Invited viewpoint: teaching programming to students in physical sciences and engineering

Lloyd Cawthorne[1,*]

[1] *Department of Physics and Astronomy, University of Manchester, M13 9PL Manchester, UK*

Received: 12 May 2021
Accepted: 18 July 2021
Published online:
5 August 2021

ABSTRACT

Computer programming is a key component of any physical science or engineering degree and is a skill sought by employers. Coding can be very appealing to these students as it is logical and another setting where they can solve problems. However, many students can often be reluctant to engage with the material as it might not interest them or they might not see how it applies to their wider study. Here, I present lessons I have learned and recommendations to increase participation in programming courses for students majoring in the physical sciences or engineering. The discussion and examples are taken from my second-year core undergraduate physics module, Introduction to Programming for Physicists, taught at The University of Manchester, UK. Teaching this course, I have developed successful solutions that can be applied to undergraduate STEM courses.

Introduction: why is teaching programming difficult?

Why is teaching programming difficult? More specifically, why is it difficult for science and engineering students in fields other than computer science? We, academics in these fields, recognise it as an indispensable tool in our day-to-day work, whilst students can often see it as a chore. For instance, I recall one student saying to me: "*I get that programming is useful, but I came here to study physics*"; they did not see programming as a means to study for their degree. This disconnect is often present early in a course whilst we introduce programming and the language we are working in before we can apply it to meaningful situations. In other words, `print('Hello World!')` does not help me solve the Schrödinger equation(!)

In this viewpoint, I will first outline more formally why there is a difficulty teaching programming and present recommendations to remedy this. These are justified in Sect. 2 which is split into four: mastery of syntax and debugging error messages, context to learning activities, diversity and careers, and teaching and assessing programming style. I then comment on the details on how you might deliver an introductory programming course before

Handling Editor: Christopher Blanford.

Address correspondence to E-mail: lloyd.cawthorne@manchester.ac.uk

summarising. A glossary of terms used in this viewpoint can be found in Table 1. Throughout we have added reflections on teaching this subject during the global pandemic. I encourage you, the reader, to reflect if any of these recommendations would benefit your course. In some cases they might not; they might not even be logistically possible.

A typical introductory programming course will assess students by having them complete coursework. So students are assessed on their ability to apply programming to solve problems. They are not assessed on their surface-level knowledge of programming. Therefore, for a student to do well, they must quickly apply abstract ideas to perform tasks [1]. To do this, there are a number of non-trivial skills they need to gain fluency in: they need to master the syntax of the language they are working in, they must recognise common error messages and how to fix them, they need to be able to break down a larger task into smaller blocks which can be performed by code.

Learning the above skills might not naturally be appealing to, say, a physics student who has dreams of researching matter-antimatter asymmetry. Alone these skills are not exciting, but the problems that they can solve are. As course leaders, we must set the expectations of students such that they recognise the importance of programming in the wider context of the field.

Another aspect we must bear in mind is that there is significant variation in ability. Some students will enter their degree with industrial experience in programming whilst the majority are novices in this field. As an instructor, it is non-trivial to manage this and create activities suitable for all. Our prime focus must lie on the novices, whilst we direct the advanced students to further reading where possible.

Finally, we must recognise that the majority of students we are teaching will not enter research careers and are likely not minded to do the extra practice required to master the material. This is a reflection of academic, or engaged, students versus the nonacademic, or non-engaged, students (there is ample discussion on these two groups in Biggs and Tang referred to as "Susans" and "Roberts" [2]). In programming, however, it is expected that students will emerge from their degree with the ability to use computing to support or drive their work. (This is indeed part of the accreditation for a physics degree in the UK [3].) Furthermore, we, educators at higher education institutions, have a sizeable role in addressing the digital skills crisis [4–6]. To that end, following this introductory course, we expect students to update their CV to include skills in programming. For the majority of students, this will be their only formal teaching of programming.

A summary of my suggestions is found in Fig. 1. Some of these are common practice, though, if you are preparing a new course, be mindful that some require significant planning.

Solutions

Mastery of syntax and debugging error messages

Before any worthwhile coding activities can be done, students must gain fluency in the language they are working in. This alone can be daunting to novices as they are unfamiliar with how arbitrary syntax can be [7]. Once they have gained confidence they then need time and activities to work in this language, therefore, use only one programming language in introductory courses [8]. The language taught should have minimal distracting syntax.

