Automatic feedback provision in teaching computational science

Hans Fangohr[1,2], Neil O'Brien[2,3], Ondrej Hovorka[2],
Thomas Kluyver[1], Nick Hale[2], Anil Prabhakar[4], and Arti Kashyap[5]

[1] European XFEL GmbH, Schenefeld, Germany
{hans.fangohr,thomas.kluyver}@xfel.eu
[2] University of Southampton, United Kingdom
{fangohr,nsob1c12,o.hovorka,n.w.hale}@soton.ac.uk
[3] Imaging Physics, University Hospital Southampton, United Kingdom
[4] Dept. of Elec. Eng, IIT Madras, Chennai, India
anilpr@ee.iitm.ac.in
[5] School of Basic Sciences, IIT Mandi, HP, 175001, India
arti@iitmandi.ac.in

Keywords: Automatic assessment tools, automatic feedback provision, programming education, Python, self-assessment technology, pytest, computational science, data science

Abstract. We describe a method of automatic feedback provision for students learning computational science and data science methods in Python. We have implemented, used and refined this system since 2009 for growing student numbers, and summarise the design and experience of using it. The core idea is to use a unit testing framework: the teacher creates a set of unit tests, and the student code is tested by running these tests. With our implementation, students typically submit work for assessment, and receive feedback by email within a few minutes after submission. The choice of tests and the reporting back to the student is chosen to optimise the educational value for the students. The system very significantly reduces the staff time required to establish whether a student's solution is correct, and shifts the emphasis of computing laboratory student contact time from assessing correctness to providing guidance. The self-paced nature of the automatic feedback provision supports a student-centred learning approach. Students can re-submit their work repeatedly and iteratively improve their solution, and enjoy using the system. We include an evaluation of the system from using it in a class of 425 students.

1 Introduction

One of the underpinning skills for computer science, software engineering and computational science is programming. A thorough treatment of the existing literature on teaching introductory programming was given by Pears *et al.* [1], while a previous review focused mainly on novice programming and topics related

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

to novice teaching and learning [2]. We provide a recent literature review in the technical report that accompanies this work (Section 1.3 in [3]).

Programming is a creative task: given the constraints of the programming language to be used, it is the choice of the programmer what data structure to employ, what control flow to implement, what programming paradigm to use, how to name variables and functions, how to document the code, and how to structure the code that solves the problem into smaller units (which potentially could be re-used). Experienced programmers value this freedom and gain satisfaction from developing a 'beautiful' piece of code or finding an 'elegant' solution. For beginners (and teachers) the variety of 'correct' solutions can be a challenge.

Given a particular problem (or student exercise), for example to compute the solution of an ordinary differential equation, there are a number of criteria that can be used to assess the computer program that solves the problem:

1. correctness: does the code produce the correct answer? (For numerical problems, this requires some care: for the example of the differential equation, we would expect for a well-behaved differential equation that the numerical solution converges towards the exact solution as the step-width is reduced towards zero.)
2. execution time performance: how fast is the solution computed?
3. memory usage: how much RAM is required to compute the solution?
4. robustness: how robust is the implementation with respect to missing/incorrect input values, etc?
5. elegance, readability, documentation: how long is the code? Is it easy for others to understand? Is it easy to extend? Is it well documented, or is the choice of algorithm, data structures and naming of objects sufficient to document what it does?

When teaching and providing feedback, in particular to beginners, one tends to focus on correctness of the solution. However, the other criteria 2 to 5 are also important. In particular for computational science where requirements can change rapidly and users (and readers) of code may not be fully trained computer scientists [4], the readability and documentation matter.

We demonstrate in this paper that the assessment of criteria 1 to 4 can be automated in day-to-day teaching of large groups of students. While the higher-level aspects such as elegance, readability and documentation of item 5 do require manual inspection of the code for useful feedback, we argue that the teaching of these high level aspects benefits significantly from automatic feedback as all the contact time with experienced staff can be dedicated to those points, and no time is required to manually check the criteria 1 to 4.

In this work, we describe the motivation, design, implementation and effectiveness of an automatic feedback system for Python-based exercises for computational science and data science, used in teaching undergraduate students in engineering [5] and physics, and postgraduate students from a wider range of disciplines.

