Teaching Programming across Disciplines

Table of contents

1 Teaching Programming across Disciplines

Welcome to our book!

We are a community of teachers and educators of computer programming outside of traditional computer science settings. Our 300+ members around the world meet every year for a Winter School[1] and co-author a book called Teaching Programming across Disciplines[2] (this very book).

Our book is an edited, open-access volume comprised of many short chapters written by groups of authors.

We will be launching the **first edition of Teaching Programming Across Disciplines** at the "Summer-time Winter School" on **22 June 2026**!

The **hard deadline for chapter submissions** is **1 April 2026** – meaning your chapter should be edited, polished, and on the GitHub Repository (watch Pawel's video[3] for instructions on how to do this). The book will be growing from now until 1 April 2026 as chapters are added. There will be a print copy available (eventually… stay tuned).

We hold monthly writing retreats, which authors or anyone interested in having dedicated writing time with a group of lovely people are welcome to join. We also have a newsletter with book updates and (interesting) events. If you would like to join our mailing list, email pairprogramming@ed.ac.uk[4]

The Book Team (Brittany Blankinship, Franziska McManus, Pawel Orzechowski, Kasia Banas, Charlotte Desvages, Beatrice Alex, Umberto Noé, Ozan Evkaya, Serveh Sharifi Far, Clare Llewellyn)

[1] https://pairprogramming.ed.ac.uk/category/winter-school/
[2] https://pairprogramming.ed.ac.uk/join-book/
[3] https://media.ed.ac.uk/media//1_7nk68kt4
[4] mailto:pairprogramming@ed.ac.uk

2 Title of Example Chapter (as will appear on top h1 header and in table of contents)

2.1 Your first top-level section

2.1.1 A subsection

This is an example subsection, starting with `###`.

2.2 Quarto Markdown quick examples

2.2.1 Citations

An example citation (L. A. Williams 2010). You can also cite L. A. Williams (2010) in the text, without the brackets.

This is a reference to a chapter of a book (Sharifi, Qu, and King 2026). You can find it in `references.bib`; it uses the `@inbook` type.

The "References" section will be populated automatically at the bottom.

2.2.2 Footnotes

This is some text with a footnote[1].

[1] Write the content of your footnote here.
If you need a multiline footnote, indent the following lines with 4 spaces…
…like this. Everything indented will be part of the footnote.

2.2.3 URL links

Use link text[2] to include URL links; please refer to the accessibility guidance in the book submission guidelines.

If it is absolutely necessary to display the full URL in the text, use `<...>`: https://quarto.org

2.2.4 Images

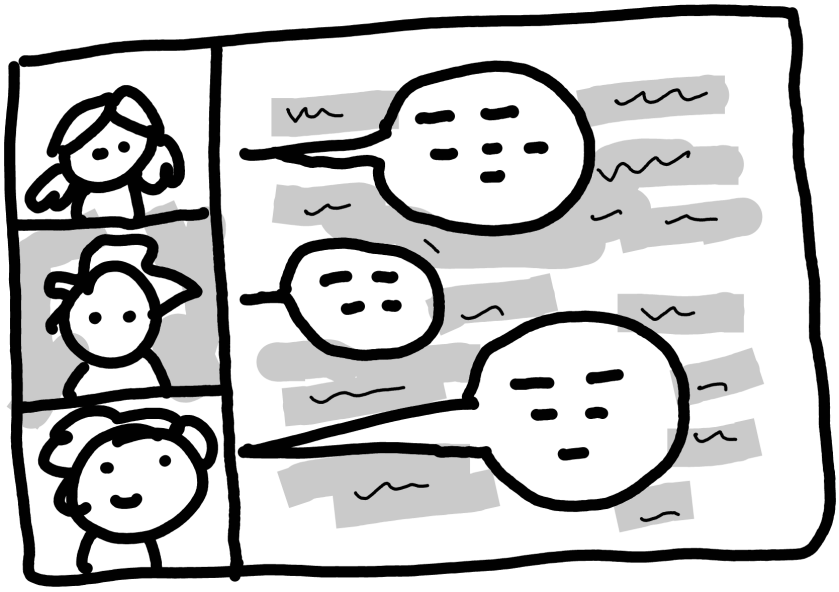

Figure 2.1: This is the caption for an image, sized to take 70% of the screen width.

