Experiences With Efficient Methodologies for Teaching Computer Programming to Geoscientists

Christian T. Jacobs,[1,a] Gerard J. Gorman,[1] Huw E. Rees,[2] and Lorraine E. Craig[1]

ABSTRACT

Computer programming was once thought of as a skill required only by professional software developers. But today, given the ubiquitous nature of computation and data science it is quickly becoming necessary for all scientists and engineers to have at least a basic knowledge of how to program. Teaching how to program, particularly to those students with little or no computing background, is well-known to be a difficult task. However, there is also a wealth of evidence-based teaching practices for teaching programming skills that can be applied to greatly improve learning outcomes and the student experience. Adopting these practices naturally gives rise to greater learning efficiency—this is critical if programming is to be integrated into an already busy geoscience curriculum. This article considers an undergraduate computer programming course, run during the last five years in the Department of Earth Science and Engineering at Imperial College London. The teaching methodologies that were used each year are discussed, along with the challenges that were encountered and how the methodologies affected student performance. Anonymized student marks and feedback are used to highlight the discussion, and also how the adjustments made to the course eventually resulted in a highly effective learning environment. © *2016 National Association of Geoscience Teachers*. [DOI: 10.5408/15-101.1]

Key words: computer programming, undergraduate, teaching methodology, feedback

INTRODUCTION

Computer programming is increasingly becoming an essential skill for geoscientists as the world becomes more digitalized. We might commonly think of computer programming as an activity carried out by a relatively small number of domain specialists developing complex application software, perhaps in collaboration with computer scientists. However, geoscientists and engineers are faced with numerous day-to-day tasks such as manipulating datasets (e.g., standardizing, reformatting, or filtering), statistical analysis, plotting, or automating repetitive tasks such as rerunning the same program with many different data inputs and gathering the results for analysis. Even in the case of software use in which no active programming is required, a basic knowledge of computer programming and experience in debugging can be critical when troubleshooting third-party software. This is because programming skills provide the user with a conceptual model for understanding what might be going wrong, systematically characterize the problem, and then either modify their workflow or constructively engage the software developers to resolve the problem. Enhanced computing power enables simulation and data inversion to play a greater role in discovery and prediction in the geosciences. Arguably, we are rapidly approaching a point where innovations will primarily come from those who are able to translate an idea into an algorithm, and then into computer code.

It has long been recognized that teaching basic programming skills to novices is difficult (Robins et al., 2003). Winslow (1996) suggests that it takes about 10 y of experience to turn a novice into an expert programmer. It is worrying that these conclusions were mostly drawn from teaching computer science students for whom computing dominates the curriculum. Therefore, it follows that careful consideration needs to be given to the design of an introductory programming course when it only forms a small part of a noncomputing curriculum. There are also motivational issues due, in part, to the subject being largely associated with the field of computer science, and geoscientists are therefore often surprised to see it as part of their own curriculum. This can lead to the opinion that the subject is not worth pursuing, or the worry that they do not have the background or potential to do well in the subject. Even in the case in which the student does have a strong computing background, learning how to program, much like learning to swim or how to ride a bike, requires a great deal of practice. The learning experience is likely going to be completely different to the experiences the student is used to. Not only is there a large amount of unfamiliar material and knowledge to understand, the student must also adapt to different methods of content delivery and a highly practical learning methodology. Furthermore, the mind has to be trained to think “like a computer” (i.e., to follow a series of steps in a logical way as a process).

In light of these challenges, a considerable amount of research has gone into developing effective strategies for teaching a course in introductory programming; an excellent review is given by Pears et al. (2007). A key consideration in a geoscience context (or indeed any course outside a dedicated computer science degree) is that introductory programming is not being taught as part of a wider computer science curriculum, but instead has to fit within an already full geoscience curriculum. One specific concern is that a single introductory programming course is unlikely

Received 19 May 2015; revised 13 January 2016 and 24 May 2016; accepted 9 June 2016; published online 19 August 2016.
[1]Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, UK SW7 2AZ
[2]Educational Development Unit, South Kensington Campus, Imperial College London, London, UK SW7 2AZ
[a]Author to whom correspondence should be addressed. Electronic mail: c.jacobs10@imperial.ac.uk. Tel.: +44 (0)20 7594 7333. Fax: +44 (0)20 7594 7444

TABLE I: Total number of students registered on the programming course each year. In 2010, the programming course was optional and the 35 students represent only a fraction of the total number of first year students in the department that year. In 2012, two separate classes were run: one comprising 73 first year students and one comprising 89 second year students, in order to transition the course from being a second year course to a first year course in later years; hence, the larger total number of students.

Year	Number of Students
2010	35
2011	89
2012	162
2013	85
2014	87

to enable the students to take these skills and reapply them to a different problem-solving context from that in which they were presented (Palumbo, 1990). Therefore, we also need to consider where else in the curriculum there are opportunities to use and extend students' programming skills and experience.

The choice of a first programming language also has a significant impact on learning. While the top three popular programming languages have been consistently C, Java, and C++ for many years (TIOBE Software, 2015), they are generally not thought to be good choices as a first programming language (Mody, 1991; Biddle and Tempero, 1998; Churcher and Tempero, 1998; Clark et al., 1998; Close et al., 2000). A big part of the problem with starting with such programming languages is that they are too low-level, and the high cognitive load associated with the syntax (Stefik and Siebert, 2013) has little to do with learning to think algorithmically and writing structured programs (Pears et al., 2007). It is interesting to note that Pears et al. (2007) also point out that similar learning issues related to excessive cognitive load arise when using professional integrated development environments (IDEs) in introductory programming due to the effort that must be invested to become a proficient user. For this reason, high-level languages such as Python are a popular choice because of their much simpler, higher-level syntax (Donaldson, 2003; Fangohr, 2004; Lin, 2012).

Böszörményi (1998) points out that a first programming language should not be selected in isolation but instead, in the context of the entire curriculum. There are many opportunities to reinforce learning if the same language can be used as a tool in other lecture courses and project work. However, this relies upon a common denominator (i.e., programming language) being accepted by the teaching staff. While many computer languages (e.g., C/C++, FORTRAN, Java) are used by research and academic staff, Python has become very popular as it is both simple and powerful. A lot of international effort has gone into providing Python interfaces to popular geoscience packages to the extent that Python can be used seamlessly across laptops, supercomputers, and cloud platforms. Examples of these packages include: GRASS GIS (GRASS Development Team, 2015), ArcGIS (Environmental Systems Research Institute, 2015), and QGIS for GIS and geomorphology (QGIS Development Team, 2009); PyLith for modeling crustal deformation (Aagaard et al., 2013; Aagaard et al., 2016); ObsPy for processing seismology data (Beyreuther et al., 2010); and Firedrake for geophysical fluid dynamics (Jacobs and Piggott, 2015; Rathgeber et al., 2015). The wide variety of Python-based geoscientific computing packages on offer provides an environment in which students feel motivated from day one that they are learning a language that can be directly applied professionally, while still benefiting from a simple syntax with relatively low cognitive overhead so they can focus on learning to think algorithmically.

In 2010, we began to develop an eight-week introductory programming course with 24 contact hours for undergraduate students majoring in geoscience, in the Department of Earth Science and Engineering at Imperial College London.[3] The objective of the course was to teach fundamental principles of programming and basic constructs such as variables, loops, conditional statements, array manipulation, plotting, classes, and objects. Each year the class size was usually between 70 and 90 students (the exact number of students can be found in Table I).

The students embarking on undergraduate study at Imperial College London have a broad range of backgrounds, but these are mostly STEM-based[4] as a result of Imperial College London's focus on STEM subjects. However, no prior knowledge of higher mathematics, programming, or computing was assumed. Each year comprised only new students; there were no course "retakers/resitters" from previous years. Of course, the individual student backgrounds and experiences vary from year to year and, in general only a few (approx. one to six) students had some prior programming experience, sometimes obtained by doing an A-Level[5] computing course at a Further Education institution prior to embarking on undergraduate study. In the 2010 class there were two students with an A-Level in Computing, in 2012 there was one, and in 2013 there were two, with no students having an A-Level in Computing in other years (see Table II for a full list of student A-Level qualifications by subject). This is typically very different from classes majoring in computer science, in which a much larger proportion of the intake have some past programming experience, or at least a strong background in logical/algorithmic thinking, which will aid them considerably when learning to program. With respect to demographics, UK "home" students (i.e., students who are ordinarily resident in the UK) made up the majority of each intake, and all years featured a larger number of male students, as detailed in Table III.

While Imperial College London is a research-intensive university with a worldwide reputation for research excellence, it has made a clear commitment to delivering a world class education for its students; in light of this, teachers across the institution are encouraged to adopt an iterative approach to course design and are offered the freedom to

[3] Originally, this was a second year course. In 2012 we had to transition the course from being a second-year only course to being a first-year only course; this was accomplished by teaching both first year (i.e., the 2012 intake) and second year (i.e., the 2011 intake) students in the same course that year.

[4] STEM stands for Science, Technology, Engineering, and Mathematics.

[5] An A-Level, more formally known as General Certificate of Education (GCE) Advanced Level, is a qualification offered by Further Education institutions in the United Kingdom.