We aim to address the shortcomings of the current literature as outlined in the review [6] by detailing our implementation and security model, as well as

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

providing sample testing scripts, inputs and outputs, and usage data from the deployed system. Some of that material is made available as a technical report [3] and we make repeated reference to it in this manuscript.

In Section 2, we provide some historic context of how programming was taught at the University of Southampton prior the introduction of the automatic testing system described here. Section 3 introduces the new method of feedback provision, initially from a the perspective of a student, then providing more detail on design and implementation. Based on our use of the system over multiple years, we have composed results, statistics and a discussion of the system in Section 4, before we close with a summary in Section 5.

2 Traditional Delivery of Programming Education

Up to the year 2009, we taught programming and the basics of computational science in a mixture of weekly lectures that alternate with practical sessions in which the students write programs in a self-paced fashion. These programs were then reviewed and assessed individually by a demonstrator as part of the laboratory session. We estimate that 90% of the demonstrators' time went into establishing the correctness (and thus obtaining a fair assessment) of the work. Only the remaining time could be used to support students in solving the self-paced exercises and to provide feedback relating to items 2 to 5 in Section 1. We provide a more detailed description and discussion of the learning and teaching methods in [3, Section 2].

3 New Method of Automatic Feedback Provision

3.1 Overview

In 2009, we introduced an *automatic feedback provision system* that checks each student's code for correctness and provides feedback to the student within a couple of minutes of having completed the work. This takes a huge load off the demonstrators who consequently can spend most of their time helping students to do the exercises and providing additional feedback on completed and assessed solutions. Due to the introduction of the system the learning process can be supported considerably more effectively, and we could reduce the number of demonstrators from 1 per 10 students as we had pre-2009, to 1 demonstrator per 20 to 30 students, and still improve the learning experience and depth of material covered.

3.2 Student's Perspective

Once a student completes a programming exercise in the computing laboratory session, they send an email to a dedicated email account that has been created for the teaching course, and attach the file containing the code they have written. The subject line is used by the student to identify the exercise; for example "Lab

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

4" would identify the 4$^{\text{th}}$ practical session. The system receives the student's email, and tests the student's code. The student receives an email containing their assessment results and feedback. Typically, the student will receive feedback in their inbox within two to three minutes of sending their email.

Please define the following functions in the file `training1.py` and make sure they behave as expected. You also should document them suitably.

1. A function `distance(a, b)` that returns the distance between numbers `a` and `b`.
2. A function `geometric_mean(a, b)` that returns the geometric mean of two numbers, *i.e.* the edge length that a square would have so that its area equals that of a rectangle with sides `a` and `b`.
3. A function `pyramid_volume(A, h)` that computes and returns the volume of a pyramid with base area `A` and height `h`.

Fig. 1: Example exercise instructions. We focus here on question 3.

For our discussion, we use the example exercise shown in Fig. 1, which is typical of one that we might use in an introductory laboratory that introduces Python to beginners. We show a correct solution to question 3 of this example exercise in Listing 1.1. If a student submits this, along with correct responses to the other questions, by email to the system, they will receive feedback as shown in Listing 1.2.

```
def pyramid_volume(A, h):
    """Calculate and return the volume of a pyramid
    with base area A and height h."""
    return (1./3.) * A * h
```

Listing 1.1: A correct solution to question 3 of the example exercise

```
Overview
========

distance       : passed -> 100%; (weight=1)
geometric_mean : passed -> 100%; (weight=1)
pyramid_volume : passed -> 100%; (weight=1)

Total mark for this assignment: 3 / 3 = 100%.
(Points computed as 1 + 1 + 1 = 3)

-----------------------------------------------

This message has been generated automatically.  Should you feel that you
observe a malfunction of the system, or if you wish to speak to a human,
please contact the course team (course-help@uni.email.address).
```

Listing 1.2: email response to correct submission, additional line wrapping due to column width

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

```
def pyramid_volume(A, h):
    """Calculate and return the volume of a pyramid
    with base area A and height h. """
    return (A * h) / 3
```