[2] https://quarto.org

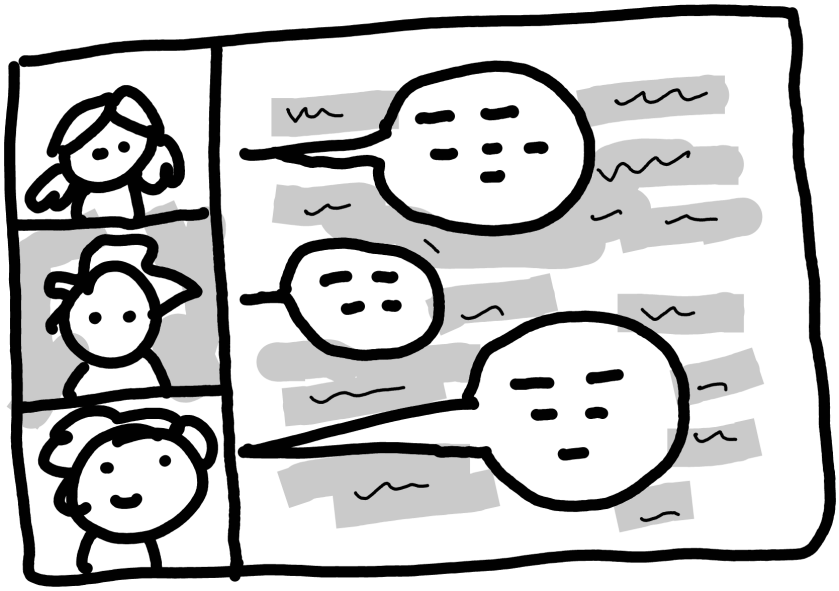

Figure 2.2: This is the caption for an image with a clickable link.

2.2.5 Putting some content in a box

Title of your box

You can put some content in a box like this, called a callout[a], if you want it to be visually distinct from the rest of your chapter. This could be e.g. if you write an example or a short case study.

[a] https://quarto.org/docs/authoring/callouts.html

3 Second Languages: Teaching C to Python+Jupyter Novices

3.1 Introduction

This chapter explores challenges associated with learning a second programming language, in particular for a population of students who are not specialising in computing. In the context of our case study, the second language challenge is compounded in two ways: firstly, the programming paradigm shifts from a high-level managed interpreted language to a low-level unmanaged compiled language (i.e. from Python[1] to C[2]), and secondly, the development environment shifts from a browser-based notebook to a terminal-based text editor (i.e. from Jupyter[3] to an editor such as Vim). We highlight common misconceptions experienced by students during this transition process and discuss potential fixes, in particular the use of a *bridge language* to ease the transition from high-level to low-level paradigms.

3.1.1 About Us

Two of the authors of this chapter — Sam Skipsey and Gordon Stewart — comprise two thirds of the staff teaching the P2T course that forms much of this chapter's basis. Sam has a background in Theoretical Physics, although they have always been interested in computing as a topic; they might well have ended up doing a computer science undergraduate course had things gone differently. Gordon is a computer scientist and software engineer by training, who now works as a research IT specialist. The other two authors are based in computer science departments and are involved with teaching programming to computer science students directly. They also have interests in computational thinking.

3.1.2 Physics at The University of Glasgow

The rise of Python as the most popular programming language in the world has been coupled with the emergence of Project Jupyter and its notebooks as an extremely popular programming interface in many disciplines. It is now frequently the case (writing in 2025/26) that the only

[1] https://www.python.org
[2] https://9p.io/cm/cs/who/dmr/chist.html
[3] https://jupyter.org

experience of coding that novice programmers may have involves "Python in Jupyter", and so naturally this becomes their sole model for how programming works. The ubiquity of Jupyter also means that some students may only know "Jupyter" as a synonym for another hosted development environment, such as GitHub Codespaces[4].

As a consequence of arguments similar to those described in (Balreira, Silveira, and Wickboldt 2023) and elsewhere, Python has displaced other languages in many university courses, not least in physics curricula. Broadly, the suggestion is that Python's perceived relative "simplicity" in syntax allows students to focus more directly on the actual process of "programming", rather than getting hung up on syntactical complexity as in (say) C. In the University of Glasgow's School of Physics and Astronomy, students encounter programming in their first-year labs when they use short Python snippets in Jupyter notebooks to process and analyse lab results; students subsequently receive a more formal introduction to Python in Jupyter during their second-year labs. In the rest of this paper, we will use "Jupyter" as a shorthand for "Jupyter with a Python kernel"; some of the aspects here are related to Jupyter itself, but others are dependent on the particular kernel (and thus the language) used, so we wish to be clear about the conflation here.

These two lab-based experiences are, for many Physics students, their very first introduction to programming and, indeed, their first introduction to the concept that programming is important to the practice of physics as a discipline. We've noticed that this sometimes requires a significant adjustment to the students' mental image of what it means to be a physicist in the modern world, which seems to be set by prior experience in primary and secondary education as "someone who does experiments and knows advanced mathematics". Students are also expected to take mathematics courses in their first two years at university, but because they expect this as part of their image of the subject, they do not dispute the necessity of this; they may, however, dislike other aspects, for example the extreme rigour of some proof requirements in mathematical processes. By contrast, a significant fraction of physics undergraduates in their first year do find the introduction of computing topics both surprising and, if not unwelcome, then at least not well justified.

3.1.3 "P2T" – a specialised course in C for physicists

In the second semester of year two, Physics offers the course "P2T[5]: C Programming Under Linux", which introduces lower-level programming in the C language[6], the use of the GNU/Linux operating system, and command-line tooling for debugging, build orchestration and version control. This course is a prerequisite for students on the Theoretical Physics track, who have the programming background mentioned in the previous section; meanwhile, approximately

[4] https://github.com/features/codespaces

[5] University of Glasgow short-codes for courses are almost opaque for historical reasons, but this was originally short for "**P**hysics level **2**: **T**heory".

