Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code

Croft, D. & England, M.

Original citation & hyperlink:
Croft, D & England, M 2020, Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code in CEP 2020: Proceedings of the 4th Conference on Computing Education Practice 2020, 1, ACM, pp. 1-4, Computing Education Practice 2020, Durham, United Kingdom, 9/01/20.
https://dx.doi.org/10.1145/3372356.3372357

DOI 10.1145/3372356.3372357
ISBN 978-1-4503-7729-4

Publisher: Association for Computing Machinery

Computing with CodeRunner at Coventry University

Automated summative assessment of Python and C++ code

David Croft
Coventry University
Coventry, U.K.
David.Croft@coventry.ac.uk

Matthew England
Coventry University
Coventry, U.K.
Matthew.England@coventry.ac.uk

ABSTRACT

CodeRunner is a free open-source Moodle plugin for automatically marking student code. We describe our experience using CodeRunner for summative assessment in our first year undergraduate programming curriculum at Coventry University. We use it to assess both Python3 and C++14 code (CodeRunner supports other languages also). We give examples of our questions and report on how key metrics have changed following its use at Coventry.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; **Student assessment**; • **Applied computing** → *E-learning*; Learning management systems.

KEYWORDS

Programming Education; Automated Assessment; CodeRunner

ACM Reference Format:
David Croft and Matthew England. 2020. Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code. In *Computing Education Practice 2020 (CEP 2020), January 9, 2020, Durham, United Kingdom.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3372356.3372357

1 INTRODUCTION

CodeRunner is a tool developed at the University of Canterbury, New Zealand, for automatically assessing student code [5]. We use it for summative assessment of programming of first year Computer Science students at Coventry University.

The use of automated assessment for coding is of course not a new topic – see for example the survey [1]. However, it seems historically there are a wide range of independent tools with few used very widely beyond the institutions which created them. In contrast, CodeRunner was developed as a plugin for the very widely used learning management system Moodle[1]. Further, it is fully approved by Moodle which means installation is easy, both technically and in terms of administration[2]. CodeRunner has the potential to become the standard automated code assessment tool of choice, but to the best of our knowledge the only publication on CodeRunner is the 2016 introduction by the developers in ACM Inroads [5]. We hence think there is value in sharing our experiences.

2 OUR CONTEXT AND MOTIVATION

The authors design and lead the first year programming curriculum for the Computer Science (CS) degree at Coventry University. This is based around two key modules[3]:

4000CEM Programming and Algorithms First semester. Introduces basic control structures in Python 3 and ideas around different algorithms to solve the same problem.

4003CEM Object Oriented Programming Second semester. Translates from Python to C++14; then introduces the ideas of objects, classes, and inheritance.

Our intake is diverse: we do not require A-level Mathematics, prior CS study, or programming experience (although many students will have some or all of these). The 2018/19 cohort had ∼280 students.

2.1 Additional Project Modules

These modules focus on core programming ideas and theory, demonstrated by fairly traditional programming exercises and tasks (albeit delivered in a modern cloud-based environment as described below). However, in each semester students also take an additional group project module where they apply all the concepts they are learning in their degree to a real world problem via an activity-led learning approach. These projects, and their unique administration and assessment, are described in [2].

2.2 Other Recent Innovations at Coventry

The authors described in CEP 2019 how they have introduced the learning environment Codio to their modules [3]. Codio provides students with online virtual Linux boxes, which staff equip with guides. In [3] we described how the adoption of Codio was, in main, a response to a low-level of formative feedback provision and uptake (being made harder still by rapidly increasing student numbers). Our Codio units contain numerous formative tasks that feedback on student code automatically by running test scripts written by the authors. Additional benefits from Codio described in [3] include a standardised development environment on diverse student hardware, cloud storage for student code, and a detailed source of additional student data on engagement and progress.

[1] According to the following 2017 report Moodle is used by an absolute majority of HE institutions in Europe, Latin America and Oceania; and a quarter in North America: http://eliterate.us/academic-lms-market-share-view-across-four-global-regions/

[2] The approved plugin status meant our IT team team would add it to our Moodle installation without any major review.