TABLE II: Total number of A-Level (Further Education) qualifications attained (by subject) by each year's departmental student intake. The 2010 and 2011 programming classes were for second year students, respectively, corresponding to the 2009 and 2010 intake. Not all of the students from the 2009 intake took the programming course in 2010, since it was optional just for that year, which is why the number of A-Levels in 2009 sometimes exceeds the total number of 2010 programming students in Table I. The 2012 programming class combined both the first and second year students, corresponding to the 2011 and 2012 intake, respectively.

Subject	2009	2010	2011	2012	2013	2014
Maths	51	68	71	67	75	74
Physics	37	53	57	49	55	60
Chemistry	38	45	52	51	63	52
Geology	18	24	24	13	35	26
Biology	19	27	25	20	26	22
Computing	2	0	1	0	2	0
Other	46	79	39	70	74	16

innovate with a view to making teaching as effective as possible. As a starting point for developing the course, we adopted the textbook, *A Primer on Scientific Programming with Python* (Langtangen, 2009 and later editions in subsequent years), which is targeted at science and engineering students with no previous programming experience. The author of the textbook, Hans Petter Langtangen, also provided us with his collection of slides based on the book, which he had developed for his own introduction to programming course at the University of Oslo. While the core topics and structure remained largely the same throughout the years that the course was run, both the teaching medium and methodology was changed radically in response to student performance and experience.

Put briefly, our initial approach was to teach in a traditional way: a 3-h lecture block to cover material, with students being given exercises at the end of the lecture as coursework. It quickly became apparent that this approach would require a full additional 3-h block timetabled in the computer lab with tutors to support the students doing the exercises. Not only did this put additional pressure on the timetable, it also resulted in the poorest student performance and experience of the five years of teaching the course.

TABLE III: Demographic information for each year's departmental student intake. Note that the term "EU" (European Union) does not include the UK here, and the term "Overseas" denotes any non-EU or non-UK country, except qualifying overseas territories determined by The Education (Fees and Awards) (England) Regulations 2007. The 2010 and 2011 programming classes were for second year students corresponding to the 2009 and 2010 intake, respectively. The 2012 programming class combined both the first and second year students, corresponding to the 2011 and 2012 intake, respectively.

Year	Gender (Male:Female)	Student Status (UK:EU:Overseas)
2009	39:25	55:0:9
2010	56:33	79:3:7
2011	50:39	68:10:11
2012	44:29	61:3:9
2013	53:32	64:4:17
2014	53:34	60:5:22

In subsequent years, many changes were made to the course in response to the student experience that were guided by the experience of the teaching community and pedagogy literature. Our current approach delivers satisfactory learning outcomes, and focuses on using the IPython Notebook software (Pérez and Granger, 2007) with a blended learning approach. This approach is largely inspired by the teaching practices promoted by the Software Carpentry organization (www.software-carpentry.org), which teaches basic computing skills to scientists (Wilson, 2006; Wilson, 2014). Currently, the 3-h block is broken down into a series of 10–15 min of lecturing to establish the context and motivation of the current topic. These include live examples that are worked out and discussed. Between these minilectures are practical exercises in which the students are allocated approximately 30 min to work on a few exercises related to the minilectures, with teaching assistants and peers providing support. The IPython Notebook software neatly integrates core course content written in the Markdown language (Gruber, 2004) with Python code that can be run interactively within the same document.

In this article, we report on our experience and the impacts of changing our teaching methodology on the performance of the undergraduate geoscience students. Of course, it can be rather difficult to directly attribute improvements in student learning outcomes to any one specific change in teaching methodology. Each year's student cohort is different and there could be other factors involved, such as variations in the backgrounds of the teaching assistants each year. However, in our study we rely on the fact that we kept the exam format and difficulty level consistent throughout the years. Furthermore, our experiences suggest that, because of the broad range of backgrounds, if we do not apply any particular methodology then some students perform poorly while some do very well. On the other hand, if we do apply particular pedagogical techniques, a consistent positive learning outcome is achieved.

In the Data Collection section, we give details of the data collected for evaluating student learning and effectiveness of the course. In the Current Curriculum and Instruction Methodology section, we present the various design aspects of the current (2014) state of the course, and the teaching practices employed. For each aspect of the course, we also contrast with previous years by detailing what we changed each year, why we did this, and the justification for why we believe such changes have helped form an efficient teaching and learning methodology.[6] Finally, some closing remarks and recommendations for adopting the current course methodology elsewhere are given in the Conclusion.

DATA COLLECTION

This section describes the types of data that we have collected with respect to student performance and feedback, and the techniques used to obtain this data.

SOLE: Student OnLine Evaluation

SOLE[7] is the central Student OnLine Evaluation tool developed and maintained by Imperial College London and the Students' Union. The tool enables students to give their view of their lecturers and modules each year, and also facilitates the assessment of the quality of their degree program. All surveys are anonymous and are run at the end of each term. The evaluation tool surveys all undergraduate students on their modules and the lecturers who have taught those modules during the current term. The SOLE module/ lecturer evaluation consists of a set of scoring criteria per module and per lecturer (see Appendix A for the criteria used in each year).

As part of the annual review of the department for quality assurance and enhancement purposes, a report is written summarizing key issues from SOLE. Individual staff give feedback to the students, often in the same academic year, and at the start of the next time the course is taught. In particular, issues raised by the students and adjustments made to resolve each issue are highlighted (e.g., "Students commented that not enough examples were discussed in depth so now we have significantly increased the number of examples walked through in lectures."). A summary of the main outcomes is then sent to each individual student to complete the feedback loop. It should be noted that this feedback only represents the students' perceptions of the course and the lecturer, and is not a measure of actual learning.

Response Rate

Since 2009, in the Department of Earth Science and Engineering, the response rate for SOLE is between 99% to 100%, thus reducing the risk of sample bias. This gives a complete picture when all students take the time to complete the survey and give their views. The survey for the Introduction to Programming for Geoscientists course is completed during week seven out of eight classes. There is a strong culture of communication and partnership within the department, and students welcome the opportunity to provide feedback on their modules. They see the outcomes of each SOLE survey and know that their comments make a difference for future classes and students. Students are given an induction in their first year about how SOLE works and reminded of it annually.

Changes to Survey Questions/Scoring Criteria

An inconsistency in the survey criteria occurred in 2012, when the number of criteria was reduced in the hope it would boost response rates across the university as a whole. The survey was reinvented for subsequent years when it was clear that the reduction in criteria did not impact response rates. In addition, two new criteria ("I have received helpful feedback on my work" and "The content of the module is intellectually stimulating") were added, while two criteria ("The organization of the module" and "The support materials available for this module") were removed.

Reflecting up-to-date pedagogic thinking, other minor changes of wording appeared, as with the above. For example, in 2010–2012 students were asked to rate the structure of the *lectures* or *teaching sessions*, whereas in 2013 and 2014 they were asked to rate the structure of the *module*. This word replacement reflected the wider nature of the survey from traditional "talk and chalk" lectures.

One other change was the wording of the response, again to reflect up-to-date learning terminology in the UK: "Very Good (A), Good (B), Satisfactory (C), Poor (D), Very Poor (E), No Response (F)" were replaced by "Definitely Agree (A), Mostly Agree (B), Neither Agree nor Disagree (C), Mostly Disagree (D), Definitely Disagree (E), Not applicable (F)." The scoring system used to interpret student feedback in a quantitative fashion in the Current Curriculum and Instruction Methodology section is given by

$$S = \frac{\sum_{i=A}^{E} w_i n_i}{\sum_{i=A}^{E} n_i} \tag{1}$$

where S is the score for a given criterion, w_i is the numerical value assigned to response i (responses A to E are given scores that scale linearly such that $w_A = 2, w_B = 1$, $w_C = 0$, $w_D = -1$, $w_E = -2$), and n_i is the total response count for response i.

The same scoring system is used throughout the 5 y the course was run, despite changes in the wording of the responses. An average score of -2 to -1.5 indicates Very Poor, -1.5 to -0.5 indicates Poor, -0.5 to 0.5 indicates Satisfactory, 0.5 to 1.5 indicates Good, and 1.5 to 2 indicates Very Good.

Examination Results

The programming course had no marked term-time coursework component, and was instead graded based on a 2-h end-of-term examination. In 2010 this involved answering a series of programming-related questions by hand-written word (in the computer lab), while in the later years it involved completing a series of practical exercises and submitting the Python source code files at the end of the exam. Anonymized mark distributions are presented along with the SOLE feedback in the Current Curriculum and Instruction Methodology section. These data are also provided in Table IV, Table V, Table VI, Table VII, and Table VIII, and illustrated in Fig. 1, Fig. 2, Fig. 3, Fig. 4, and Fig. 5, for the years 2010, 2011, 2012, 2013, and 2014, respectively.

CURRENT CURRICULUM AND INSTRUCTION METHODOLOGY

In this section, we will describe the current state of the course's instruction methodology and explain the advantages of our approach. We then compare and contrast the

[6] Here we define an "efficient methodology" as an approach to teaching that (1) not only delivers the course material within the time and resource constraints, but also (2) improves the students' programming knowledge and skillset, and (3) effectively achieves all the desired learning outcomes and course objectives.