[6] This is true of the 2024/25 iteration of the course, which was in place when we wrote the first draft of this chapter. We'll discuss which choice we made for 2025/26 at the end of this chapter.

half of the cohort each year is made up of second-year students from the School of Computing Science, who begin with a much stronger background in programming concepts, including experience of assembly language. In fact, P2T is a small (10 credit) course designed to stand alone with no formal prerequisites and thus is available to any student; in practice, however, only a few students outside of physics and computing science choose to take it.

When first developed, at the turn of the millennium, P2T was considered a rather specialised course and was delivered to a small cohort. Today, mostly due to interest from computing students, it has grown to around seventy students a year; for comparison, the total number of students studying second-year physics and computing science is approximately 180 and 330 respectively.

We have observed that the difficulties experienced by the physics cohort in P2T have changed as their prior experience has become dominated by Jupyter. In this chapter, we will discuss how this relates to the mental models formed when learning to program in Jupyter, and their differences from the models needed for lower-level programming. We will also suggest some potential bridge languages that would ease the transition and help students to develop better mental models.

3.2 Misconceptions + Difficulties

This section comprises a brief list of the most common conceptual challenges we observe in students used to Jupyter when learning C in P2T. Quoted text is paraphrase from a genuine student question, other titled sections are summaries of the general area of misapprehension. There is no implication in the ordering of the examples.

1. "If you have a test (e.g. $3 < 1$) which is false, does that make the compiler stop or the program stop?"

   This is quite a subtle distinction, which is less evident perhaps when learning first in an entirely interpreted interactive programming environment such as Jupyter, where code-blocks themselves obscure the flow of execution, let alone the distinction between compilation / interpretation and execution itself.

2. Source Code == Executable Code

   More generally, we often observe confusion between a program's code, its executable, and a process which represents a running instance of the program. The very first C exercise in the P2T labs walks students through the process of fixing some provided code, introducing an iterative sequence of compile-fix-repeat, and finally executing the compiled program. Despite this, towards the middle of the course we usually find that a significant number of students start trying to execute their source code, often after compiling it. In the case of P2T, this confusion is exacerbated by the fact that students are introduced

to shell scripting in Bash, an interpreted language, around the time they develop this confusion.

3. Import == #include

   In order to do most useful things in C, we start students off with the magic incantation `#include <stdio.h>` and explain that this enables them to use functions that interact with the terminal, such as printing out text or reading input. We do allude to the fact that `#include` only provides half of the functionality that `import` does in Python, but when we introduce the full concepts of header files and libraries later in the course, students find this hard to reconcile with their idea of `import`. In particular, the need for a linking step is an alien concept.

4. Types

   Types represent a significant learning curve for our students, especially those who are entirely new to programming, or who been introduced to Python via the physics educational flow (despite being encouraged to use NumPy almost immediately in that case). Students' misapprehensions in this category are varied, ranging from the difference between variable declaration and assignment (in C and more strongly-typed languages, declaration requires the type of a variable to be specified, while assignment does not) through allowable type conversions and the specific differences between different kinds of numeric scalar types, to uncertainty about the role and use of derived types.

5. CStrings and CArrays == Lists… and C types "are objects"

   In a sense, this is a subcategory of misconception 4, but it feels distinct because Python so strongly centres dynamic lists as the default container with a syntax strongly resembling that of C's decidedly static and strongly-typed arrays. This results in a number of conceptual problems for students either directly — trying to append to, slice or copy-by-assignment arrays — or indirectly trying to apply Python's "for-each" loop semantics to C via implicit array iteration (that is, writing a loop as a "for each i in CArray").

6. Scope and Lifetime / Encapsulation

   Python does have rules for both the scope of name bindings and the lifetime of allocated variables, albeit via garbage collection in the latter case, but the pragmatic way in which Python is taught in first and second year — and moreover the confusing way in which Jupyter cells interact with these concepts — means that students have not been exposed to these prior to starting P2T.

3.3 Mental Models

None of the above misconceptions are the students' fault: most are perfectly reasonable generalisations from an initial programming experience involving only Python in Jupyter, which

naturally insulates developers from some of the underlying mechanics required to translate a series of high-level source code excerpts into a program that a computer can actually execute (Johnson et al. 2022).

3.3.1 Blaming the Snake or the Planet

It's important to spend some time teasing apart the relative contributions of the notebook context and the language itself in forming these misconceptions. As one of the authors has noted previously (Singer 2020), there are specific pedagogic *disadvantages* to the notebook approach which have been understood for some time, and long pre-date the advent of Jupyter; one of the authors remembers some of these issues arising when using *Maple* in the late 1990s!