[3] These modules ran for the first time in 2018/19. Later references to what happened in prior years refer to two very similar modules which these replaced.

CEP 2020, January 9, 2020, Durham, United Kingdom

ACM ISBN 978-1-4503-7729-4/20/01…$15.00
https://doi.org/10.1145/3372356.3372357

2.3 Assessment Prior to CodeRunner

As part of the project modules, students give group reports, presentations, and individual vivas on their programming. Modules 4000CEM and 4003CEM are then assessed using tests: each has one in the middle of the term and one at the end. It is the mid-term tests where we have made a change to employ CodeRunner.

Given the mid-semester time, tight feedback deadlines, and large quantity of students, an auto-marked mid-term was essential (especially with the human feedback committed to the projects). We use Moodle quizzes, but prior to 2018/19 these employed only the standard Moodle question types like multiple choice. Such questions can assess knowledge of programming concepts and theory, but not the ability to formulate an algorithm and code to solve a problem. 4000CEM has also a written exam which does assess these abilities, however, this is an inauthentic environment; different to both the one in which students learnt the material and any professional environment in which they would later apply it. E.g. no syntax checker, no access to language documentation, no ability to edit the code without writing it all out again. The additional verbosity required for C++ code meant that 4003CEM has no written exam. Our positive results with automated formative feedback [3] led us to consider the same approach for summative assessment.

2.4 Codio for Summative Assessment?

The software that administers Codio tasks is stored on the virtual machine to which students have sudo rights. Hence they could access and edit the task code if they knew how. There is no motivation for this with formative assessment, but it would be an unacceptable risk for summative. Further, our university regulations forbid the use of a third-party in summative assessment, which also precluded the use of any alternative providers with more *locked down* environments. We thus looked for another method of automated code testing for summative assessment and found CodeRunner.

3 CODERUNNER

CodeRunner is a free tool for testing student code[4]. It is an approved plug-in for the widely used learning management system Moodle and is also available open source for those who wish to customise[5].

We use CodeRunner with Python3 and C++14, but the standard installation also has question types for C, Java, Javascript (NodeJS) PhP, Matlab, and Octave; and the system essentially allows for use with any language that can be executed on the Linux server.

3.1 Main Functionality

CodeRunner provides an additional Moodle quiz question type[6] where the student answer is code, sent to a separate (university) server where it is executed against unit tests. The tests can simply check for desired standard output, or perform something more involved which the teacher codes up. Some of our favourite features:

- Students type code into a box with basic IDE functionality, e.g. syntax highlighting and automated indentation.
- Students can see the results of a syntax checker on the code before submitting it to the full set of unit tests.
- If student code creates error messages they are shown.
- After seeing the results of the unit tests students can edit their code and try again. The number of attempts, and penalty regime (if any) is fully customisable.
- Students can be given links to resources such as the standard documentation for a language.

Thus a CodeRunner test an *authentic* examination of programming ability, and we can still maintain test integrity. Students take tests under exam conditions with the usual Moodle quiz security tools: passwords, required IP, use of the Safe Exam Browser, etc.

3.2 Options for Individual Unit Tests

The number of marks for each unit test can be varied. Individual tests can also be flagged as one of the following special cases:

- **Examples:** meaning the expected input and output is displayed prior to the student attempt. These can aid student's understanding of what is being asked of them.
- **Pre-check:** meaning these tests are run for free (i.e. without penalty). We use these to check the student has spelt the function name correctly, and to inform them of syntax errors.
- **Hidden:** meaning students do not see the input and output even after the tests. Essential to avoid a student hard-coding a large if-statement with all the known test inputs!

3.3 Other Useful Features

There is an option to pre-populate the answer box with partially complete / buggy code which you can then ask the student to extend / fix. This is particularly useful for assessing object oriented programming: e.g. asking students to implement an additional method which interacts with the other methods and attributes in specified ways. Students can reset the answer box to its original state.