[7] https://www.imperial.ac.uk/students/academic-support/student-surveys/ug-student-surveys/ug-sole/

TABLE IV: Module (top) and lecturer (bottom) scores for various criteria, in the year 2010. An average score of −2 to −1.5 indicates Very Poor, −1.5 to −0.5 indicates Poor, −0.5 to 0.5 indicates Satisfactory, 0.5 to 1.5 indicates Good, and 1.5 to 2 indicates Very Good. The relatively high scores here suggest that the traditional lecturing style met the students' expectations.

Module Criterion	Score
Support material	1.20
Organization	0.89
Structure/delivery	1.11
Explanation	1.11
Approachability	1.65
Interest generated	1.26
Mean	1.20 (Good)
Lecturer Criterion	**Score**
Structure/delivery	1.11
Explanation	1.11
Approachability	1.65
Interest generated	1.26
Mean	1.28 (Good)

current state with the previous years that the course was run, describing the changes that were made, and give evidence for why these changes were successful. This evidence comprises trends seen in the SOLE data, student comments, and the exam marks. We also explain what changed between the years to cause an improvement in the marks and comments, and how it affected the learning outcomes.

TABLE V: Module (top) and lecturer (bottom) scores for various criteria, in the year 2011. An average score of −2 to −1.5 indicates Very Poor, −1.5 to −0.5 indicates Poor, −0.5 to 0.5 indicates Satisfactory, 0.5 to 1.5 indicates Good, and 1.5 to 2 indicates Very Good. The relatively high scores suggest that the traditional lecturing style met the students' expectations here.

Module Criterion	Score
Support material	0.58
Organization	0.82
Structure/delivery	0.65
Explanation	0.37
Approachability	1.10
Interest generated	0.95
Mean	0.75 (Good)
Lecturer Criterion	**Score**
Structure/delivery	0.65
Explanation	0.37
Approachability	1.10
Interest generated	0.95
Mean	0.77 (Good)

TABLE VI: Module (top) and lecturer (bottom) scores for various criteria, in the year 2012. An average score of −2 to −1.5 indicates Very Poor, −1.5 to −0.5 indicates Poor, −0.5 to 0.5 indicates Satisfactory, 0.5 to 1.5 indicates Good, and 1.5 to 2 indicates Very Good. The much lower scores here suggest that the students were put off by the flipped classroom format since it was not what they were used to.

Module Criterion	Score
Structure/delivery	0.38
Content	0.49
Mean	0.44 (Satisfactory)
Lecturer Criterion	**Score**
Structure/delivery	0.38
Mean	0.38 (Satisfactory)

Blended Learning Approach

The current state of the course adopts a blended learning approach, following many of the pedagogical methods used by the Software Carpentry organization (Wilson, 2006; Wilson, 2014). Blended learning essentially brings together different modes and techniques of content delivery (Bonk and Graham, 2006; Friesen, 2012). Concepts and ideas are introduced via face-to-face interaction to build a solid theoretical knowledge base. A computer-mediated environment is then used to help apply this knowledge in a practical sense, enhance the students' learning, and develop the necessary skills. Our experience has found that this is an example of a highly efficient methodology since, much like learning to swim or learning a new spoken language, the students' time is most effectively applied to developing their programming skills through practice while still having the face-to-face component to acquire the necessary theoretical and background knowledge. Indeed, it has already demonstrated success in other computer programming classes (e.g., Boyle et al., 2003).

TABLE VII: Module (top) and lecturer (bottom) scores for various criteria, in the year 2013. An average score of −2 to −1.5 indicates Very Poor, −1.5 to −0.5 indicates Poor, −0.5 to 0.5 indicates Satisfactory, 0.5 to 1.5 indicates Good, and 1.5 to 2 indicates Very Good. The much lower scores here (particularly for the Explanation criterion) suggest that the students were put off by the flipped classroom format since it was not what they were used to.

Module Criterion	Score
Structure	0.45
Intellectual stimulation	0.72
Feedback	0.47
Overall satisfaction	0.36
Mean	0.50 (Satisfactory)
Lecturer Criterion	**Score**
Explanation	−0.06
Interest generated	0.67
Approachability	1.33
Overall satisfaction	0.71
Mean	0.66 (Good)

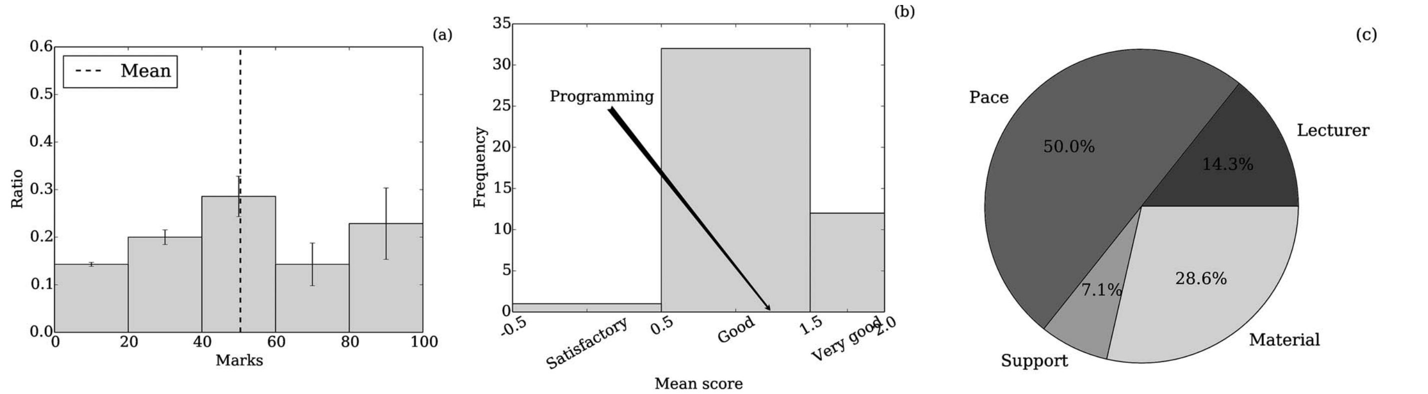
TABLE VIII: Module (top) and lecturer (bottom) scores for various criteria, in the year 2014. An average score of −2 to −1.5 indicates Very Poor, −1.5 to −0.5 indicates Poor, −0.5 to 0.5 indicates Satisfactory, 0.5 to 1.5 indicates Good, and 1.5 to 2 indicates Very Good. The scores were much higher for this year after reassuring and justifying the blended learning approach to the students.

Module Criterion	Score
Structure	1.26
Intellectual stimulation	1.30
Feedback	1.32
Overall satisfaction	1.21
Mean	1.27 (Good)
Lecturer Criterion	**Score**
Explanation	1.11
Interest generated	1.59
Approachability	1.73
Overall satisfaction	1.69
Mean	1.53 (Very good)

In the case of our programming class, the students are expected to read the lecture notes beforehand. The 3-h workshops are divided up into intervals; since empirical evidence has shown that the average adult student can only maintain focus for approximately 15–20 min (Middendorf and Kalish, 1996), each interval comprises approximately 10 min of lecturing to establish the context of a particular section of the lecture notes, followed by a period of time for the students to complete the exercises individually with the lecturer and teaching assistants available to help. Empirical evidence has also shown that students who see worked examples before attempting exercises by themselves are better able to tackle future problems (Guzdial, 2015). With this in mind, worked examples are presented during the short lecture before the actual workshop session begins when students can attempt new problems on their own with support on hand if they need it.

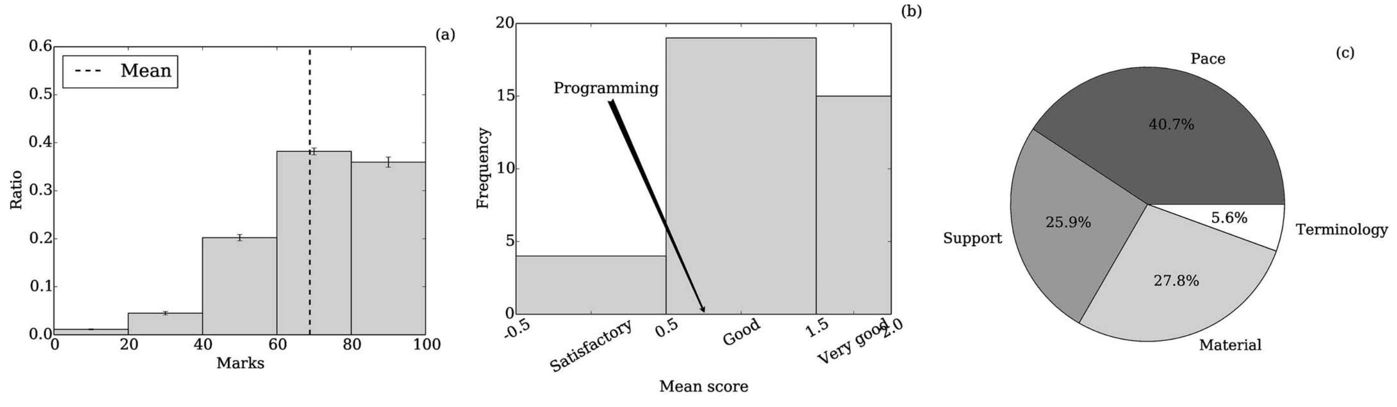
At the preuniversity level, many students are used to a learning environment in which traditional passive learning and "note-taking" classes are the norm. The student spends the day in a passive state, recording the knowledge being delivered to them by the teacher, often by taking notes directly from the teacher's blackboard. The student is expected to absorb this knowledge and regurgitate it for examination purposes, but they have little responsibility over their own learning. We found that this expectation of a passive, teacher-led environment can therefore be very difficult to change as the student starts university for the first time, particularly as this is still the dominant method of lecturing in most universities. This is a particular issue for computer programming, as this course is taught in the first term of their first year. Reassurance, explicit justification, and a brief explanation of the blended learning approach are therefore given in the very first lecture of the course.