Broadly, we observe that notebooks in particular make it very hard for students to develop mental models of the flow of execution through code, and the way in which state is transformed in this process. There are notebooks — for example, the *Pluto* notebook implemented in Julia — which partially fix this problem. Pluto addresses this issue by specifically re-executing *all* cells by default, in order, when one cell is modified. However, Jupyter remains the mainstream choice of notebook platform, by sheer weight of deployments. Thus, it may be obvious that it is the use of Jupyter, not Python, that is the primary cause of misconception 1, and that Jupyter contributes to some significant extent to misconception 6.

Misconception 2 may also be caused by the use of Jupyter to a greater extent than the use of Python. Students also find it hard to reason about files as objects, a well-documented issue as increasingly locked-down operating systems prefer to present search tools and containerised applications with their own local files, rather than the OS's own filesystem (Chin, n.d.)[7]. The use of Jupyter correlates with this, as notebooks are both seen as more natural for students used to browser interfaces (as has become increasingly common over the 2010s and 2020s) *and* themselves promote that same disconnection of content from location. In this sense, we believe that misconception 2 is promoted less by the fact that interpreted languages are not translated into a permanent, separate, executable artifact, and more because even the concept of code as living in static, separate files is elided by the interface. Of course, this issue might also be raised with any REPL, but the image of the modern web notebook as "just another browser app" surely does not help matters.

Misconceptions 3, 4 and 5, conversely, clearly originate with the language itself, or at least with the pedagogic approach adopted by courses the students have previously undertaken.

3.3.2 A "Whorfian" third way

Pedagogically, the programming languages we teach students in a specific discipline should be chosen to promote particular modes of thought. Pragmatically, we are also pressured to teach

[7]Tangentially, we also see students confused by the "content" of a file not being a function of its filename suffix: an internalised Windowsism.

languages which provide students with employable skills, and these two goals may not perfectly overlap. Esoteric programming languages, for example, can help to develop specific problem solving or mental models in programmers (Singer and Draper 2025), but are by definition not widely used.

For our physics and computing science cohorts, there are therefore different reasons to wish to teach C in the first place:

Physics students:

- To develop mental models of low-level interfaces, important for those students who will develop experimental skills with electronic data taking, or hardware design; for example, the design of particle detectors may involve firmware development and FPGA programming.
- To give experience in performance programming in a language in which many core scientific libraries are still written.

Computing students:

- To develop mental models extending their knowledge of machine behaviour from assembly courses studied in first year, and feeding into low-level topics covered at honours level, such as operating system and compiler design.
- To appreciate that the Python runtime abstracts away many aspects of program behaviour, including dynamic memory management.

As such, appropriate mechanisms for developing those mental models may differ between the two cohorts. We explore two potential 'bridge' languages to scaffold the mental models for students before their introduction to C: one already existing for computing science students, and one proposed for physicists that may even be able to supplant C for many of these students.

3.4 Bridge Language: Java

In the School of Computing Science at the University of Glasgow, students learn Python in their first year of undergraduate study, alongside a specialised assembly language designed specifically for teaching. In their second year, students undertake several courses using the Java language, most notably a course on Object-Oriented Software Engineering. By the time they begin P2T in the second half of this year, they have been exposed to multiple languages for several months, and have started to develop the agility needed to adapt their mental models appropriately for whichever language they happen to be using at the time (Tshukudu, Cutts, and Foster 2021).

The choice of Java may be particularly useful as a bridge between the conceptual frameworks implied by Python and those required for efficient development of code in C and C-like languages. In some ways, we can consider Java to be a halfway house: like Python, it is

deeply object-oriented (in the C++ sense), to the extent that there are classes that wrap even the primitive fundamental types; like C, it is strongly-typed and its default containers are statically-sized, although Java does provide dynamic, heap-allocated containers in its standard library. Furthermore, Java syntax superficially resembles that of C and C++, further easing the later transition to C.

3.5 Alternative Bridge Language: Julia

We suggest that, for physicists in particular, there are other languages which are equally or indeed more compelling as bridge languages for low-level concepts.

Julia (Bezanson et al. 2017) is a just-ahead-of-time (JAOT) compiled, strongly-typed language significantly influenced by R, MATLAB and other languages oriented towards engineering and statistics. This influence means that much of the notation is designed to be familiar to mathematicians, and functionality for array programming concepts and statistical data processing is built in to the language, rather than being a feature of a supplementary package like NumPy. This has led to Julia becoming the teaching language of choice for a number of quantitative disciplines — for example, the University of Michigan features Julia in its theoretical courses in robotics (Grizzle, n.d., and others) — and has also seen support for it grow in other disciplines, including high-energy physics (Stewart, Graeme Andrew et al. 2025).

Indeed, one of the issues with the arguments for Python and against performance languages, such as those given in (Balreira, Silveira, and Wickboldt 2023), is that they beg the question, assuming that by definition a "performance" language must also be inherently less syntactically and semantically natural than an "intuitive" one (such as, it is argued, Python). This is neither obvious, nor actually true in the general case; C and C++ are arguably very unapproachable languages, but this is a consequence of their age and accretion of features more than it is due to their performance characteristics.