A teacher can opt to store their own model answer to a question. This can then be revealed to students after the test. It can also be evaluated against the unit tests after each change – a very useful error check during the development of quizzes and their tests.

4 EXAMPLES

4.1 Python

Figure 1 shows a typical question from the assessment of our Python module. Students must write a function to calculate the average string length for a given list of strings by iterating with a for loop. The expected correct answer would be similar to the one shown passing all tests in Figure 2. Our tests mark this question out of 10 marks as follows (in the order of Figure 2):

- **0.5** for defining a function with the correct name. Checked with the Python code `"avgWordLength" in globals()`
- **0.5** for the function have the correct number of parameters. Checked using the `signature` function of the `inspect` module of the Python Standard Library.
- **0.5** for the function returning data of the correct type (float).
- **0.5 × 3** for three test cases for which the student can observe the input and output. The first of these is flagged as an *Example* and so was included with the question in Figure 1.
- **2.0 × 2** for two further test cases which are flagged as hidden (they appear opaque when a teacher views as in Figure 2).

[4] http://coderunner.org.nz
[5] https://github.com/trampgeek/moodle-qtype_coderunner
[6] Thus CodeRunner questions can be combined in a test with those of other types.

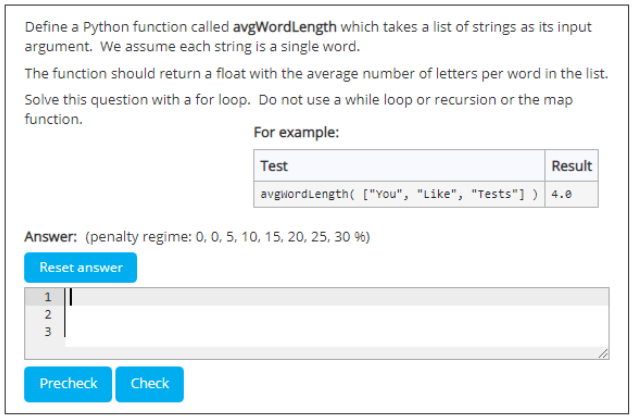

Figure 1: Example of a CodeRunner question

1.0 × 3 for checks that the requested control structure was used. The student code is stored as a string so we just check for substrings `" for "`, `" while "` & `" map "`[7] and check recursion by counting occurrences of the function name.

An initial attempt at a solution to the problem may be the following:

```
def avgWordLength(listOfWords):
    for word in listOfWords
        totalLetters = totalLetters + len(word)
    average = totalLetters/len(listOfWords)
```

This code contains 3 errors: (1) missing colon at the end of line 2; (2) no initialisation of the variable `totalLetters`; (3) not returning the final answer. (1) is a syntax error: the Precheck button would display the Python interpreter message pointing to the end of line 2. Once we add the colon the PreCheck would return no further information – students should then use Check to get the test results in Figure 3. The testing stops after one triggers a runtime error with students shown the usual message. Initialising `totalLetters` and checking again gives the results in Figure 4. This time there are no runtime errors so all unit tests are evaluated. Students do not see the opaque hidden test details but are informed of their failure.

If students then added in the return statement and pressed Check again they would pass all tests as in Figure 2. They would have only used Check twice and thus receive no penalty, getting full marks for the question. This reflects our view that errors (1)-(3) are not critical gaps in student knowledge, *so long as they are able to read the error messages and unit test feedback to correct them quickly.*

Gaming the system? A function that simply returned 4.0 (the example result) would get 20%, while one with an if-statement for the three visible test answers with any float otherwise could obtain 30%. Similarly, students could randomly guess the right answer on a multiple choice question. We could follow that if-statement with a syntactically correct but meaningless for-loop to pass all but the hidden tests and score 60%. However, in that case the student has actually demonstrated knowledge of Python for-loops: it is unlikely they would posses this knowledge but still need to game the system.

[7] The spaces around the words are deliberate as we care only for these Python keywords and their appearance within longer words. Python's required indentation for function body means they cannot occur at the start of a line.