The blended learning methodology can potentially be applied to many other fields of study, not just to programming classes. Indeed, blended learning has seen success in the teaching of biology (Yapici and Akbayin, 2012), finance/accounting (Dowling et al., 2003), and human



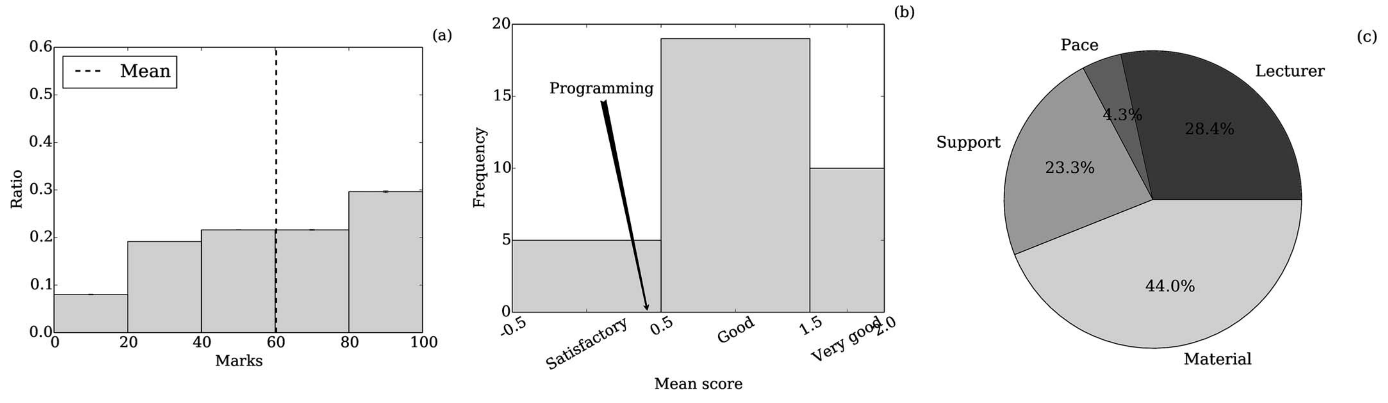
- Lectured in the traditional sense but with less time for practical exercises.
- Largely negative sentiments about pace and difficulty of material.
- Students were positive about traditional lecturing approach, although the learning outcomes were poor.

FIGURE 1: Data for the 2010 class: (a) histogram of final exam marks, (b) histogram showing mean (combined) module and lecturer SOLE scores for all modules in the Department of Earth Science and Engineering, (c) pie chart showing the emergent topics from the SOLE feedback, and (bottom row) overall student sentiments. There were 35 students who sat the final exam in 2010. Note that in all years, there were no mean combined (lecturer and module) SOLE scores below Satisfactory.



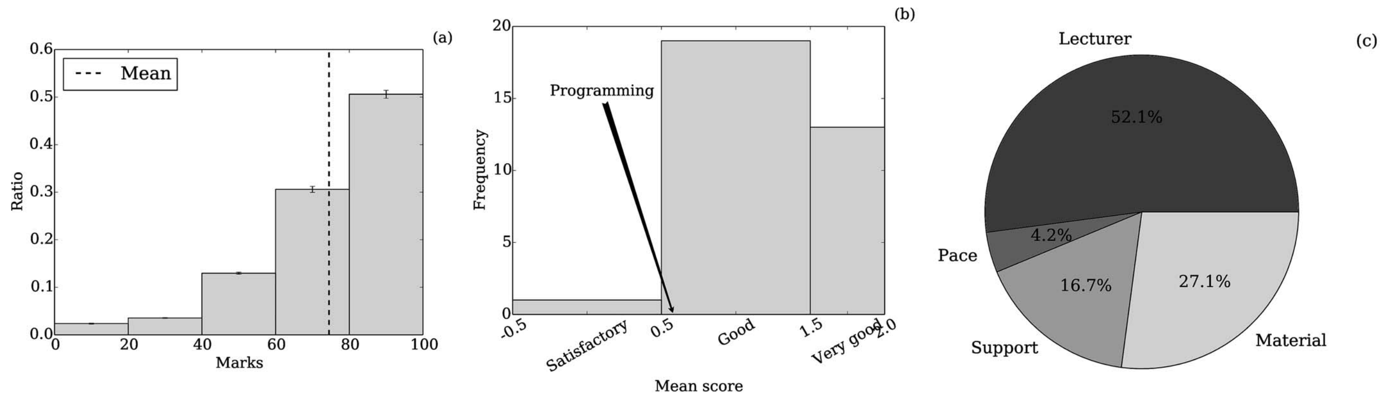
- Doubled the teaching time to 6 hours by introducing a separate 3-hour practical workshop.
- Significantly more positive comments about the support received.
- However, the pace was still too fast and material still hard to follow.

FIGURE 2: Data for the 2011 class: (a) histogram of final exam marks, (b) histogram showing mean (combined) module and lecturer SOLE scores for all modules in the Department of Earth Science and Engineering, (c) pie chart showing the emergent topics from the SOLE feedback, and (bottom row) overall student sentiments. There were 89 students who sat the final exam in 2011. Note that in all years, there were no mean combined (lecturer and module) SOLE scores below Satisfactory.



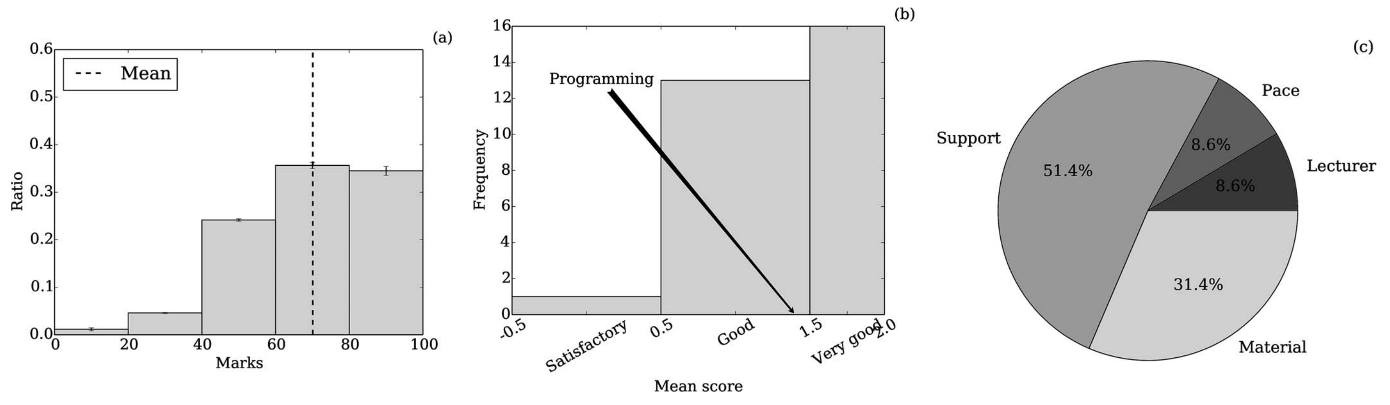
- YouTube videos introduced.
- Largely negative comments about the lecturer and method of delivery, as a result of the moving away from the traditional passive classroom environment.
- Comments regarding the material were, unlike previous years, mostly positive; students liked that they could work through the lecture material at their own pace, and go back to repeat certain parts of the video. Furthermore, the students reponded positively to the removal of advanced material such as classes and objects.

FIGURE 3: Data for the 2012 class: (a) histogram of final exam marks, (b) histogram showing mean (combined) module and lecturer SOLE scores for all modules in the Department of Earth Science and Engineering, (c) pie chart showing the emergent topics from the SOLE feedback, and (bottom row) overall student sentiments. There were 162 (73 first year and 89 second year) students who sat the final exam in 2012. Note that in all years, there were no mean combined (lecturer and module) SOLE scores below Satisfactory.



- Moved away from the video and traditional style of lecturing, to a blended learning approach.
- Positive feedback regarding one-to-one support, and almost no negative comments regarding quantity of material covered, unlike previous years.
- But the majority of comments about the lecturing style were negative because it did not match with their expectations of what a traditional lecture was.

FIGURE 4: Data for the 2013 class: (a) histogram of final exam marks, (b) histogram showing mean (combined) module and lecturer SOLE scores for all modules in the Department of Earth Science and Engineering, (c) pie chart showing the emergent topics from the SOLE feedback, and (bottom row) overall student sentiments. There were 85 students who sat the final exam in 2013. Note that in all years, there were no mean combined (lecturer and module) SOLE scores below Satisfactory.