Table 1 Glossary of terms in this viewpoint

Control flow	The order in which individual statements or definitions are executed.
Currying	In computer science, defining a function within a function.
IDE	Integrated development environment, software that supports code development.
Jupyter	A community that develops open-source software to support interactive data science and scientific computing.
Linter	A tool that flags programming bugs, stylistic errors and suspicious constructs.
Notebook	An interface used for literate programming; mixes code results, graphics, text, and more.
Scope	The visibility of a variable or method to different parts of a programme.
Spyder	An open-source IDE for scientific programming in Python.

Figure 1 Tips for teaching programming to science and engineering students not on a computer science course.

- Use only one programming language in introductory courses.
- Choose a language like Python that has minimal distracting syntax.
- Create early learning activities focused on debugging.
- Present many examples in a context relevant to the students' field.
- Create tasks and assignments that are directly transferable to other areas of study.
- When demonstrating code, provide students with prewritten code that they can edit.
- Formalise a programming style and support this with a style guide that includes examples.
- Formalise how style is marked.
- Include some open-ended assessment.
- Showcase historical figures to change impressions of what a computer scientist *should* look like.
- Highlight career paths and increased employability that comes from coding ability.
- Employ one teaching assistant for every 15–20 students in a computer-based laboratory session.
- Give teaching assistants enough time to review the weekly content and prepare for the laboratory session.
- Create summaries of each week's content and make them accessible to teaching assistants.

With these aspects in mind, we will give examples in Python, a language which has a design philosophy that emphasises code readability. Furthermore, the popularity of Python has surged in both industry and research [9, 10]. Despite being a high-level language, a great deal of work has been done to enable it to perform tasks that were previously only achievable in C (or a similar lower-level language) [11]. As an instructor, we can be confident that teaching Python over Java, C or C++ will not hinder our students later in their career.

Together with the syntax of the language, there are a handful of basic concepts that need to be covered early on: variable types, control flow and variable scope. The concept of scope is the most abstract and hence requires highlighting in many different examples. Limited content on how the machine interprets code and handles data needs to be taught in an introductory course. Working in Python, this could be restricted to how different variables are represented in binary, which could be expanded to rounding and overflow errors.

To teach the basic concepts above, we suggest diverging slightly from the commonly held approach of *programming is best learnt by programming*. We do not disagree with this ethos, just suggest a different form of learning activity should take place at the start which is more suited to non-specialist (or less motivated) students. That is to create learning activities focused on debugging errors. I have done this by creating quizzes. This was chosen to automate the marking process and facilitate remote learning. This was driven by increasing class size (300+ students) to enable support to be concentrated for those most in need, though it is also well suited for delivery amidst the pandemic. I deliver the quizzes through regular tests on our virtual learning environment (VLE): BlackBoard. These are weekly to begin with, so students can build their confidence, then taper so students can concentrate on their assignments. The quiz questions themselves typically consist of a variety of multiple choice, multiple answer and matching questions. The latter, where students need to correctly pair question and answer items, is very versatile.

For all of the quizzes, we provide two versions to students: a practice version and an assessed version. Students are allowed unlimited attempts in the practice version so that they can gain familiarity with the questions and the specific answers they require, and also get automated feedback. The assessed version can only be done once and there are question pools to reduce collusion. The possibility of collusion cannot be removed entirely in these tasks, a point that must be accepted by the course leader. The difficulty of the assessed version must be less than or equal to the practice test.

An example of a quiz question is presented in Fig. 2, where students need to select the correct order of the three lines of code, thus reinforcing the concept of control flow. This question encourages novices to check their result by running the code. Later, the question can be expanded upon to decide the order in which three functions should be called. Another example is given in the supporting information (`week_2_doppler_practice.py`) where students need to debug basic syntax errors. I present this problem to students soon after they have been introduced to functions. I intend for them to use the IDE's error flagging to solve it. There are a number of aspects to this question that support novices. Firstly, it simply reinforces the syntax associated with functions. Secondly, it does this in a way where the code is broken in a specific way; we have concentrated what we want them to focus on in this particular task. Thirdly, as the code is given as broken, the student cannot assign any blame to themselves or question their ability as it was not broken by them. Finally, it simply introduces a different example of using functions, namely a function being called in the argument of another. Here, the students are instructed how many errors they need to fix to assist them.

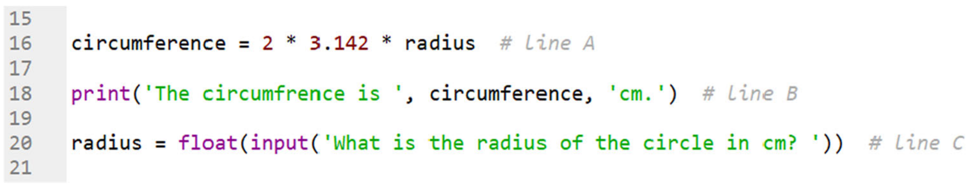
If the student had to write the code from scratch—the traditional approach—then they would likely make these same errors but now get frustrated at their own ability, disengaging them from the topic. There would also be instances of them generating unseen errors which they would struggle to debug at this stage.