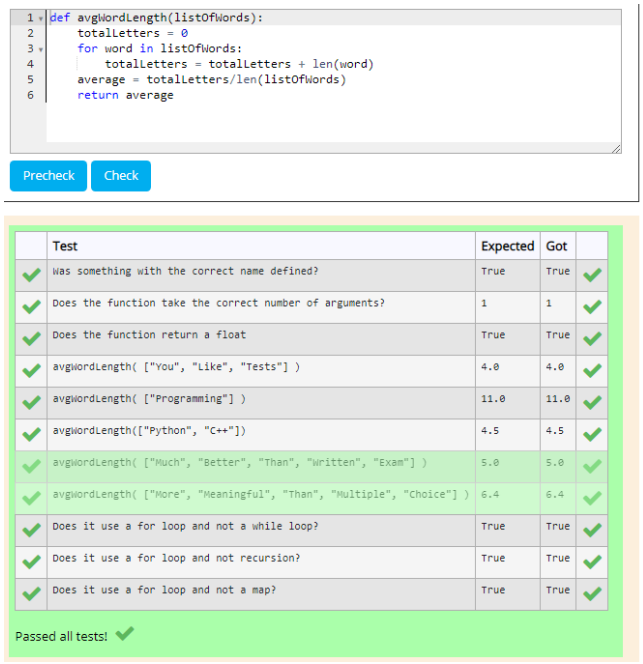

Figure 2: Example of a student answer to the question in Figure 1 that would pass all of tests (the opaque rows are hidden tests which would not be seen by the student)

4.2 C++

Utilising CodeRunner in the assessment of C++ code poses additional challenges when compared to Python. It is likely that these issues will also manifest in the assessment of other statically typed and compiled languages (e.g. Java). Essentially, a range of student mistakes which would cause runtime errors in Python cause compiler errors in C++. Then, as compilation is unsuccessful, it is impossible to run the unit tests and so students receive no marks for what may have been correct code with just minor errors.

In the code below the student was meant to declare a function that doubled its argument but misunderstood and wrote a function to simple return 10 (probably this was the example case).

```
/** correct solution **/
int double_it( int arg ) { return arg*2; }
/** student submission **/
int double_it() { return 10; }
/** unit testing **/
std::cout << double_it( 5 ) << std::endl;
```

Of course, the student does not deserve many marks but they have shown they can create a C++ function and so if marked by hand it would get some partial credit. However, a unit test like that shown on the final line would cause a compilation error. It is a trivial example but the issue applies generally: e.g. if a student declared a class with correct inheritance properties then awarding zero on the basis on a minor error in one method is extremely undesirable.

To address this we utilise a combination of Substitution Failure Is Not An Error (SFINAE) [6, § 8.3.1] and preprocessor macros to perform code introspection during compilation. A simplified

	Test	Expected	Got	
✓	Was something with the correct name defined?	True	True	✓
✓	Does the function take the correct number of arguments?	1	1	✓
✗	Does the function return a float	True	***Error*** Traceback (most recent call last): File "__tester__.python3", line 15, in <module> print(float==type(avgWordLength(["Hello"]))) File "__tester__.python3", line 3, in avgWordLength totalLetters = totalLetters + len(word) UnboundLocalError: local variable 'totalLetters' referenced before assignment	✗

Figure 3: Test results for a student answer that caused a runtime error

	Test	Expected	Got	
✓	Was something with the correct name defined?	True	True	✓
✓	Does the function take the correct number of arguments?	1	1	✓
✗	Does the function return a float	True	False	✗
✗	avgWordLength(["You", "Like", "Tests"])	4.0	None	✗
✗	avgWordLength(["Programming"])	11.0	None	✗
✗	avgWordLength(["Python", "C++"])	4.5	None	✗
✗	avgWordLength(["Much", "Better", "Than", "Written", "Exam"])	5.0	None	✗
✗	avgWordLength(["More", "Meaningful", "Than", "Multiple", "Choice"])	6.4	None	✗
✓	Does it use a for loop and not a while loop?	True	True	✓
✓	Does it use a for loop and not recursion?	True	True	✓
✓	Does it use a for loop and not a map?	True	True	✓

Some hidden test cases failed, too.