- Explained the blended learning approach and emphasised the benefits throughout the course.
- Split up the workshops into 'bite-size' chunks; a series of 10 minute lectures followed by practical exercises, rather than one long practical session.
- Students felt reassured, resulting in only positive comments regarding the style of lecturing.
- The sticky notes and ability to give feedback at the end of each workshop resulted in positive comments regarding support.

FIGURE 5: Data for the 2014 class: (a) histogram of final exam marks, (b) histogram showing mean (combined) module and lecturer SOLE scores for all modules in the Department of Earth Science and Engineering, (c) pie chart showing the emergent topics from the SOLE feedback, and (bottom row) overall student sentiments. There were 87 students who sat the final exam in 2014. Note that in all years, there were no mean combined (lecturer and module) SOLE scores below Satisfactory.

anatomy (Pereira et al., 2007), for example. However, while the face-to-face delivery will be similar (i.e., lecturing with slides or a blackboard), the exact form of the technology-based media may be different in the case of nonprogramming classes. In the case of an electronics class, for example, the practical exercises may involve designing a circuit rather than writing a program.

Comparison With Previous Years

In contrast to the first years that the course was run (2010–2013), in which students were taught through 2-h lectures with the use of slides, blended learning was extremely successful and significantly boosted student performance.

The initial passive lecturing approach was more in line with student expectations. This is illustrated by the 2010 SOLE feedback, with 94% of students giving a rating of Satisfactory or higher for the category "The structure and delivery of the lectures." Furthermore, the "Structure and delivery of course materials" scored a relatively high (Good) mark of 1.11 based on the SOLE feedback (see Table IV), as did the course in general (as shown by the Good mean combined score [i.e., the mean of the module and lecturer scores] of 1.24; see Fig. 1b). The lecturer/lecturing style was not the main trending issue in that year's student survey (see Fig. 1c). Similarly, a high mean combined score was achieved in 2011 (see Fig. 2b), with a Good score of 0.65 for "Structure/delivery" (see Table V). This standard approach of lecturing for 3 h may be seen as an efficient methodology from a material dissemination point of view (i.e., delivering a large amount of material to a high number of students all at once; Beard and Hartley, 1984), but is not necessarily effective at achieving learning outcomes (Ramsden, 1992; Isaacs, 1994). Indeed, this proved to be the case, with the 2010 mean exam mark of 50.5% reflecting this outcome (see Fig. 1a). In 2011 it was felt that the addition of an extra 3-h practical session (while still covering the same amount of course material) helped to push mean grades to 68.9%; however, this was not sustainable for future years. In addition, students who could not keep up with the pace of the lectures during 2010 and 2011 quickly lost track (e.g., one student in 2011 commented "the lecture PowerPoints are sometimes delivered too quickly and then [I] cannot understand the lecture from then on."); this was the biggest trending issue based on the student feedback.

In 2012, a set of lectures were recorded and uploaded to YouTube (www.youtube.com). The YouTube video lectures were approximately 10–15 min long, following an approach largely inspired by online teaching resources such as the Khan Academy (www.khanacademy.org) and Massive Open Online Courses (MOOCs; Yuan and Powell, 2013). The students could watch these videos in class at their own pace and complete the exercises within the 3-h timeslot. However, we found that the students in 2012 then felt unsupported by this approach, as they were simply not accustomed to such a flipped classroom approach in which they have much greater responsibility for their own learning:

> *"I think that it would help if the course was actually taught—at the moment, the way it's structured there is no need for a teacher to be in the room as he does not cover any content within the lessons"*

> *"I found the teaching to be very impersonal but maybe that is because programming is something you have to learn for yourself"*

> *"He doesn't lecture"*

> *"People end up being able to do things but not having a clue why as nothing is explained"*

This was also reflected in the SOLE feedback and course scores; a much higher proportion (28%) of first year students indicating that the "The structure and delivery of the lectures" was less than satisfactory, a relatively low score of 0.38 for "Structure/delivery" (see Table VI), and a fairly broad distribution of marks with the mean being 60.3% (see Fig. 3a). One of the reasons for the lack of engagement in this approach may in part be related to the place in the curriculum. As the course is in the first term of their first year in university, not all students had developed their independent learning skills. Furthermore, evidence provided by effect sizes (Hattie, 2008) has shown that web-based learning only has a small positive influence on learning relative to the traditional classroom environment. Despite the negative feedback from the students, the learning outcomes were much better than the first year the course was run.

Because of the lack of constructive impact, we decided to no longer use online videos and instead opted for traditional lecture notes in 2013. A short lecture of 20–30 min was delivered at the beginning of the class, but students were expected to have read the notes before the class began in order to maximize the amount of time that could be spent on the practical exercises that followed. However, while a few students in 2013 understood the need for this self-preparation

> *"He gives you the responsibility to succeed and the tools to do so, I do not see that he can do any more unless he were to baby feed us, which isn't why we are at university."*

the majority of the student feedback regarding expectations from the lecturer still amounted to "the lecturer is not lecturing us":

> *"Being shown a lecture and then expected to complete an exercise with little teaching is difficult."*

> *"The lecturer should teach us the content of the course, instead of us having to read it off notebook with limited explanation to go with those examples."*

> *"More lecture-based learning. Not enough explanation of lecture notes and poor quality lecture notes."*

despite reassuring them that it was more effective for them to read the lecture notes themselves at their own pace and spend the majority of the time doing the exercises, rather than the lecturer spending most of the available time reading the lecture notes out to the students. This was also reflected in the lecturer scores, with a negative score of −0.06 for the "Explained material" category (see Table VII) contributing to a relatively low mean combined score of 0.58 for the whole course, as illustrated in Fig. 4b. Despite this, our data suggested the change to a flipped classroom environment was beneficial to the students' performance; this is reflected in the mark distribution, which is skewed toward the higher end of the spectrum, and the mean course mark of 74.5% was considerably higher than previous years, as shown in Fig. 4a.

Finally, in 2014 when the blended learning approach was adopted (and, crucially, justified to the students), it was clear that students understood the need for such an approach while learning to program. This was clear from the SOLE feedback in 2014:

"I agree with [the lecturer] that we gain much more from practicals than from being lectured."

"[I] understand the need for self teaching."

The score for course structure and delivery increased to an all time high of 1.26, yet the score for explanation of the material also remained high at 1.11 (see Table VIII) through breaking the 3-h workshop down into individual small lectures and reassuring the students that the teaching approach taken was beneficial, albeit unlike what they were used to. The mean combined score of 1.4 for the whole course was also relatively high compared to other courses run by the department that year (see Fig. 5b). At the same time, the mean exam mark did not change considerably. The overall mark distribution looked much like the one of 2011, but required just 3 h of teaching time per week compared to the (unsustainable) 6 h allocated in 2011.

Tailoring of Material, Exercises, and Pace

Students majoring in computer science will immediately see a need for, and typically have a strong interest in, learning to write computer programs. In contrast, our experience with teaching geoscience students has shown that it is crucial to motivate the need for programming from the very first lecture, since many often feel that they are being forced to do something that is not relevant or worthwhile in their undergraduate syllabus. To that end, the course includes several exercises that are tailored toward geophysical scenarios. For example, students are asked to create a program that reads in seismic data from a file and locates the earthquake of the largest magnitude. Another exercise involves reading a tidal gauge data file supplied by the British Oceanographic Data Centre, plotting the tide level against time, and then using that to spot the tidal constituents. In addition, we use our own computational-based research to further justify why students would want to develop programming skills by showing them simulations of volcanic eruptions and seismic wave propagation produced by computer programs.

It is also important to make the learning experience as interactive as possible. We therefore encourage the students to discuss problems with one another to aid peer-learning. One exercise is particularly successful at engaging the students; the exercise tasks them with creating a "Battleship" game in Python, which they later play against their neighbor. Not only is this an enjoyable exercise, but it is also challenging enough to bring together many of the topics the students learn in the course. Furthermore, we observed that this can promote several instances of "rubber duck debugging" (Hunt and Thomas, 1999) wherein students manage to spot inconsistencies between what their program is actually doing and what they expect it to do, simply by walking through their program with one of their peers without that peer necessarily saying anything at all. This form of peer assessment proved to be an efficient way of testing the application of knowledge while allowing a degree of peer instruction as students could correct each other if necessary. The informal nature of the game-based scenario promoted independence, control, active engagement, and enjoyment, which have been held up as key principles in effective teaching and learning in higher education (Ramsden, 1992).

Finally, in order to facilitate effective learning we needed to manage the pace and cognitive load being placed on students without diluting the quality or academic rigor of the course. The course material was therefore refined each year following reviews of the course while still keeping the core learning outcomes in place.

Comparison With Previous Years

Throughout the years that the course was run, there was always initial resentment from a minority of students who, as geoscientists, felt that they were being forced to do something that was not useful (e.g., "I don't understand how it is related to the other stuff we study in geology" [2012]). It was therefore crucial to underline the importance of programming skills from the first lecture with the hope this would have a knock-on effect on the students' motivation to learn. This was partially accomplished by citing real geophysical applications that involve software development. This was reflected in the lecturer-specific feedback given toward the end of the term; in 2010 and 2011, respectively, 97% and 92% of students thought that "The interest and enthusiasm generated by the lecturer" was at least satisfactory. When the SOLE rating options changed in 2013 and 2014, 89% and 99%, respectively, of students either had no opinion, mostly agreed, or strongly agreed that "The lecturer generated interest and enthusiasm." The student comments also show how the resentment was overcome once the student's own resistance to programming was mitigated through motivation: "I feel that although at first I despised programming this was due to my own block against the subject" (2013).