The power of Julia as a bridge language from Python is that one can gradually add more layers of subtlety and complexity as students develop their mental models. The equivalent process in Python requires learning large external modules such as NumPy, which effectively provide an additional domain-specific language wrapped in Python for dealing with these concepts. In Julia, such concepts are supported within the core of the language itself.

A common pattern in development in Julia is to write functions in a general sense first, with no type specifiers on their parameters:

```
"quotient(numerator, denominator) : Quotient of two values"
function quotient(numerator, denominator)
    numerator / denominator
end
```

Then, when necessary, one can specialise methods of the function for particular cases:

```
"quotient(numerator::Integer, denominator::Integer) : Quotient of two Integers - returns a Ra
function quotient(numerator::Integer, denominator::Integer)
    numerator//denominator
end
```

(Here we specialise `quotient` for any concrete `Integer` type variables to return a `Rational` fraction, rather than a simple division.)

This provides a natural way to introduce types and their relevance to students. In fact, internally Julia always specialises a function invocation by the types of the argument, and we can also introspect this if we want to demonstrate this fact to students.

Of course, Python does provide *type hints* as an optional feature from version 3.9. Unlike true type constraints, however, Python's hints have no effect on the Python interpreter itself; they are annotations to be parsed by external linters, or to act as additional documentation for the human reader. The actual output of the Python interpreter itself is unchanged by their introduction.

Furthermore, as almost all of Julia's standard library is itself implemented in Julia, Julia's introspection tools can provide a way for learners to look under the bonnet and explore the internals of the language. Unlike Python, there are almost no black-boxes that we can't examine. For example: the `@less` macro allows the student to look at the source code for any function that is loaded from a file in the current session. While this doesn't work for things defined in the REPL itself, there are packages that provide this functionality should it be needed.

```
@less isodd(3)

> isodd(n::Real) = isinteger(n) && !iszero(rem(Integer(n), 2))
```

(In fact, a view of the file in which this is defined, with the above line highlighted.)

Teaching performance programming in Python is — by contrast with teaching fundamental programming concepts — quixotically harder, in that much of the answer is to use Python fundamental constructs as rarely as possible. Whilst algorithmic design plays some part, the fact that we can make an implementation orders of magnitude faster by using NumPy intrinsics instead of for-loops muddies the lesson that we need students to extract from the experience. By contrast, in both C and Julia, implementations in the base language are about as fast as they can be. Efficiency is almost always gained by either algorithmic improvement or, in extremis, by making use of deep systems knowledge (cache-width, layout of data in memory, etc.) which is mostly language-agnostic, and can be relegated to a later course.

Because Julia *is* JAOT-compiled, however, lower-level representations are always available to the programmer, which can be useful for pedagogic purposes; something which is not possible

with Python. That is, we can always use the `@code_` family of macros to expose what Julia code is compiled to, in an active session, at various levels in the compilation process, including at that of the native assembly code (via `@code_native`). For example, this lets us show the significant changes in code generation when switching between floating-point and integer types for mathematical operations, or how tail-call recursion is optimised to more efficient loops!

As a result, Julia is both a more complete bridge-language towards low-level programming concepts than Java (or directly C) *and* a more suitable *single* language for teaching students in physics and other mathematics-oriented disciplines than Python + NumPy (+ Numba + …).

Of course, Julia is not a perfect language. In terms of popularity, it is still rather specialist; it is considerably less popular than Python, and less popular than Python + NumPy in physics circles. This is partly attributable to its relative youth — only 14 years to Python's 35 — but also due to network effects in language adoption. That youth, and a small development team relative to Python, also means that Julia's development cycles can still introduce more significant changes than the relatively stable Python 3. (Python, of course, passed through this phase during its 2 to 3 transition, when it was a similar age to Julia now.) This presents some difficulty in maintaining static teaching materials for some topics; automatic differentiation tools, for example, have been in some flux for the past year or so, due to changes in the language internals.

3.6 Alternative Bridge: Just Drop Jupyter

As roughly half of the misconceptions we note in this article are at least somewhat due to the abstraction and disconnect that Jupyter itself brings to the coding experience, a third and perhaps less dramatic choice might be simply to teach Python as a native language in a terminal context.

The Python interpreter is ubiquitous, especially now that Microsoft's significant interest in the language has led to it being available to script Excel workbooks; in MacOS and Linux contexts, of course, it is almost always available as a system component as well.

Writing Python code in actual files enforces structural discipline that isn't present when using notebooks, and also provides at least some reinforcement of the difference between source code and the thing that's actually doing the work. Indeed, one can wave at `.pyc` files if we wish to show a transformed output.

Other misconceptions in our list are not ameliorated by this approach, of course; in particular, misconception 4 is only partially addressable, and 5 is arguably intractable as C and Python simply have different object models. However, in a physics context, Python is almost always used with an implicitly-loaded NumPy package to handle scientific data. NumPy, in contrast to Python, cares deeply about types, and also enables new iterative constructs not found in the base language, which are broadly those constructs typical of array programming languages.