Listing 1.3: An incorrect solution to question 3 of the example exercise, using integer division

```
Overview
========

distance       : passed -> 100%; (weight=1)
geometric_mean : passed -> 100%; (weight=1)
pyramid_volume : failed ->   0%; (weight=1)

Total mark for this assignment: 2 / 3 = 67%.
(Points computed as 1 + 1 + 0 = 2)

Test failure report
====================

test_pyramid_volume
-------------------
def test_pyramid_volume():

    # if height h is zero, expect volume zero
    assert s.pyramid_volume(1.0, 0.0) == 0.

    # tolerance for floating point answers
    eps = 1e-14

    # if we have base area A=1, height h=1, we expect a volume of 1/3.:
    assert abs(s.pyramid_volume(1., 1.) - 1./3.) < eps

    # another example
    h = 2.
    A = 4.
    assert abs(s.pyramid_volume(A, h) - correct_pyramid_volume(A, h)) < eps

    # does this also work if arguments are integers?
>   assert abs(s.pyramid_volume(1, 1) - 1./3.) < eps
E   assert 0.3333333333333333 < 1e-14
E    + where 0.3333333333333333 = abs((0 - (1.0/3.0)))
E    +   where 0 = <function pyramid_volume at 0x7f0ce1af4e60>(1, 1)
E    +     where <function pyramid_volume at 0x7f0ce1af4e60> = s.
     pyramid_volume
```

Listing 1.4: email response to incorrect solution

If the student submits an incorrect solution, for example with a mistake in question 3 as shown in Listing 1.3, they will instead receive the feedback shown in Listing 1.4. The submission in Listing 1.3 is incorrect because integer division is used rather than the required floating-point division. These exercises were based on Python 2, where the "/" operator represents integer division if both operands are of integer type, as is common in many programming languages. (We have since upgraded the curriculum and student exercises to Python 3.)

Within the testing feedback in Listing 1.4, the student code is visible in the name space `s`, i.e. the function `s.pyramid_volume` is the function defined in Listing 1.3. The function `correct_pyramid_volume` is visible to the testing system but students cannot see the implementation in the feedback they receive

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

– this allows us to define tests that compute complicated values for comparison with those computed by the student's submission, without revealing the implementation of the reference computation to the students.

3.3 Design and Implementation of Student Code Testing

The incoming student code submissions are tested through carefully designed unit tests, as discussed below. The details of the technical design and implementation are available in [3, Section 3.3]. We discuss three aspects here.

Iterative testing of student code. We have split each exercise on our courses into multiple questions, and arranged to test each question separately. Within a question, the testing process stops if any of the test criteria are not satisfied. This approach was picked to encourage an iterative process whereby students are guided to focus on one mistake at a time, correct it, and get further feedback, which improves the learning experience. This approach is similar to that taken by Tillmann *et al.* [7], where the iterative process of supplying code that works towards the behaviour of a model solution for a given exercise is so close to gaming that it "is viewed by users as a game, with a byproduct of learning". Our process familiarises the students with aspects of test-driven development [8] in a practical way.

Defining the tests. Writing the tests is key to making this automatic testing system an educational success: we build on our experience before and after the introduction of the testing system, ongoing feedback from interacting with the students, and reviewing their submissions, to design the best possible unit testing for the learning experience. This includes testing for correctness but also structuring tests in a didactically meaningful order. Comments added in the testing code will be visible to the students when a test fails, and can be used to provide guidance to the learners as to what is tested for, and what the cause of any failure may be (if desired). A more detailed discussion including the test code for the example shown in Listing 1.2 and 1.4 is given in [3, Section 3.4.3].

Results and feedback provision to students. Students receive a reply email from the testing system that provides a per-question mark, with a total mark for the submission, and then details on any errors that were encountered. In the calculation of the mark for the assessment, questions can be given different weights to reflect greater importance or challenges of particular questions. For the example shown in Listing 1.2 all questions have the same weight of 1.

We describe and illustrate a typical question, which might form part of an assignment, in Section 3.2. As shown in Listing 1.4 on page 5, when an error is encountered, the feedback that is sent to the student include the testing code up to the point of the failing assertion. The line that raises the exception is indicated with the `>` character (in this case the 6^{th}-last line shown) as is usual for pytest output [9]. This is followed by a backtrace which illustrates that, in this case, the

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

submitted `pyramid_volume` function returned `0` when it was expected to return an answer of $\frac{1}{3} \pm 1 \times 10^{-14}$.

All the tests above the failing line have passed, i.e. the functionality in the student code that is tested by these tests appears to be correct. The report also includes several comments, which are introduced in the testing code (shown in Listing 1.4), and assist students in working out what was being tested when the error was found. For example, the comment "does this also work if arguments are integers?" shows the learner that we are about to test their work with integer parameters; that should prompt them to check for integer division operations. If they do not succeed in doing this, they are able to show their feedback to a demonstrator or academic, who can can use the feedback to locate the error in the student's code swiftly, then help the student find the problem, and discuss ways to solve the issue.