From this point in my lectures, I start introducing questions where the IDE will *not* flag the errors. This change requires students to test the code by including useful print statements or running a debugger. An example of such a problem is presented in given in the supporting information, `week_3_tribonacci_practice.py`, where students need to debug code that finds members of the tribonacci sequence (a variant of the Fibonacci sequence). The errors are such that students reinforce their knowledge of syntax and start to build wider problem-solving skills for when code invariably does not work as intended. Here, students complete the question by matching inputs to given possible outputs. The input numbers are chosen such that there are a couple they can check by hand and a couple where they will need the code to run correctly. These also include edge cases. Note we do not tell students how many errors they need to find here to encourage them to be confident with their work before continuing, i.e. checking their answer is correct for small numbers.

Alongside the quizzes, we also provide a simple introductory assignment for them to write from scratch. We give week-by-week guidance on what they should be able to complete to ensure that they are applying the weekly content appropriately. As stated earlier, the traditional approach of *learn by doing* is not removed; however it is no longer the primary focus whilst students build their core skills.

Context to learning activities

Students must apply any new skill in new settings outside of the course to embed its learning and for it to develop after their formal instruction is complete. It is unlikely they will be doing this during the course; however, material can still be provided to try and encourage this. Following the course, if students



```
circumference = 2 * 3.142 * radius  # Line A

print('The circumfrence is ', circumference, 'cm.')  # Line B

radius = float(input('What is the radius of the circle in cm? '))  # Line C
```

Figure 2 Example multiple choice quiz question where students need to select the correct order of the code lines from different permutations of *ABC*. It is given as part of a BlackBoard quiz.

are not continuing to apply programming in their degree, then the course has failed its purpose. It is naïve for us to assume that students will naturally see how programming can translate to their wider studies. Instead, we must make that link explicit. To do this, one should present many examples in a context relevant to the field and create tasks and assignments that are directly transferable to other areas of study. An example of how this has been done for a first year course as part of a material science degree has been shared by Prof. Quinta da Fonesca and Dr. Race; this can be found in [12]. Early on, making this link is as simple as presenting code that performs short calculations relevant to the field, see Fig. 2. Later these need to be more meaningful and useful. For instance, in a parallel course, students are taught about Fourier series approximations, well we can show them how these can be visualised with code. We can implement this in a quiz question where the emphasis is on them exploring plotting options rather than having to debug or write code. Students are presented with a plot, see Fig. 3; some skeleton code, (see `week_8_intro_to_plotting.py` in the supporting information), and various plot options. So, in this quiz question, we are showcasing a variety of plotting options and also demonstrating wider applicability in other courses. We then hope students recognise that they could amend this code, or write their own, to study similar problems in future.

Throughout the students' degree, they will be exposed to other examples of programming from different authors. Undoubtedly these will be written in different styles and perhaps even different languages. This issue will vary depending on discipline

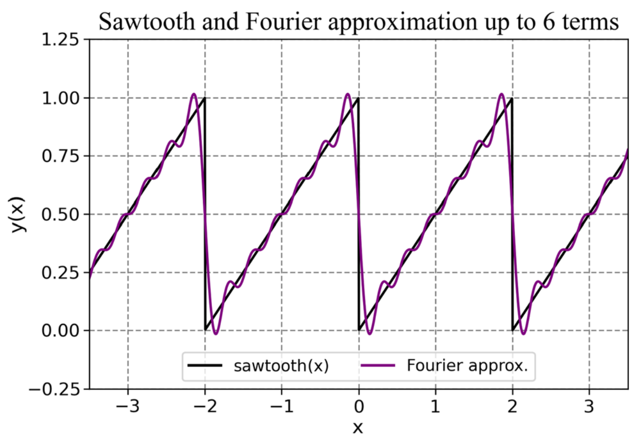


Figure 3 Sixth-order Fourier approximation of the sawtooth function. Demonstrates many useful plotting options.

and therefore creating uniformity is beyond the scope of this article. However, being exposed to code in these settings will strengthen the idea that programming is a core skill in science and engineering.