3.7 Conclusions

An important step towards students reaching competency in a programming language is to develop a mental model of the way in which the code one writes becomes a series of instructions that a computer can execute. While environments such as Jupyter undoubtedly provide an accessible interface to programming tools, this accessibility can come at the cost of rendering the internal workings of the languages opaque, which can make it harder for students to gain the insight necessary to progress to more advanced domains. Rather than attempt to move straight from interactive Python development in Jupyter to a low-level language such as C, it may be fruitful first to introduce a bridge language that allows students to explore more advanced aspects of programming within a modern environment.

The choice of bridge language can be discipline-specific, however, since different disciplines program in different contexts. Whilst there are arguments for languages such as Java for computing science students, we suggest that choosing superficially more "esoteric" languages, such as Julia, may be a better pedagogic fit for physics students. If that suggestion is considered too radical, the alternative of simply teaching Python *without* Jupyter may help to ground student understanding in ways useful to their future development. This is particularly true in a scientific context where "Python" is actually taken to mean "Python with some combination of NumPy / SciPy / Pandas / <insert your scientific library of choice here>".

3.8 Postscriptum

After this chapter was drafted, the decision was made to change the contents of P2T. The current version, which is in progress at the time of writing this postscript, broadly follows the "Just Drop Jupyter" approach. Whilst the pedagogic advantages of Julia were recognised by the relevant committees, it was felt that retaining a through-line with Python was more generally useful, and also allowed a wider subsection of academic staff to be able to teach the course.

In exchange, the course now covers software engineering concepts more substantially, with lab material on unit testing, performance profiling and reproducible code and packaging now provided. The performance profiling aspects have been the most challenging, in overcoming the intrinsic "just translate it to NumPy" problem, but we have taken the opportunity to introduce the physicists to some light conceptual exercises on the scaling of different data structures as part of this.

Interestingly, immediate feedback from the physics students who had taken the previous version of course has been mixed: those who felt the need to comment mostly argued that learning a performance language like C had been valuable to their deeper understanding of programming as a discipline. It is, of course, too early to say if this is a wholly successful change, although

interim feedback has been generally positive… except, again, for a small number who were hoping to learn C!

4 Sequential vs. Simultaneous: Approaches to Learning Programming and Statistics

4.1 Introduction

We live in a data driven world. Much research in STEM departments requires skills in data description, prediction and inference which are usually developed during statistics modules but, increasingly, non-trivial statistical analysis requires some degree of computer programming. Therefore, for many undergraduate degree programmes, two closely interrelated but fundamentally distinct disciplines, programming and statistics, must often both be taught to undergraduates. There are different possible approaches regarding how to teach these subjects, most notably whether to deliver them simultaneously or sequentially. In this chapter, we focus on undergraduate level teaching in subject areas that are not primarily programming or statistics. Based on our own teaching experiences, we discuss the advantages and disadvantages to different approaches in an attempt to provide guidance on the best way to approach curriculum design.

The topic of teaching statistics to non-specialists has been discussed at length (Kelly 1992; Mustafa 1996; Metz 2008; Gimenez et al. 2013; O'Hara 2016; Bromage et al. 2022). How to do so while acknowledging that programming is also an important and related skill has received less focus which we hope to begin to address here. In all scenarios teachers need to be mindful of their own educational context, the learning outcome of their course and, above all, what knowledge and skills they want students to get out of learning both programming and statistics.

4.2 Teaching statistics before programming

Based on our experience, this is perhaps the original or "old school" approach. Many authors in fact were taught in such a way in their own undergraduate degrees prior to the advent of high quality open-source programming languages (e.g., Python, R, etc.) being used widely in academia. The primary advantage of this approach is the focus on core theoretical concepts in statistics (e.g., probability distributions, critical value tables) without additional considerations regarding technology or implementation. In disciplines where students are likely to need to perform statistical analyses in different applications and using a range of technology depending

on the subfield they study, this may be the most useful approach. This is also true in disciplines where computers are not always available such as fieldwork, where we want students to be able to identify where they could do 'back-of-the-envelope' statistical analyses. This approach serves to more prominently highlight to students that statistics need not always require computational implementation, which can be lost when programming and statistics teaching are commingled. However, from our experience and perspective, this may be less preferred in undergraduate contexts where the primary subject area is not statistics or programming as this can lead to the links between statistics and programming being lost - potentially leading to a reduced insight into the applications of both.

4.2.0.1 Example:

A first year geoscience course, which first teaches statistics independently of programming before moving on to a combined approach using the Python programming language. Later in the year, students were on fieldwork at the "Ammonite Pavement" in Dorset, and were asked to measure the size of some ammonite fossils. As they were comfortable with pen-and-paper statistical analysis, they took the initiative to more rigorously find the distribution of ammonite sizes, and to calculate how many ammonites they needed to measure to be confident of having a representative sample. Though the statistics course did continue on to students applying programming to solve statistical questions, the in-the-field experiment was made possible by students first learning to do statistics by hand.