In addition, students receive a weekly summary of all their past submissions and marks. The course lecturer has access to all data in a variety of ways.

Clean code. Our system optionally supports coding style analysis and in the majority of our tests for Python submissions, we perform an automated style check against the PEP8 coding standard [10]. We typically add $2^{-N_{\mathrm{err}}}$ (where N_{err} is the number of stylistic errors detected) to the student's overall mark so that full compliance with the guideline is rewarded most generously.

4 Results

4.1 Testing System Deployment

The automatic testing system was first used at the University of Southampton's Highfield Campus in the academic year 2009/2010 for teaching about 85 Aerospace engineers, and has been used every year since for growing student numbers, reaching 425 students in 2014/2015. The Southampton deployment now additionally serves another cohort of students who study at the University of Southampton Malaysia Campus (USMC) and there is a further deployment at the Indian Institute of Technology (IIT) Mandi and Madras campuses, where the system has been integrated with the Moodle learning management system [3, Section 4.5].

The testing system has been used in a number of smaller courses at Southampton, typically of approximately 20 students, such as one-week intensive Python programming courses offered to PhD students. The feedback system system has also been used in the Physics department at the University of Hamburg (Germany) to support an introduction into computational science and data science in 2019. It also serves Southampton's courses in advanced computational methods where around 100 students have submitted assignments in C (this requires an extension of the system which is beyond the scope of this contribution).

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

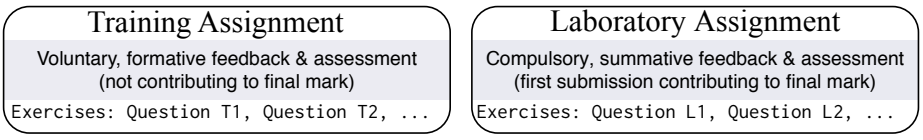


Fig. 2: Overview of the structure of the weekly computer laboratory session: A voluntary set of training exercises is offered to the students as a "training" assignment, followed by a compulsory set of exercises in the same topic area as the "laboratory" assignment which contributes to each student's final mark for the course. Automatic feedback and assessment is provided for both assignments and repeat submissions are invited.

4.2 Case Study: Introduction to Computing

In this section, we present and discuss experience and pertinent statistics from the production usage of the system in teaching our first-year computing course at the University of Southampton. In 2014/15, there were about 425 students in their first semester of studying Acoustic Engineering, Aerospace Engineering, Mechanical Engineering, and Ship Science.

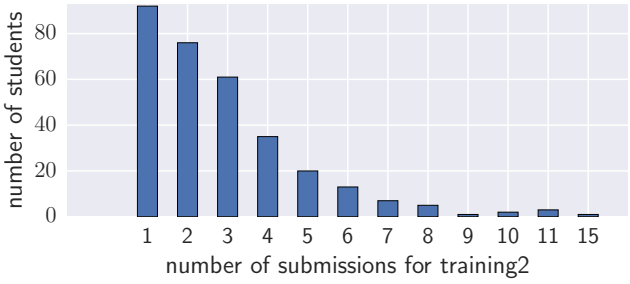
Course Structure. The course is delivered through weekly lectures and weekly self-paced student exercises with a completion deadline a day before the next lecture takes place (to allow the lecturer to sight submissions and provide generic feedback in the lecture the next day). Students are offered a 90 minute slot (which is called "computing laboratory" in Southampton) in which they can carry out the exercises, and teaching staff are available to provide help. Students are allowed and able to start the exercise before that laboratory session, and use the submission and testing system anytime before, during and after that 90 minute slot.

Each weekly exercise is split into two assignments: a set of "training" exercises and a set of assessed "laboratory" exercises. This is summarised in Fig. 2.