When the course commenced for the first time in 2010, the exercises in the book by Langtangen (2009) were considered. These were largely subject-independent and some students therefore found it hard to relate these exercises to problems that they would deal with as geoscientists in the real world. For example, in 2013 one student "felt that it took some time to get the general picture, the lecturer would better help the students if he applied the idea of programming to real life." The tailoring of the exercises toward geophysical scenarios was therefore a change that was well received by the students.

Many of these exercises required the implementation of equations and mathematical functions (e.g., implementing the Heaviside step function). The students were tasked with translating these formulas into code. Although a very high proportion of the students had strong backgrounds in mathematics at A-Level (see Table II), it is not a prerequisite and the students were not required to understand the mathematics behind the formulas. Nevertheless, the presence of mathematics concerned some students and demotivated them; anonymous student comments from 2013 that highlight this include

> *"Some of the exercises assumed A-Level maths as a prerequisite, which was not appropriate for all students and very distressing for some."*

> *"You have to figure out the maths before you can even start to program."*

> *"A lot of the exercises assume a previous understanding of some complicated maths, which adds to the confusion of the programming itself."*

It was therefore necessary to provide reassurance and demonstrate that an in-depth knowledge of the derivation and use of the mathematical formulas is not required for them to be able to complete the exercise (e.g., showing that a finite sum of sine functions can be implemented using a for-loop).

Particularly during the first few years that the course was run, students largely felt overwhelmed with the amount of material covered ("At the start there was too much to do." [2012]) and that many lectures had to be rushed at the end due to lack of time:

> *"Far too much content to fit into 8 lectures!!!!!! Not once did we finish a lecture."* (2010)

> *"The amount of material is covered in a—sometimes—too short amount of time."* (2011)

This is clearly visible in Fig. 1 and Fig. 2 where the largest (negative) trending topic in 2010 and 2011 concerned the pace of the lectures. The students responded positively to the refinement of the course in 2012, with one student commenting that "The content of the programming for geoscientists was perfect when cut down to only 6 parts." (2012).

Practice

We found that the short live lecture in a flipped classroom environment followed by the bite-sized chunks of practical exercises, was a highly effective and efficient way of developing programming skills. The context, background material, and example code are all covered just before the students attempted the associated exercises. The material is therefore fresh in their minds and gives students a firmer foundation on which to practice their skills in the particular topic under consideration. Furthermore, any questions that students have can be asked at the classwide-level during the live lecture, thereby addressing queries that may be shared by many students at the same time.

Comparison With Previous Years

In comparison with the 2010 run of the course, the students initially spent 2 h being lectured on new material, and then given 1 h to complete a range of practical exercises. However, students were struggling to complete these exercises in the time allocated. The course greatly improved a few weeks into the term after extending the practical sessions by 3 h (on the same day), as reflected in the students' feedback in 2010:

> *"This course definitely improved after the reading week[8] when they introduced workshops in the afternoon with the demonstrators."*

> *"I believe the new structure of having 3 hours of GTA help and worked through solutions help."*

> *"The introduction of a separate practical session was good."*

However, in the same year, at least one student believed that this was "too little too late." Furthermore, it was likely that students became exhausted since the majority of that day involved a constant focus on programming.

When the course commenced in 2011, a stand-alone 3-h practical workshop was held each week so that the students could focus entirely on the exercises; the students responded positively to this ("The workshops are absolutely essential. Well run and incredibly helpful.") and it appears that this resulted in a more positive skewness in the mark distribution for 2011 as shown in Fig. 2a. However, the extra 3 h of allocated practical time was unsustainable for future years, and a more efficient means of content delivery and practice was therefore required.

The use of YouTube videos in 2012 permitted the students to review the course material before or during the practical session, thereby leaving more time for practice in the computer lab with the support of the graduate teaching assistants (GTAs) and lecturer. However, if a student did not understand a concept in the videos then more support time was taken up explaining concepts at an individual level, instead of practicing and obtaining help with the exercises. In comparison, the change to a flipped classroom environment followed by the longer practical session in 2013, and the bite-sized chunks of practical exercises in 2014, was a far more effective method of programming skill development as shown by the improved examination marks and learning outcomes in Fig. 5.

Technical Considerations

All lecture and examination material is currently written in the Interactive Python (IPython) Notebook format, which has proven to be an extremely effective learning environ-

[8] During Reading Week, students have no lectures and are expected to concentrate solely on coursework.

ment as it dispensed with the cognitive load of learning an editor, integrated development environment (IDE), or the command-line interface. It allows students to write and execute their programs in among the lecture notes themselves so that everything "flows" and they have all the course material in one place. It also facilitates the running of live examples with the lecturer since the students can follow along more easily.

In order to open the IPython Notebooks, we use Python distributions that run locally on the Microsoft Windows-based lab computers. Students can choose between Anaconda (Continuum Analytics, 2015) and Enthought Canopy (Enthought Scientific Computing Solutions, 2015). The students are comfortable with this simple browser-based environment, and it also removes a considerable amount of complexity and setup time. In addition, the ease of installation and the cross-platform nature of the environment (running on Microsoft Windows, Mac OS, and Linux) means that it is simple for the students to install these distributions on their own computers for use outside of the computer lab.

Note that, since the students' Python code is combined within these "notebooks," this approach comes with the caveat that it is not possible to easily apply updates or corrections to the course material after the lecture. However, once the course material was revised throughout the years and became stable/mature enough, this was no longer a significant issue.

Comparison With Previous Years

Different learning environments were considered throughout the 5 y that the programming course was run. IDEs were discounted from the outset because of their potential for causing high cognitive load on novices in introductory programming courses (Pears et al., 2007). For the first 4 y, students accessed a central server running the Ubuntu Linux operating system in order to write and execute their programs. During the first week of term before any programming was taught, the students were given a primer on basic Linux command-line tools by the local system administrator. While knowledge of Linux commands was not being examined, students had to focus not only on learning to program but also deal with the challenge of getting used to a new learning environment. The additional level of complexity, which could be interpreted as placing extraneous load on the learners, sometimes led to frustration and demotivation, particularly when some technical difficulties could not be readily resolved by the teaching For example, in 2010, student comments included:

"Not enough work on Linux."

"Started quite badly. Most students had never touched Linux or Python."

Most students were accustomed to graphical interfaces in their day-to-day computer use as opposed to command line interfaces. The other complication was that they could not (or found it difficult to) install Linux or Python on their own computers, thereby preventing self-study outside of the computer lab.

In 2010, students wrote their programs in a console-based text editor such as Nano (Nano Development Team, 2009) and executed the program with the Python interpreter directly at the command line. Students had to continually switch between the editor and the command line, and it was clear that writing and running stand-alone program files in this way required a considerable amount of additional expertise, which placed an extraneous cognitive load on the students. To help remedy this in 2011 and 2012, students adopted the Interactive Python (IPython) tool, which simplified the process of seeing a program's results and lowered the turnaround time for debugging considerably. Additionally, in 2013, the IPython Notebook format was adopted (see above). The Git version control system was used in an attempt to apply any updates/corrections to the lecture notes as gracefully as possible, and also gave the students an insight into using version control to manage their work. However, this turned out to be somewhat counterproductive as merge conflicts frequently had to be resolved manually by the teaching assistants, which lowered the confidence the students had in the system they were using and added to the number of commands the students had to remember to download the latest revision of the lecture material. The majority of students in the class struggled to cope with both Linux and Git, which were both completely new to them.

Formative Feedback: The "Sticky-Note" System

The "sticky-note" system, promoted by the Software Carpentry organization, was used in every workshop since 2014. This almost counterculture low-tech system proved both to be incredibly powerful and popular with students for obtaining support and providing feedback that was acted on promptly. Essentially, each student is given a red and a green sticky note at the start of every lecture, and they are used in two ways.

In the first case, the sticky notes act as a status indicator when completing practical exercises; the student sticks up the red note to indicate that they need assistance, which eliminates the need to wait passively with their hand raised or similarly trying to get the attention of the lecturer or GTAs (the need to hold up their hand while waiting for a GTA to become available had been a frequent point of frustration). This in turn increases class productivity. When the student has finished a set of practical exercises, they stick up the green note to let the lecturer know the overall progress of the class.

In the second use case, the sticky notes act as "exit tickets." Students must leave one piece of positive and negative feedback on the green and red notes, respectively, before exiting the computer lab at the end of the class. This helped to identify immediately if there were issues arising in the course so they could be resolved before the next lecture, and also how the class was finding particular aspects of the course. When students struggle on exercises, they tend to complain a great deal via the anonymous sticky notes, so changes in class progress in response to feedback were more easily identifiable with this technique.