These examples, seen in quizzes and lectures, may not offer enough incentive for a student to continue practising after the course. If we want students to use programming post-instruction, then one must think of what skills are most immediately valuable to them; we must constructively align assessment to the programme-level learning outcomes. We, academics, might write code for a variety of purposes. These range from long, well documented and versatile scripts that will be seen by large collaborations; to short and rough calculations or checks that are for our use only. There could be debate about which extreme to concentrate on. I feel there is more merit in teaching the former as it assists marking and supports the student in any project work. Furthermore, a recent graduate that can provide polished examples would stand out to a potential employer.

In the context of a science and engineering degree, their programming skills are directly applicable to their laboratory work where they are organising, filtering, manipulating, visualising and fitting data. So, we give them a final assignment where they have to validate and combine data sets to perform a minimised χ^2 fit on two parameters. The data sets include trivial errors along with some outliers that require some statistical justification to remove. If students implement these skills in their laboratory work they obtain high marks, and they are told this repeatedly. That final aspect of communication is key. Applying these skills in the laboratory, students will encounter new problems, or want the code to produce something slightly different, and they will have to figure this out themselves. This is the approach that all of us use when coding. For our course in Manchester, this desired outcome was witnessed by the laboratory tutor.

> "[T]here was some very impressive analyses by a good number of the students in the second semester this year utilising nonlinear curve fitting that they must have learned from your course which was great to see (and a good way of earning the top mark for analysis)".
> (2nd year lab tutor)

Furthermore, these tasks that are aligned with their wider studies can be framed with contexts that could enthuse the students. For instance, in my department, we have used examples looking at determining the mass of exoplanets, finding the decay constants of exotic nuclei, finding the width of a material through quantum tunnelling, etc. These can help give the project some meaning and application. This guise could also be what is needed to motivate some students. For many, this will be the first research-like project they have done and they will become engrossed in it. In this case, we might encounter the opposite problem where students are spending more time than it is worth on their project. To counter this, we could give guidance on how much time they should be devoting to it and recognising that the longer they work on it, the fewer additional marks they gain.

Diversity and careers

Like many fields in STEM, computer science lacks engagement from underrepresented groups which in turn fosters a damaging preconception about what a computer scientist should *look like* [13, 14]. These issues are societal and it will take far more than an introductory course to address them. However, instructors can be mindful of these problems in our delivery. Specifically, we can showcase historical figures in computer science and computational science and highlight potential career paths.

Spending a few minutes each week on these topics can have a significant positive effect on students' perceptions of the field. Moreover, it actively counters any erroneous prejudices about *who can be a programmer*. There are many figures worthy of our attention. For instance: Ada Lovelace, Douglas Hartree, Alan Turing, Gladys West, Donald Knuth, Grace Hopper, etc. Here, it is worth commenting further about the person and not just their contributions. It might be that a student is indifferent to Lovelace's development of the first programme, but the complex relationship she had with her mother could resonate. Or discussing the institutional homophobia that Turing was subjected to. Or, more light-heartedly, Knuth's eccentricity to finish The Art of Computer Programming.

When highlighting these people, we naturally discuss a broad range of applications demonstrating the versatility of the field. Students can then recognise potential career paths which for some is needed to boost motivation. As an example, this year I discussed the protein folding problem and how it was solved with AI [15]. Students were then surprised to learn that working on projects like this is accessible to them in future.

Below is some written feedback from students when asked about what they found most helpful and what helped them feel part of a learning community.

> "I especially liked the segment where he told us about prominent people in computer science/physics".
> (Anon.)

> "I loved the "5 minutes of history" part of the synchronous sessions".
> (Anon.)

These recommendations are not restricted to programming and should be common place throughout a STEM degree as stated by the Institute of Physics [16].

Teaching and assessing programming style

In a course that concentrates on programming (rather than the application of it in a topic), it is not uncommon to include some discussion and some assessment on coding style. This forces students to reflect on what they have written and encourages good habits (or discourages bad ones) [17, 18]. Additionally, as a demonstrator or marker, having code written in a clear style makes these tasks much easier. As articulated by van Rossum:

> "Code is read much more often than it is written".
> (Guido van Rossum - Author of Python)

We have implemented this by marking style on par with output. This approach might not suit all courses, but we feel this is an important aspect of coding and should be considered as part of the intended learning outcomes. To implement this, one should formalise the style, provide a style guide that includes examples, and formalise how this is marked.

The above suggestions are not revolutionary. However, with large class sizes, you should expect to see great variation in approach and need to consider carefully what deserves merit and what should be penalised. Furthermore, students will need to

- Useful variable & function names.
- Code structure.
- Useful function docstrings.
- Appropriate use of functions.
- Negatively mark use of `global`.
- How files are opened and closed.
- How data is filtered and removed.
- How plots are created.
- How versatile the code is.