Figure 4: Failed unit tests without raising errors

demonstration of SFINAE is shown in the code below where we define a series of increasingly more specific templates to test the existence of the required function with a signature increasingly closer to the correct one. This approach allows for partial credit, and also more intuitive error messages than a C++ complier.

```
template<typename T>
std::string func_test( T *f, int arg )
{  return "Not a function";  }
template<typename R, typename A>
std::string test_function( R (*f)(A), int arg )
{  return "Incorrect parameter type";  }
template<typename R, typename ...Args>
std::string func_test( R (*f)(Args...), int arg )
{  return "Incorrect number of parameters";  }
int func_test( int (*f)(int), int arg )
{  return (*f)( arg );  };
std::cout << func_test( &double_it, 5 ) << std::endl;
```

We have developed C++ introspection code to determine: the existence of functions, parameter number/types, return types, class inheritance/attributes/methods, and more. It was a significant time investment but mostly a one-off payment[8]. Student code may still cause error messages: but these errors are from within the student's code (rather than the test code) and so the error message should be enough to correct them.

[8]Contact the first author for access to the C++ Library of introspection code.

5 RESULTS AT COVENTRY

In both modules our CodeRunner assessments had lower average marks and pass rates that the multiple choice tests they replaced. In our opinion this is because the multiple choice tests overrated student's programming ability. The final written exam for 4000CEM has been of a similar format and difficulty for several years - the cohort who were assessed in the mid term using CodeRunner actually had average exam mark 7% higher and pass rate 10% higher!

We measure student satisfaction with an online questionnaire that mimics the UK National Student Survey. In 4000CEM overall satisfaction increased 4% to 93% following the introduction of CodeRunner for the mid term assessment. More importantly, students satisfaction with assessment and feedback increased 7% to 90%. Assessment and feedback is often the area where students are most dissatisfied so this result is particularly promising. More generally, the literature suggests that the *relative correctness* of work implies by unit tests is accepted by the student body [4].

In 4003CEM the fall in initial student marks was more pronounced and there was a corresponding fall in student satisfaction. This prompted the development of the introspection code to better reward partial credit described above[9]. With this in place we anticipate a smoother experience assessing C++ next year. One topic for our future work with CodeRunner is an investigation on the most appropriate penalty regime to apply for checking the unit tests.

ACKNOWLEDGMENTS

We are grateful to the developers of CodeRunner; and the organisers of the 2017 CodeRunner workshop at the University of Edinburgh.

REFERENCES

[1] K.M. Ala-Mutka. 2005. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education* 15, 2 (2005), 83–102. https://doi.org/10.1080/08993400500150747
[2] S. Billings and M. England. 2020. First Year Computer Science Projects at Coventry University: Activity-led integrative team projects with continuous assessment. In *Proc. CEP '20*. ACM, In Press. https://doi.org/10.1145/3372356.3372358
[3] D. Croft and M. England. 2019. Computing with Codio at Coventry University: Online Virtual Linux Boxes and Automated Formative Feedback. In *Proc. CEP '19*. ACM, Article 16, 4 pages. https://doi.org/10.1145/3294016.3294018
[4] Y.B.D. Kolikant. 2005. Students' Alternative Standards for Correctness. In *Proc. ICER '05*. ACM, 37–43. http://doi.org/10.1145/1089786.1089790
[5] R. Lobb and J. Harlow. 2016. CodeRunner: A Tool for Assessing Computer Programming Skills. *ACM Inroads* 7, 1 (2016), 47–51. https://doi.org/10.1145/2810041
[6] D. Vandevoorde and N.M. Josuttis. 2002. *C++ Templates: The Complete Guide*. Addison-Wesley Professional.

[9]Approved and applied retrospectively by the exam board for 2018/19.