4.3 Teaching programming before statistics

A complementary approach we have identified is to first teach programming, and then follow this with statistics lessons that make use of students' programming skills. Here, students have time to consolidate their programming skills and build their programming mindset whilst still being able to build knowledge and understanding through a computational investigation of statistical concepts.

The main disadvantage of this approach is that it requires more curriculum time, which many programmes cannot afford. It also requires that the two courses are designed with interaction in mind. For example, using the same programming language in both courses (e.g., jamovi), or purposefully keeping the programming course language agnostic (e.g., giving examples in multiple languages). Teaching statistics in a manner which is reliant on the use of programming may present a barrier for students who are less confident in programming. Indeed depending on the overall curriculum design, students may have forgotten much of the programming learnt by the time they are learning statistics.

4.3.0.1 Example:

The concept of variance was introduced using a programming approach prior to describing the formal statistical method when teaching Analysis of Variance (ANOVA) to a second-year undergraduate course in biomedical informatics. Students were first shown how to use simulations (sampling within replacement) to compare differences between pairs of samples within and across treatment groups. These empirical results were used to derive the probability of these groups being statistically different from each other and introduced the concept of variance as a quantitative measure. In the subsequent week, the formal mathematics behind the ANOVA were introduced. Students learnt how to implement this using R and, when reasonable, to perform post-hoc tests to determine which treatment group pairs were significantly different from each other. This deliberate approach allowed students to fully comprehend the underlying theory before learning how to implement this and interpret the output in real-world scenarios.

4.4 Teaching programming and statistics together

The final approach we have identified is to teach programming and statistics together. This can take multiple forms: within a single module one can teach either programming or statistics first, and then immediately use these skills to teach the other, or the teaching of programming and statistics can be truly interleaved.

Teaching these subjects together has clear advantages. Students can experience a more immediate and deeper understanding of statistics, since they can independently probe the concepts being taught, e.g by running simulations. This could facilitate students "building" their statistical knowledge, after an introduction to a new programming and statistics concept each week. For those students where the barrier to statistics learning is past experience with maths (Pletzer et al. 2010), presenting statistical concepts through code snippets may reduce this barrier compared to presenting the same idea using mathematical notation. However, for others the opposite may be true and they could find it easier to explore concepts programmatically after a formal mathematical introduction.

A major disadvantage of combining the teaching of programming and statistics is the risk of cognitive overload. Where both topics are taught concurrently, students do not have the time to consolidate information into their long-term memory and thus the working-memory load is higher (Sweller 2018). Many students will experience hurdles in their learning of both statistics and programming, and these may interact to further impede student attainment compared to students who are more comfortable with one or both topics. For students who are not being taught in their native language, the need to acquire domain-specific vocabulary in two topics at once may further exacerbate issues of cognitive overload. Thoughtful course design can reduce this burden, but it is likely to remain higher than with standalone courses. Besides, the volume of the weekly content should be balanced and tied sequentially in a logical flow to avoid further issues on students' cognitive overload. There may also be concern that students

will experience reduced development of computational and algorithmic thinking, since they may only use programming to do statistics. For some disciplines, this may be an appropriate outcome, if the field generally only uses programming for statistical applications and analysis. However, where the course aims to set students up to progress to higher-level programming courses, students may benefit from programming teaching being focussed on a broader range of applications. Besides, the use of suitable programming language can be a key aspect that may impact the student learning performances positively or negatively.

4.4.0.1 Example:

After seeing a concept such as linear modeling as a part of statistical modeling toolkit, 1st-year undergraduate students from different mathematical degree programmes were shown the related code snippet. Slides either showed the mathematical and code representation side by side or sequentially within this same class, allowing students to make connections across the two approaches.

After lectures, students participated in workshops where context-specific questions with new data sets were shared to give students a fresh start each time. For workshop exercises, they needed to either adapt code snippets based on the given data or create new versions of code to solve the given exercises. It is important to leave sufficient pauses - either during or between classes - during this style of delivery to allow students to fully digest the content from both disciplines.

4.5 Practical strategies for teaching statistics and programming

We next include a number of specific approaches to teaching both programming and statistics. Firstly, we believe that it is important to embed all teaching of these topics into the subject matter of the overreaching course or programming, particularly when programming or statistics are not the principal teaching focus. This may include creating appealing visualisations, testing data to determine whether the patterns observed are statistically significant, or discussing the conclusions that can be obtained in the research field from this work.

Secondly, statistical methodologies should be described clearly using a computational approach. While it is helpful to discuss the mathematical approach to generating a test statistic, we recommend that this is routinely followed up with the programming approach required to generate the same result. In this way, the close relationship between these two types of work will be demonstrated while also consolidating the underlying statistical concepts by repeating them from different perspectives.

A third strategy is to develop a “programming or coding mindset” by foregrounding the use of programming as an important transferable skill in itself, rather than solely a tool to support the work of performing statistical analysis. Students need to be taught how to interact with

data using programming languages (e.g., R and/or Python) rather than directly through apps (e.g., give careful consideration to file paths, focus on good naming conventions in code, not using spaces in file names, and discriminating characters, numbers and special characters). Time should be devoted to debugging code, perhaps through live coding during lectures.