Figure 4 Suggested points of style to grade in students' code.

understand their mark if they have done something ill-advised. For instance, if you come across an instance of currying (i.e. defining a function within a function), is this a student that has expertise in functional programming or is this a novice who has not understood scope? It is far easier to mark if these decisions have already been made by the course leader and are communicated to students in the style guide. Aspects we have marked against are listed in Fig. 4. In practice, it is much easier to deduct marks for failing to adhere to the style, rather than awarding marks for including particular aspects. This also caters for the great variety in approach from the students.

With numerous teaching assistants, consistency in marking can be difficult to manage. Clear guidance is necessary here. For instance, when marking variable and function names, we expect them to be meaningfull, in full English and written in `snake_case`. A marking method that suits this is discussed in Sect. 3.4.

Another side to style is general formatting of code. We have adopted and mark against PEP8 standards [19]. This is accepted as the industry standard and many styles used in industry are deviations from this. Asking students to adhere to this requires some justification as they are unhappy when marked down for whitespace, though there are tools that can help with this. Again, these tedious aspects need to be laboured in the style guide. To help students recognise the importance, polls and quiz questions can be produced where students select the code that looks best. To mark this, a linter can be used that checks the code for PEP8 compliance and outputs a numerical score. This has had the unexpected benefit of encouraging students to reflect further on their code; especially with refactor warnings.

Including style as part of the assessment also poses a challenge for students who have prior programming experience. As many are self-taught, they might have picked up some bad habits which take time to adjust. To mitigate any criticism from these students, the reasons why we are assessing style need to be clear. The statement *"If you were working in industry you would be asked to adhere to the house style."* is often suffice. If they dislike the linter, explain how this creates consistency in marking which the majority of students will favour.

Another criteria to mark against which can help discriminate high-achieving students is code versatility. This could consist of testing edge cases or similar data files. I have often found that students might programme the validation and fitting procedure to be versatile, but the plotting routine is not.

Open-ended assessment

Open-ended assessments are often used in programming courses. This type of assessment can test higher-levels of understanding and problem-solving, and promotes life-long learning [2]. This assessment method is more common in advanced courses where there are a limited number of students. However, there is still space to have some open-ended element in an introductory course, which will test students with prior experience in programming. Care must be taken to ensure the task remains accessible to students that are engaged with the course, but are new to programming. Guidance should be given to students to clarify how much work is expected and some suggested avenues to explore. We have awarded marks for a range of additions varying in difficulty. For instance, we have awarded marks for simply modifying the plot settings or creating additional plots. Other possible additions include estimating the fit starting values, how data are validated and removed, using sophisticated programming techniques that are only mentioned in the course. We recommend placing caps on how much can be awarded for each of these additional features; full-marks here should have a variety of different aspects done well. Furthermore, these should not compensate for failings in other areas; no matter how beautiful a plot is, it does not make up for the wrong answer.

We have allocated 25% of the marks to open-ended tasks. We would expect 4 or 5 additions done well to achieve full marks for this category, and we tell students this.

Delivery details

Choice of coding environment

The choice of whether to work in a traditional Integrated Development Environment (IDE) or Notebook is key and dictates both how you deliver content and how students interact with it. There are also issues guaranteeing version control if students are expected to work on their own machine. The technological landscape here is rapidly evolving and so what we discuss might soon be outdated.

There are many editors to work with, here we will comment on Jupyter Notebooks and Spyder. Both are available from the Anaconda distribution [20], which also includes many popular libraries in scientific computing (NumPy, Pandas, matplotlib, SciPy, etc.). A direct comparison between notebooks and IDEs is presented in Table 2.

Another important aspect to consider is that Jupyter Notebooks can be created, edited and run through online services whilst Spyder must be local. The latter can create difficulties when assigning versions of truth across libraries. However, minimal technical knowledge is required to remedy this using the anaconda navigator. Of course, there will be difficulties with students working on inappropriate personal equipment (tablets, chromebooks, etc.).

If you plan to integrate PEP8 into your marking, then the linter PyLint comes as standard with Spyder. If you want to include whitespace in your checks, then you should downgrade to version 2.5.3 as this is absent in later releases.

One must also consider where future courses are headed. For instance, if there is a subsequent programming course which is taught in a different style (object orientated, functional, procedural, etc.), what students experience in their introductory course must form a foundation for the advanced course. The transition from IPython notebooks to C++ in an IDE is markedly more challenging than if python had been taught using an IDE.

I teach using Spyder. There are two main reasons for this: the accessibility of the variable explorer and debugger, and inbuilt linter. As the variable explorer is visible by default, it allows the student to see what the code is doing without having to delve deeper or specifically request it. This also supports demonstrators when asked to assist with fixing code. Similarly, debugging is simple to initiate and couples well with the variable explorer.