Comparison With Previous Years

A great deal of positive SOLE feedback resulted from the use of these sticky notes, as shown by the large positive

TABLE IX: Total number of graduate teaching assistants (GTAs) who helped in the programming course each year. In 2013 and 2014 there were a larger total number of GTAs assisting. In these years, the GTAs took turns and operated a rota system, to ensure that approximately 8–10 GTAs were present in each workshop.

Year	Number of GTAs
2010	4
2011	7
2012	9
2013	18
2014	14

proportion (51.4%) of support-related comments in Fig. 5c and the individual SOLE comments such as "Sticky notes work well," "WE LOVE RED AND GREEN POST IT NOTES!!!!" and "Love the post it notes." Its effectiveness is also demonstrated through the increase from a score of 0.47 in 2013 to 1.32 in 2014 for the "Feedback" criterion (see Table VII and Table VIII).

Teaching and Learning Support

It is an important, albeit often challenging, task to bring all the students up to a similar standard within a fixed period of time. The learning pace of students varies significantly for many reasons. This is particularly true for a first year course in which the students come from a diverse range of educational backgrounds. GTAs play a critical intervention role here.[9] They help to clarify concepts, support progress through exercises (identifying key difficulties), provide instant feedback on the students' work, and resolve practical computing issues. We therefore ensure that a high number of GTAs are present in each workshop. The total number of GTAs who assisted each year is given in Table IX. Typically, there are between 8 and 10 GTAs per workshop in the current version of the course, yielding a student to GTA ratio of at least approximately 10:1.

Comparison With Previous Years

While the class could be large (typically 70–90 students each year in this case), it proved critical to have a low ratio of students to GTAs so that the students did not experience long and unproductive/demoralizing waiting times for one-to-one support. This issue consistently featured strongly in student feedback; highly negative when there was a shortage of teaching assistants (typically four or fewer), high degree of satisfaction and praise for the teaching assistants when there were enough (typically 8–10 from 2013 onward). This was an issue in 2012 when, as an exception, the course was run for both first and second year students concurrently. This was necessary to transition the course from being a second-year only course to being a first-year only course in later years. Two computer labs were used at the same time, but the number of available teaching assistants had to be spread out; this resulted in comments such as "Could do with more teaching assistants" and "Need more demonstrators". However, in the other years (especially in 2013 and 2014) when just first year students were being taught, the ratio of students to GTAs was low and the teaching assistants received a great deal of praise, such as "Good demonstrators and feedback as well. Don't have to wait very long if there's a problem, and usually solved quickly" (2010), "GTAs are particularly helpful" (2013), and "Very helpful GTAs" (2014).

CONCLUSION

The successful development of an introductory computer programming course represents a significant challenge, particularly when targeting undergraduate students with little or no computing background and outside a mainstream computer science degree program. This study has outlined the influence of different methodologies on student perceptions of learning, and what appears to be a positive impact on student examination results, over the course of 5 y. Our findings reinforce the evidence in the literature that flipped classrooms and blended learning approaches are much more effective at teaching programming than traditional passive lecturing. However, we accept that our study would require further, more rigorous investigation, potentially using a quasi-experimental design, to demonstrate learning gain.

The traditional passive lecturing style that featured in the early years of the course (2010 and 2011) was more in line with what students were used to, yet was ineffective at developing the students' skill set. The video lectures that were implemented in 2012 allowed students to work through material and review it at their own pace, but a high student–GTA ratio and the lack of traditional lecturing style was a concern for many students. In 2013 when a flipped classroom approach was used, students did not feel like they were being taught, leading to a lack of confidence both in themselves and in the course. It is problematic to make a direct link between these interventions and end-of-term examination performance, but we believe this demonstrates the rewards of a learning-through-doing approach, which would benefit from further exploration in future studies. It is our belief that we have now converged on a successful teaching strategy through the use of blended learning and formative feedback, which featured in the 2014 run.

That said, many challenges surrounding the teaching of programming still remain. As class sizes grow, some degree of automated marking would be beneficial for the lecturer and teaching assistants; however, this is technically difficult to implement. While it is possible to automatically run all the students' programs and determine whether they produce the correct answer, such an answer must usually be something simple, for example, like a single integer rather than a plot. Furthermore, much like systems such as flake8 (Flake8 Development Team, 2015) that check Python coding style compliance with the PEP 8 standard (van Rossum et al., 2001), it is possible to automate checks such as "Has the student added docstrings for each function?"; however, determining whether these docstrings or comments are actually useful or not presents a much bigger challenge.

There is a large amount of pedagogical research and knowledge regarding how people learn and how best to

[9] All teaching assistants in the department receive training in pedagogical techniques and marking before they are permitted to help out in undergraduate courses.

teach programming skills. Despite this, many courses still follow the more traditional passive lecturing style, since change may be viewed as a risky process from an institutional point of view. It is therefore hoped that the findings presented in this article will inform and encourage departments and educators to reconsider their existing approach to teaching programming.

Recommendations for Adoption

The various aspects of our course's instructional design can be readily applied to other courses. For example, the sticky-note system is not just applicable to computer programming, and can indeed be applied to most courses that involve the completion of in-class exercises and require one-to-one help from the lecturer/GTAs. However, the appropriateness of the technical aspect will need to be considered carefully; we found that the IPython Notebook is an excellent environment to teach programming, but for practical exercises not involving computer code, a different practical setup may have to be designed. For example, for mechanical engineering design problems, a computer aided design package could be a more appropriate learning environment. For pure mathematics an interactive symbolic algebra package such as Maple may be desirable, although symbolic algebra can also be handled by readily importable Python modules such as SymPy (Joyner et al., 2012).

The course material is freely available under the Creative Commons Attribution 3.0 Unported (CC-BY 3.0) and MIT license and can be tailored to an individual setting. It can be downloaded from https://github.com/ggorman/Introduction-to-programming-for-geoscientists.

Outlook

Although the findings in this article provide a valuable insight into the methodologies used to teach computer programming, we acknowledge that the limitations in the type and quantity of the available data may be detrimental to the reproducibility of our findings. We therefore believe that it would be beneficial to conduct a more rigorous, quasi-experimental study in the future, with a more formal data collection plan from the outset.

Given that using a blended learning methodology throughout the last two years that the course was run featured consistently high performance, we hope that similar performance would be maintained in future classes. However, a much more rigorous study would be required to confirm this. With such a study we could also potentially investigate whether there is any correlation in, for example, the number of students with a Mathematics or Computing A-Level and examination performance. Although we don't envisage any significant effects of a change of lecturer on student performance (as long as the methodology was applied consistently and the new lecturer had the appropriate background and experience), this would be another avenue of investigation to consider, once again requiring appropriate evidence to back up this statement. On the other hand, we found that some GTAs were mentioned by name more than others and given high praise in the student feedback; it is unclear whether the unavailability of these particular GTAs during some weeks (due to the rota system) affected performance since student satisfaction is not a measure of actual learning. However, this is something that will be looked at more closely in later work.

For the purposes of demonstrating how student examination responses were graded, and to illustrate that the responses were of a consistent quality throughout the years, a revised study would also record anonymized examples during the data collection phase. We plan to consider both high- and low-quality responses for comparison.

Further improvements in the software supporting the students' learning (in this case, the IPython Notebook) may potentially affect student performance. For example, the version used in 2014 was prone to crashing when infinite "while" loops occurred. Students were able to seek assistance swiftly from the GTAs when such technical difficulties occurred, but sometimes the difficulties caused frustration, especially if the IPython Notebook had to be restarted and part of the student's work was lost. If such software issues were to be resolved, this may boost the students' satisfaction and confidence in the course. However, once again, a more rigorous study would be needed to show whether this positively affects student performance, since improved student satisfaction does not necessarily yield learning gain.

Other courses that form part of the geoscience curriculum would also be considered, as it could act as a way of measuring how well information is retained from the introductory programming course. For example, in 2014 another course took place the following term that required the use of Python. More specifically, it focused on the application of Python's numerical and scientific libraries NumPy and SciPy to perform statistical analysis of geoscientific data. This has since been replaced (for the 2015 class, outside the timeframe of the current study) by a course that concerns the Python implementation of numerical methods to solve systems of equations modelling geophysical phenomena. Both of these courses aim to further the students' education in computational geoscience.

Acknowledgments

We would like to thank all the undergraduate geoscience students that we have taught over the years in the Department of Earth Science and Engineering at Imperial College London for providing constructive feedback on the course as it evolved. We are also extremely grateful for all the hard work and support received from the GTAs over all the years teaching this course. Special thanks go to Hans Petter Langtangen for giving permission to use content from his book and course notes, and to Greg Wilson of the Software Carpentry organization for training and useful discussions on evidence-based pedagogy for programming. The feedback from the anonymous reviewers was warmly received and greatly improved this article.

This work was carried out under the approval from the Education Ethics Review Process (EERP) at Imperial College London.

REFERENCES

Aagaard, B.T., Knepley, M.G., and Williams, C.A. (2013). A domain decomposition approach to implementing fault slip in finite-element models of quasi-static and dynamic crustal deformation. *Journal of Geophysical Research: Solid Earth,* 118(6):3059–3079. DOI: 10.1002/jgrb.50217.

Aagaard, B., Knepley, M., and Williams, C. (2016). PyLith User Manual, Version 2.1.2. University of California, Davis. Available at https://geodynamics.org/cig/software/pylith/ (accessed 30 June 2016).