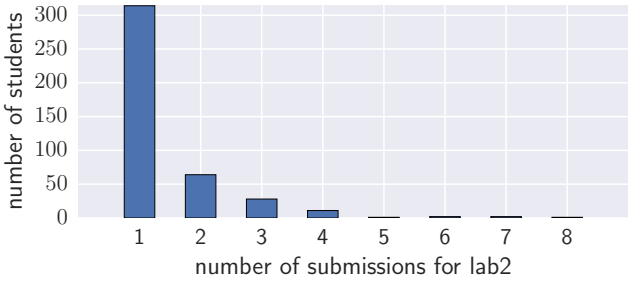
The training assignment is checked for correctness and marked using the automatic system, but whilst we record the results and feed back to the students, they do not influence the students' grades for the course. Training exercises are voluntary but the students are encouraged to complete them. Students can repeatedly re-submit their (modified) code for example until they have removed all errors from the code. Or they may wish to submit different implementations to get feedback on those.

The assessed laboratory assignment is the second part of each week's exercises. For these, the students attempt to develop a solution as perfect as possible before submitting this by email to the testing system. This "laboratory" submission is assessed, and marks and feedback are provided to the student. These marks are recorded as the student's mark for that week's exercises, and contribute to the final course mark. The student is allowed (and encouraged) to submit further solutions, which will be assessed and feedback provided, but it is the first submission that is recorded as the student's mark for that laboratory.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45



(a) training 2



(b) lab 2

Fig. 3: Histogram illustrating the distribution of submission counts per student for the (a) voluntary training and (b) assessed laboratory assignment (see text in Section 4.2)

The main assessment of the course is done through a programming exam at the end of the semester in which students write code on a computer in a 90 minute session, without Internet access but having an editor and Python interpreter to execute and debug the code they write. Each weekly assignment contributes of the order of one percent to the final mark, i.e. 10% overall for a 10 week course. Each laboratory session can be seen as a training opportunity for the exam as the format and expectations are similar.

Student behaviour: exploiting learning opportunities from multiple submissions. In Figure 3a, we illustrate the distribution of submission counts for "training 2", which is the voluntary set of exercises from week 2 of the course. The bar labelled 1 with height 92 shows that 92 students have submitted the training assignment exactly once, the bar labelled 2 shows that 76 students submitted their training assignment exactly twice, and so on. The sum over all bars is 316 and shows the total number of students participating in this voluntary training assignment. 87 students submitted four or more times, and several students submitted 10 or more times. This illustrates that our concept of students being free to make step-wise improvements where needed and rapidly get further feedback has been successfully realised.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

We can contrast this to Figure 3b, which shows the same data for the compulsory laboratory assignment in week 2 ("lab2"). This submission attracts marks which contribute to the students' overall grades for the course. In this case the students are advised that while they are free to submit multiple times for further feedback, only the mark recorded for their first submission will count towards their score for the course. For lab 2, 423 students submitted work, of whom 314 submitted once only. However, 64 students submitted a second revised version and a significant minority of 45 students submitted three or more times to avail themselves of the benefits of further feedback after revising their submissions, even though the subsequent submissions do not affect their mark.

Significant numbers of students choose to submit their work for both voluntary and compulsory assignments repeatedly, demonstrating that the system offers the students an extended learning opportunity that the conventional cycle of submitting work once, having it marked once by a human, and moving to the next exercise, does not provide.

The proportion of students submitting multiple times for the assessed laboratory assignment (Figure 3b) is smaller than for the training exercise (Figure 3a) and likely to highlight the difference between the students' approaches to formative and summative assessment. It is also possible that students need more iterations to learn new concepts in the training assignment before applying the new knowledge in the laboratory assignment, contributing to the difference in resubmissions. The larger number of students submitting for the assessed assignment ($423 \approx 100\%$) over the number of students submitting for the training assignment ($316 \approx 74\%$) shows that the incentive of having a mark contribute to their overall grade is a powerful one.

Submission behaviour changes over time. During the 10 practical sessions of this course, the number of assessed submissions decays slightly, and the participation in voluntary submissions decays more dramatically, before it increases slightly as the exam is approached. This is discussed in detail in [3, Section 4.2.3 and figures 4 and 5].

4.3 Feedback from Students

We invited feedback from the learners explicitly on the automatic feedback system asking for voluntary provision of (i) reasons why students liked the system and (ii) reasons why students disliked the system. The replies are not homogeneous enough to compile statistical summaries, but we provide a representative selection of comments in [3, Section 4.3] and discuss the main observations here.