Much like how we advise against teaching multiple languages, we would not recommend teaching with different IDEs. Of course, students are not restricted in their choice in environment and a handful will explore different options.

Table 2 Key differences between notebooks and integrated development environments (IDEs)

Notebook	IDE
A notebook is organised into cells. These can take the form of code, markdown and heading. This allows for thorough documentation whilst clearly distinguishing between what is and what is not code.	An IDE will primarily consist of a text editor. Spyder also comes with an IPython console visible by default and some useful panes such as files, variable explorer and help.
The presentation is *clean* with minimal buttons that are rarely used.	There are many parts to the presentation with different windows and toolbars.
When giving example code, the intention would be that a student would read through the notebook and (should) run each code snippet as they progress.	The code needs to be well commented with useful variable/function names for the student to follow. Otherwise, the students will require some accompanying information describing the code.
When writing code traditionally, one had to make use of various print statements to debug and perform checks. However, recently there have been improvements that include a debugger and variable explorer.	A debugger and variable explorer are key features in an IDE and are visible by default in Spyder.
	Spyder also comes with a linter, PyLint, by default.

The choice of which environment to work in and what software to use is ultimately yours. Working with what you are most familiar with is easiest, but do consider potential issues raised here.

Student support

Engaging with teaching assistants

Not all students will learn independently and so support must be provided. Ideally, a computing-tutor could be assigned to a small group of students who would have informal discussions and provide formative feedback on a regular basis. However, with large cohorts, in a non-computer science department, it is unlikely the number of specialised personnel will be available to deliver this.

More realistically, a large computer cluster is booked with help from graduate teaching assistants (GTAs). If attendance is optional, it might be that these are relatively quiet with demand increasing as deadlines approach. It is here when demonstrator support is vital, furthermore if assignments need to be marked individually with written feedback then aim to have a demonstrator to student ratio between 1:15 and 1:20.

Demonstrators require guidance to work effectively. If they are new to the course, they will need reminding that the students are not fluent in the topic and may have anxieties. What is more difficult as a course leader is ensuring the methods demonstrators are suggesting have been covered. For example, demonstrator might recommend using pandas to read-in data files when `np.genfromtxt` has been taught explicitly. Pandas is indeed a very powerful library and definitely a tool a data scientist should know, however, for a novice, too many alternatives is confusing; this is the message that needs to be given to the demonstrator. To help avoid issues like these, allow for demonstrators to have time to prepare/review the weekly content and create summaries of the content delivered each week which are accessible to demonstrators.

There should also be provision to respond to queries outside of support hours. Discussion boards enable all questions to be visible and allow follow-up discussion. Here, we opted for a Piazza discussion board. This was very active—especially close to deadlines!—thanks to facets of Piazza that encourage discussion and create an inclusive environment [21]. Students are able to post anonymously, which encourages participation from under-represented groups [22]. It also allows students to respond to each other and the endorsement of one another's questions and responses, thus improving online discussion and community [23]. Furthermore, Piazza will format code-snippets in the thread and one can run them in the discussion board.

Providing support remotely

As a consequence of the pandemic, we were compelled to move all teaching online. In order to remain consistent with other courses and experimental laboratory, we used Microsoft Teams to provide an online support space. Other options we considered were Zoom and Slack. Like the in-person laboratory, this was meant to provide drop-in conceptual or technical support. Microsoft Teams worked well and we will continue to use it if remote learning must continue. Teams allows the creation of channels which can be used for particular days or topics. Initiating a call is simple and creates a chat box between the users. This chat box remains after the call which is ideal for quick follow-ups. Furthermore, you can send code snippets through the chat.

In practice, a student would post in the relevant channel that they needed help, and a demonstrator would reply saying they would initiate a call. The written reply is key so other demonstrators know the query is being dealt with, especially if there is a sudden flurry of posts.

In this guise, the demonstrator loses the ability to *walk the floor*. This loss cannot be understated as it provides opportunity to assist students before they recognise they need help, or if they are shy to raise a hand. Moreover, it allows for informal feedback and allows the course leader to see how well the content is being understood.

Providing support remotely did also yield some unexpected benefits. Firstly, demonstrators could have long discussions with students uninterrupted. Secondly, and most importantly, students could not see each other working. This removed the scenario where a novice is sat next to an expert and sees them complete tasks in a few minutes which have taken them significantly longer. This somewhat removes an experience that can trigger aspects of imposter syndrome.