Beard, R.M., and Hartley, J. 1984. Teaching and learning in higher education. London, UK: Harper & Row.

Beyreuther, M., Barsch, R., Krischer, L., Megies, T., Behr, Y., and Wassermann, J. 2010. ObsPy: A Python toolbox for seismology. *Seismological Research Letters,* 81(3):530–533.

Biddle, R., and Tempero, E. 1998. Java pitfalls for beginners. *ACM SIGCSE Bulletin,* 30(2):48–52.

Bonk, C.J., and Graham, C.R. 2006. The handbook of blended learning: Global perspectives, local designs. San Francisco, CA: Pfeiffer.

Böszörményi, L. 1998. Why Java is not my favorite first-course language. *Software—Concepts & Tools,* 19(3):141–145.

Boyle, T., Bradley, C., Chalk, P., Jones, R., and Pickard, P. 2003. Using blended learning to improve student success rates in learning to program. *Journal of Educational Media,* 28(2–3):165–178.

Churcher, N., and Tempero, E. 1998. Java as a first programming language. *In* Purvis, M., Cranefield, S., MacDonald, S., Proceedings of the 1998 International Education & Practice. Las Alamitos, CA: IEEE Computer Society, p. 390–393.

Clark, D., MacNish, C., and Royle, G.F. 1998. Java as a teaching language—Opportunities, pitfalls, and solutions. *In* Carrington, D.A., Proceedings of the 3rd Annual Australasian Conference on Computer Science Education. New York: ACM, p. 173–179.

Close, R., Kopec, D., and Aman, J. 2000. CS1: Perspectives on programming languages and the breadth-first approach. *Journal of Computing Sciences in Colleges,* 15(5):228–234.

Continuum Analytics. 2015. Anaconda Scientific Python Distribution. Available at https://store.continuum.io/cshop/anaconda/ (accessed 16 May 2015).

Donaldson, T. 2003. Python as a first programming language for everyone. *In* Chase, G., Franklin, P., Gee, R., Lu, F., Niscak, F., Stephens, S. Proceedings of the Western Canadian Conference on Computing Education (WCCCE) 2003. North Island College, Courtenay, BC: Vancouver, BC. Available at: https://www.cs.ubc.ca/wccce/Program03/papers/Toby.html

Dowling, C., Godfrey, J.M., and Gyles, N. 2003. Do hybrid flexible delivery teaching methods improve accounting students' learning outcomes? *Accounting Education,* 12(4):373–391.

Enthought Scientific Computing Solutions. 2015. Enthought Canopy: Python distribution and integrated analysis environment. Available at https://www.enthought.com/products/canopy/ (accessed 16 May 2015).

Environmental Systems Research Institute. 2015. ArcGIS Project. Available at http://www.arcgis.com (accessed 16 May 2015).

Fangohr, H. 2004. A comparison of C, MATLAB, and Python as teaching languages in engineering. *In* Bubak, M., van Albada, G.D., Sloot, P.M.A., and Dongarra, J., eds. Computational Science–ICCS 2004. Berlin, Germany: Springer. 3039:1210–1217.

Flake8 Development Team. 2015. Flake8 project documentation. Available at http://flake8.readthedocs.org (accessed 16 May 2015).

Friesen, N. 2012. Defining blended learning. Available at http://learningspaces.org/papers/Defining_Blended_Learning_NF.pdf (accessed 21 June 2016).

GRASS Development Team. 2015. Geographic Resources Analysis Support System (GRASS GIS) Software. USA, Open Source Geospatial Foundation. Available at https://grass.osgeo.org (accessed 21 June 2016).

Gruber, J. 2004. The Markdown language. Available at http://daringfireball.net/projects/markdown (accessed 21 June 2016).

Guzdial, M. 2015. What's the best way to teach computer science to beginners? *Communications of the ACM,* 58(2):12–13.

Hattie, J. 2008. Visible learning: A synthesis of over 800 meta-analyses relating to achievement. Abingdon-on-Thames, UK: Routledge.

Hunt, A., and Thomas, D. 1999. The pragmatic programmer: From journeyman to master. Boston, MA: Addison-Wesley.

Isaacs, G. 1994. Lecturing practices and note-taking purposes. *Studies in Higher Education,* 19(2):203–216.

Jacobs, C.T., and Piggott, M.D. 2015. Firedrake-Fluids v0.1: Numerical modelling of shallow water flows using an automated solution framework. *Geoscientific Model Development,* 8(3):533–547.

Joyner, D., Čertík, O., Meurer, A., and Granger, B.E. 2012. Open source computer algebra systems: SymPy. *ACM Communications in Computer Algebra*, 45(3/4):225–234.

Langtangen, H.P. 2009. A primer on scientific programming with Python. Berlin, Germany: Springer.

Lin, J.W.-B. 2012. Why Python is the next wave in Earth sciences computing. *Bulletin of the American Meteorological Society*, 93(12):1823–1824.

Middendorf, J., and Kalish, A. 1996. The "change-up" in lectures. *The National Teaching & Learning Forum*, 5(2):1–11.

Mody, R.P. 1991. C in education and software engineering. *ACM SIGCSE Bulletin,* 23(3):45–56.

Nano Development Team. 2009. GNU nano project. Available at http://www.nano-editor.org/ (accessed 16 May 2015).

Palumbo, D.B. 1990. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1):65–89.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin* 39(4):204–223.

Pereira, J., Pleguezuelos, E., Merí, A., Molina-Ros, A., Molina-Tomás, M.C., and Masdeu, C. 2007. Effectiveness of using blended learning strategies for teaching and learning human anatomy. *Medical Education,* 41(2):189–195.

Pérez, F., and Granger, B.E. 2007. IPython: A system for interactive scientific computing. *Computing in Science and Engineering,* 9(3):21–29.

QGIS Development Team. 2009. QGIS Geographic Information System, Open Source Geospatial Foundation. Available at: http://qgis.osgeo.org.

Ramsden, P. 1992. Learning to teach in higher education. Hove, UK: Psychology Press.

Rathgeber, F., Ham, D.A., Mitchell, L., Lange, M., Luporini, F., McRae, A.T.T., Bercea, G.-T., Markall, G.R., and Kelly, P.H.J. 2015. Firedrake: Automating the finite element method by composing abstractions. Submitted to ACM TOMS.

Robins, A., Rountree, J., and Rountree, N. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education,* 13(2):137–172.

Stefik, A., and Siebert, S. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE).* 13(4):1–19.

TIOBE Software. 2015. TIOBE programming community index. Available at www.tiobe.com/index.php/tiobe_index (accessed 21 June 2016).

van Rossum, G., Warsaw, B., and Coghlan, N. 2001. PEP 0008—Style guide for Python code. Available at https://www.python.org/dev/peps/pep-0008/ (accessed 16 May 2015).

Wilson, G. 2006. Software Carpentry: Getting scientists to write better code by making them more productive. *Computing in Science & Engineering,* 8(6):66–69.

Wilson, G. 2014. Software Carpentry: Lessons learned. *F1000Research,* 3(62):1–24.

Winslow, L.E. 1996. Programming pedagogy—A psychological overview. *ACM SIGCSE Bulletin,* 28(3):17–22.

Yapici, İ.Ü., and Akbayin, H. 2012. The effect of blended learning model on high school students' biology achievement and on their attitudes towards the Internet. *Turkish Online Journal of Educational Technology,* 11(2):228–237.

Yuan, L., and Powell, S. 2013. MOOCs and open education: Implications for higher education [White paper]. Bolton, UK: Centre for Educational Technology, Interoperability and Standards, University of Bolton.

APPENDIX A. SOLE Lecturer and Module Scoring Criteria

Responses to the criteria below were requested as part of SOLE with respect to both the module and the lecturer. Note that in 2010, 2011, and 2012, the available responses were Very Good, Good, Satisfactory, Poor, Very Poor, No response, whereas in 2013 and 2014 the available responses were Definitely Agree, Mostly Agree, Neither Agree or Disagree, Mostly Disagree, Definitely Disagree, Not applicable. In addition to the criteria, text boxes were also provided so students could provide additional constructive feedback.

Lecturer Score Criteria (2010, 2011)

1. The structure and delivery of the lectures
2. The explanation of concepts given by the lecturer
3. The approachability of the lecturer
4. The interest and enthusiasm generated by the lecturer

Module Score Criteria (2010, 2011)

1. The support materials available for this module (e.g., handouts, blackboard/web pages, problem sheets, and/or notes on the board)
2. The organization of the module
3. The structure and delivery of the lectures
4. The explanation of concepts given by the lecturer
5. The approachability of the lecturer
6. The interest and enthusiasm generated by the lecturer

Lecturer Score Criteria (2012)

1. The structure and delivery of the teaching sessions

Module Score Criteria (2012)

1. The structure and delivery of the teaching sessions
2. The content of this module

Lecturer Score Criteria (2013, 2014)

1. The lecturer explained the material well
2. The lecturer generated interest and enthusiasm
3. The lecturer was approachable
4. Overall, I am satisfied with this lecturer

Module Score Criteria (2013, 2014)

1. The content of the module is well structured
2. The content of the module is intellectually stimulating
3. I have received helpful feedback on my work
4. Overall, I am satisfied with the quality of the module