The most frequent positive feedback from students is on the immediate feedback that the system provides. Some student comments mention explicitly the usefulness of the system's feedback which allows to identify the errors they have made more easily. In addition to these generic endorsements, some students mention explicitly advantages of the test-driven development such as re-assurance regarding correctness of code, quick feedback on refactoring, the indirect introduction of unit tests through the system, and help in writing clean code. Further

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

student feedback welcomes the ability to re-submit code repeatedly, and the flexibility to do so at any time. One student mentions the objectiveness of the system – presumably this is based on experience with assessment systems where a set of markers manually assess submissions and naturally display some variety in the application of marking guidelines.

The most common negative feedback from the students is that the automatic testing system is hard to understand. This refers to test-failure reports such as shown in Listing 1.4. Indeed, the learning curve at the beginning of the course is quite high: the first 90 minute lecture introduces Python, Hello World and functions, and demonstrates feedback from the testing system to prepare students for their self-paced exercises and the automatic feedback they will receive. However, a systematic explanation of the `assert` statements, `True` and `False` values, and exceptions, takes only place after the students have used the testing system repeatedly. The reading of error messages is of course a key skill (and the importance of this is often underestimated by our non-computer science students), and we think that the early exposure to error messages from the automatic testing is useful. In practice, most students use the hands-on computing laboratory sessions to learn and understand the error messages with the help of teaching staff before these are covered in greater detail in the lectures (see also Section 4.4).

We add our subjective observation from teaching the course that many students seem to regard the process of making their code pass the automatic tests as a challenge or game. The students play this game "against" the testing system, and they experience great satisfaction when they pass all the tests – be it in the first or a repeat submission. As students enjoy this game, they very much look forward to being able to start the next set of exercises which is a great motivation to actively follow and participate in all the teaching activities.

4.4 Discussion

Key benefits of automatic testing. A key benefit of using the automatic testing system is to reduce the amount of repeated algorithmic work that needs to be carried out by teaching staff: establishing the correctness of student solutions, and providing basic feedback on their code solutions is virtually free as it can be done automatically.

This allowed us to very significantly increase the number of exercises that students carry out as part of the course, which helped the students to more actively engage with the content and resulted in deeper learning and greater student satisfaction.

The marking system frees teaching staff time that would otherwise have been devoted to manual marking, and which can now be used to repeat material where necessary, explain concepts, discuss elegance, cleanness, readability and effectiveness of code, and suggest alternative or advanced solution designs to those who are interested, without having to increase the number of contact hours.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

Because of the more effective learning through active self-paced exercises, we have also been able to increase the breadth and depth of materials in some of our courses without increasing contact time or student time devoted to the course.

Quality of automatic feedback provision. The quality of the feedback provision involves two main aspects: (i) the timeliness, and (ii) the usefulness, of the feedback.

The system typically provides feedback to students within 2 to 3 minutes of their submission (inclusive of an email round-trip time on the order of a couple of minutes). This speed of feedback provision allows and encourages students to iteratively improve submissions where problems are detected, addressing one issue at a time, and learning from their mistakes each time. This near-instant feedback is almost as good as one could hope for, and is a very dramatic improvement on the situation without the system in place (where the provision of feedback would be within a week of the deadline, when an academic or demonstrator is available in the next practical laboratory session).

The usefulness of the feedback depends on the student's ability to understand it, and this is a skill that takes time and practice to acquire. As we use the standard Python traceback to report errors, we suggest that it is an advantage to encourage students to develop this ability at an early stage of their learning. Students at Southampton are well-supported in acquiring these skills, including timetabled weekly laboratories and help sessions staffed by academics and demonstrators. Once the students master reading the output, the usefulness of the feedback is very good: it pinpoints exactly where the error was found, and provides – through didactic comments – the rationale for the choice of test case as well.

A third aspect of the quality of feedback and assessment is objectivity: the system also improves the objectivity of our marking compared to having several people each interpreting the mark scheme and applying their interpretations to student work.

A more detailed and thorough discussion with respect to the flexible learning opportunities that the automatic testing provides in practical use is provided within Section 4 of [3].

Robustness and performance. The chosen user interface is based on sending and receiving email and is thus asynchronous. The testing server pulls emails using imap and fetchmail, and received emails are stored in a file-based queue. A "receipt email" is sent to the students, and the submitted files are tested in order of their arrival. Finally, results are communicated with a second email. (See Figure 1 in [3] for details.) The system is robust towards interruption of the network or failures of the server as the state is captured through emails (which are inherently robust regarding network failures) and in files once the emails have arrived on the system.