Code demonstration

Often blocks of code or even entire programmes will need explanation. For instance, debugging the code to see how the variables update, or highlighting changes in parameters to obtain different results. This is where we would also discuss how to test code. In an introductory course, we do not have time to do anything further than print statements and debugging.

Traditionally, these demonstrations will have been delivered live. Now, somewhat forced by the pandemic, these explanations can be prerecorded allowing the course to follow a blended approach. In short, this involves content that students can access in their own time *asynchronously* that supports live *synchronous* sessions. Blended learning is known to be a successful approach to teach programming and our experiences agree [24, 25].

One item to consider here is if coding live is beneficial. The intention being that students would naturally see us make mistakes, how we find them and how we fix them. Delivering content this way in person is extremely challenging. One has to write code within a time-slot, whilst explaining what they are doing and monitoring the audience. Invariably when something does go wrong, explaining how you solve that issue is difficult as you might not immediately know what it is or how long it will take to fix. Delivering an entire lecture in this way will consist of you huddled over the computer in the corner with eyes fixed on the screen: this will be boring. We learnt this the hard way. A more elegant and engaging approach would be to discuss and edit prepared code provided by the course leader. With blended learning, you may choose to record this discussion. Here, you have opportunity to rehearse, retake and edit the video. So it might well be appropriate to include these errors and their solution in a video if you think it is valuable to the students. In this case, consider scripting the inclusion of the bug as part of the video.

Assessment and feedback

The details of marking and feedback are key for students to improve their coding skills. Ideally, this would be in person to allow a discussion and ensure what has been stated is understood. However, in many instances, there may be neither enough demonstrators nor time. The next option is for this feedback to be written and individual. The challenge then, with multiple demonstrators, is to ensure consistency. In my experience, the majority of students querying their mark is due to insufficient feedback. Figure 5 lists some suggestions to improve feedback by making it more meaningful and more efficient. Remarking any task after a student complaint is the least desirable scenario and embarrassing. However, there are steps that can be taken to reduce this. Automation is guaranteed in online quizzes, though one should expect a few teething problems with new questions. Otherwise, you could specify the required names of certain functions and check their output with a script. If you choose to grade coding style, a linter can help significantly.

Given this is a programming course, we could write a script or develop an interactive spreadsheet that is used for marking. This then simplifies allocating marks for different aspects and, if negative marking is used, can cap deductions for different areas. This will significantly improve the time it takes to mark a task, though care should be taken to not have too many categories. As an example, when marking variable and function names, we could mark by selecting the appropriate option from: *All correct*, *1-2 unclear*, *3-4 unclear* or *5+ unclear*. This could be implemented as a drop-down menu or a check box. You might not want to share the full mark scheme with students, as it removes the need for their thought about design and process. In this case, produce a simplified rubric to serve as guidance to what is expected from them. If marking is done online, then double-marking can be implemented. If the discrepancy is visible to demonstrators, they can then take responsibility for their work and review where needed. Communicating that each assignment is double-marked to students also provides reassurance to them. In practice, discrepancies are typically due to human errors from markers missing a single item from a script. Double-marking also forces reviewing code where students have done something non-standard, but not necessarily wrong (a more common circumstance).

Summary

In summary, we have provided detailed reflections and observations on developing an introductory programming course to non-specialists. This task is

Figure 5 Suggestions for assessing programming when taught to science and engineering students not on a computer science course.

- Provide model answers where appropriate.
- Record video of the lecturer or demonstrator going through the model answers.
- Automate marking where possible.
- Create a clear marking scheme for demonstrators.
- If resources permit, double-mark all assignments and flag significant inconsistencies.
- Give clear guidance to demonstrators on what is expected in the feedback.
- Allow time in case a demonstrator needs to improve their written feedback.
- Direct students to demonstrator support to receive further/understand their feedback, in the first instance.
- Include the demonstrator in correspondence with a student querying their assignment (where appropriate).
- Allow time for re-marking if there has been an error in the marking.

non-trivial. Given the importance of this skill in modern society, we must do more to engage students that might otherwise ignore this work.

We have presented a multitude of suggestions and commented on our experience. We did not implement all of these changes all at once. Indeed, it might take time to incorporate the ideas listed here. It might also be that some of the suggestions are not as imperative as others. That is fine; we are trying to accommodate a large number of student needs in one place. However, we do urge you to think about each point and how it might fit into your structure.

After releasing a summary of our course on a University blog, we received the below feedback from a student [26]:

> This course actually got me from being scared to code to taking the computational module in semester 2 and looking for placements to do with coding... Amazing.
> (Student comment on Twitter)

Getting students to dismiss their incorrect preconceptions about what a programmer does and who can be a programmer is difficult, but it can be done.

