the result. A Python script was then used to compile and execute the Haskell program and redirect the output to a file. Finally, the auto-grader reads the output file and parses the result to determine the student's grade. The metrics collection functions can be easily modified to deal with Haskell syntax.

5. Conclusion

We have developed an open-source framework for an auto-tester/auto-grader that allows both students to test their implementations of Python functions and instructors to automatically grade Python assignments. Our approach enables instructors to easily include additional specifications for the auto-grader and retrieve relevant statistics about students' submissions, which will facilitate the grading process.

REFERENCES

Carbunescu, Devarakonda, Demmel, Gordon, Alameda, and Mehringer (2014). "Architecting an Autograder for Parallel Code," Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, XSED '14.

Douce, Livingstone, and Orwell (2005). "Automatic Test-based Assessment of Programming," Journal on Educational Resources in Computing, 5(3):4.

Haldeman, Tjang, Babeş-Vroman, Bartos, Shah, Yucht, and Nguyen (2018). "Providing Meaningful Feedback for Autograding of Programming Assignments," Proceedings of the 49th ACM Technical Symposium on computer Science Education, SIGCSE'18.

Helmick (2007). "Interface-based Programming Assignments and Automatic Grading of Java Programs," Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE'07.

Johnson (2012). “SpecCheck: Automated Generation of Tests for Interface Conformance,” Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE’12.

Liu, Wang, Wang, and Wu (2019). “Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics,” Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET’19.

Maicus, Peveler, Patterson, and Cutler (2019), “Autograding Distributed Algorithms in Networked Containers,” Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE’19.

Malan (2013), “CS50 Sandbox: Secure Execution of Untrusted Code,” Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE’13.

Moghadam, Choudhury, Yin, and Fox (2015). “AutoStyle: Toward Coding Style Feedback at Scale,” Proceedings of the Second (2015) ACM Conference on Learning @ Scale, L@S’15.

Nordquist (2007). “Providing Accurate and Timely Feedback by Automatically Grading Student Programming Labs,” Journal of Computing Sciences in colleges, 23(2).

Peveler, Maicus, and Cutler (2019). “Comparing Jailed Sandboxes vs Containers Within an Autograding System,” Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE’19.