The system provides scalable automatic feedback provision: we have used the system with up to 500 students in one course, and not experienced any noticeable delays in the feedback time. In the academic year 2019-2020, at Southampton

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

the testing system was running in a Linux operating system, hosted on a virtual machine with 4GB of RAM on a single core of an Intel Xeon E5-2697A v4 @ 2.60GHzE5 CPU. In the earlier years of using the system, an older CPU was used and the machine had 1GB RAM. Each test for a student submission typically takes a few seconds to complete. A test that has not completed after one minutes is interrupted: this protects from errors such as infinite loops, and ensures the testing queue cannot be blocked. An appropriate email message is fed back to the student, if such a long test is detected. Memory requirements arise from the nature of the test problem and are thus controllable, and for the courses described here low.

More detailed discussion is available [3] with respect to the dependability and resilience of the system [3, Section 3.4.7], flexible learning opportunities that the automatic testing provides [3, Section 4.8.3], support of large class teaching [3, Section 4.8.4], student satisfaction [3, Section 4.8.5], dealing with syntax errors in student submissions [3, Section 4.4.1], use of undeclared non-ASCII encoding [3, Section 4.4.2], and changing PEP8 [10] standards [3, Section 4.4.3]. We have also connected the system to Moodle [3, Section 4.5], used it to assess C code [3, Section 4.6], and used it to pre-mark exams the students have written [3, Section 4.7].

5 Summary

We have reported on the automatic marking and feedback system that we developed and deployed for teaching programming to large classes of undergraduates. We provided statistics from one year of use of our live system, illustrating that the students took good advantage of the "iterative refinement" model that the system was conceived to support, and that they also benefited from increased flexibility and choice regarding when they work on, and submit, assignments. The system has also helped reduce staff time spent on administration and manual marking duties, so that the available time can be spent more effectively supporting those students who need this. Attempting to address some of the shortcomings of other literature in the field as perceived by a recent review article, we provided copious technical details of our implementation in the supplementary report [3]. With increasing class sizes forecast for the future, we foresee this system continuing to provide us value and economy whilst giving students the benefit of prompt, efficient and impartial feedback.

Acknowledgements. This work was supported by the British Council, the Engineering and Physical Sciences Research Council (EPSRC) Doctoral Training grant EP/G03690X/1 and EP/L015382/1, and the OpenDreamKit Horizon 2020 European Research Infrastructure project (676541).

Data shown in this manuscript and [3] is available in Reference [11].

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45

References

1. A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.
2. A. Robins, J. Rountree, and N. Rountree. Learning and Teaching Programming: A Review and Discussion. *Computer Science Education*, 13(2):137–172, 2003.
3. H. Fangohr, N. O'Brien, A. Prabhakar, and A. Kashyap. Teaching Python programming with automatic assessment and feedback provision. Technical report, University of Southampton, IIT Madras, IIT Mandi, 2015. https://arxiv.org/pdf/1509.03556.pdf.
4. Arne Johanson and Wilhelm Hasselbring. Software engineering for computational science: Past, present, future. *Computing in Science & Engineering*, pages 1–1, 2018.
5. H. Fangohr. A comparison of C, MATLAB, and Python as teaching languages in engineering. In M. Bubak, G.D. van Albada, P.M.A. Sloot, and J. Dongarra, editors, *Computational Science – ICCS 2004*, volume 3039 of *Lecture Notes in Computer Science*, pages 1210–1217. Springer Berlin Heidelberg, 2004.
6. P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93, New York, NY, USA, 2010. ACM.
7. N. Tillmann, J. de Halleux, T. Xie, S. Gulwani, and J. Bishop. Teaching and learning programming and software engineering via interactive gaming. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 1117–1126, May 2013.
8. K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 1st edition, 2003.
9. Holger Krekel, Bruno Oliveira, Ronny Pfannschmidt, Floris Bruynooghe, Brianna Laugher, and Florian Bruhin. pytest 3.7, 2004.
10. G. van Rossum, B. Warsaw, and N. Coghlan. PEP 8 - Style Guide for Python Code. Online, 2016. Accessed at `https://www.python.org/dev/peps/pep-0008/`, 16 December 2016.
11. Supplementary material: data used in figures, 2016. https://arxiv.org/src/1509.03556/anc.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50436-6_45