Acknowledgements

I thank Prof. João Quinta da Fonseca, Dr. Christopher Race and Dr. Tom Shearer for their discussion and input.

Declarations

Conflict of interest The author declares no conflict of interest exists with the publication of this manuscript or its data.

References

[1] Robins A, Rountree J, Rountree N (2003) Learning and teaching programming: a review and discussion. Comput Sci Educ 13(2):137–172

[2] Biggs J, Tang C (2011) Teaching for quality learning at university. McGraw-Hill Education, Maidenhead

[3] The Institute of Physics (2011) The physics degree. https://www.iop.org/sites/default/files/2019-10/the-physics-degree.pdf. Accession date 2021–05–08

[4] Great Britain. Parliament. Science and Technology Committee (2016) Digital skills crisis: second report of session 2016–17

[5] Russon M-A, Hooker L (2021) Uk 'heading towards digital skills shortage disaster'. https://www.bbc.co.uk/news/business-56479304. Accession date 2021–05–08

[6] Vahrenhold J, Nardelli E, Pereira C, Berry G, Caspersen ME, Gal-Ezer J, Kölling M, McGettrick A, Westermeier M (2017) Informatics education in Europe: are we all in the same boat. Assoc Comput Mach 10:3106077

[7] Koulouri T, Lauria S, Macredie RD (2014) Teaching introductory programming: a quantitative evaluation of different approaches. ACM Trans Comput Educ (TOCE) 14(4):1–28

[8] Brown NCC, Wilson G (2018) Ten quick tips for teaching programming. PLoS Comput Biol 14(4):e1006023

[9] Stephen C (2020) The top programming languages: our latest rankings put python on top-again-[careers]. IEEE Spectr 57(8):22

[10] Srinath KR (2017) Python-the fastest growing programming language. Int Res J Eng Technol 4(12):354–357

[11] Dhruv AJ, Patel R, Doshi N (2021) Python: the most advanced programming language for computer science applications. In: *Proceedings of the international conference on culture heritage, education, sustainable tourism, and innovation technologies (CESIT 2020)*, pp 292–299

[12] da Fonesca João Q, Race C (2020) Jupyter notebooks for the computing and communication 1st year materials science course at the university of manchester. https://github.com/JQFonseca/computing_materials_science. Accession date: 2021-05-10

[13] Cheryan S, Master A, Meltzoff AN (2015) Cultural stereotypes as gatekeepers: increasing girls interest in computer science and engineering by diversifying stereotypes. Front Psychol 6:49

[14] Google (Firm) Gallup (Firm) (2016) Diversity gaps in computer science: exploring the underrepresentation of girls, blacks and hispanics

[15] Ewen C (2020) 'it will change everything': deepmind's ai makes gigantic leap in solving protein structures. Nature pp 203–204

[16] Institute of Physics. Inclusive teaching: 10 tips for teachers poster. https://www.iop.org/sites/default/files/2019-07/IGB-10-tips-whole-school-poster.pdf. Accession date 2021–05–08

[17] Teodosiev TK, Nachev AM (2015) Programming style in introductory programming courses. International Journal of Applied Engineering Research

[18] Ala-Mutka K, Uimonen T, Jarvinen H-M (2004) Supporting students in c++ programming courses with automatic program style assessment. J Inf Technol Educ Res 3(1):245–262

[19] van Rossum G (2001) Pep 8 – style guide for python code. https://www.python.org/dev/peps/pep-0008/. Accession date 2021–05–06

[20] Anaconda (2012) Anaconda software distribution. https://www.anaconda.com. Accession date: 2021–05–06

[21] Grasso SJM (2017) Use of a social question answering application in a face-to-face college biology class. J Res Technol Educ 49(3–4):212–227

[22] Washington T II, Bardolph M, Hadjipieris P, Hub ET, Ghanbari S, Hargis J (2019) Today's discussion boards: the good, the bad, and the ugly. Online J New Horiz Educ 9(3):219

[23] Constantinescu ND (2015) Piazza–a tool for class discussion. Benefits of its use and future requirements. J Sci Arts, pp 19–24

[24] Boyle T, Bradley C, Chalk P, Jones R, Pickard P (2003) Using blended learning to improve student success rates in learning to program. J Educ Media 28(2–3):165–178

[25] Hadjerrouit S et al (2008) Towards a blended learning model for teaching and learning computer programming: a case study. Inf Educ Int J 7(2):181–210

[26] [@yelnastyy] (2021) https://twitter.com/yelnastyy/status/1362858648062599177. Twitter, Accession date: 2021–05–08