Finally we note that it is critical to motivate and enthuse students about the potential for both statistics and programming. For example, using a computational approach to perform statistical analysis can increase the reproducibility of the obtained results and improve the confidence that genuine scientific discoveries are reported. These skills are also highly in demand in the job market. As an illustrative summary, the gap in data-driven technical skills reported in 2023 in the UK includes specialist data skills (including statistics), programming and software engineering skills to manage and analyse data sets (Fearns, Harriss, and Lally 2023).

4.5.0.1 Example:

In teaching a postgraduate course that covers both Python and R, a course activity was introduced to promote the development of a "coding mindset" called 'Error of the Week'. This feature celebrates errors as a learning opportunity rather than a source of embarrassment or frustration. Students share errors on discussion boards, including solutions if applicable. The course team responds to every post and during the live session the course lead or tutor highlights a particularly interesting or common error to the class, debugging it live. These discussion boards then act as a common error glossary for the course. These strategies reduce the hurdle that programming can often present to students, allowing for more time to focus on understanding and interpreting statistical outputs.

4.6 Conclusion

We have outlined three approaches for teaching programming and statistics: teaching statistics first; teaching programming first; and teaching both skills together. These approaches each have advantages and disadvantages, and will be best suited in different settings. In deciding which approach is most suitable it is important to consider the desired learning outcomes, overall programme design and the cognitive load for students, which will vary depending on the educational context and prior student experiences. Considered design of teaching activities can overcome many of the barriers that students may experience through their programming and statistics learning. We hope that through highlighting the advantages and disadvantages of different approaches this chapter will help educators make strategic decisions about how best to teach and encourage the learning of programming and statistics to their students.

Links to other chapters

Possible links to other chapters, which we might need to articulate with (this might not be an exhaustive list, just those we identified so far - might be useful info for the editorial team too)

1. Curriculum Design Overview
2. A Practical Guide to Teaching Python as a Computational Tool for Introductory Data Analysis
3. A guide to empowering students to develop a coding mindset

5 Where do I even start with Pair Programming in my classroom? Conversation with seasoned practitioners.

5.1 Context

In this chapter we present a conversation between three educators who have been using pair programming for many years. They will introduce the practice and answer frequently asked questions about using pair programming in education.

You can also listen to the audio podcast version of this interview[1]. Thank you to Miłosz Karpowicz and Jan Banaś for audio editing advice and help.

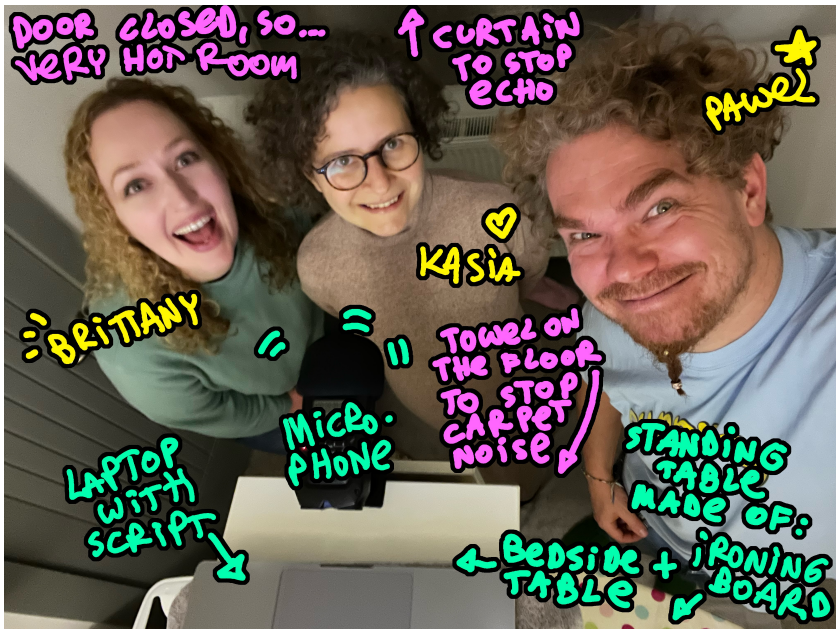

Figure 5.1: Authors recording the podcast version. We rigged the space for optimal audio quality and comfort.

[1] https://media.ed.ac.uk/media/1_xnjvow1u

5.2 Communicating the why

5.2.1 Let's jump right in:

Interviewer (Kasia): Hi, we are a group of educators at the University of Edinburgh. We have been using an asymmetrical group working technique in our classrooms, called Pair programming. It's used a lot in the industry, but not so much in education. Recently, a lot of colleagues have become interested in our experience of this, asking us questions at conferences and within our own university.

I've decided to interview my team who are experienced with pair programming in teaching. They will answer many of those common questions, and we hope this will benefit the educators who want to know how to start using Pair Programming in their classrooms.

Pawel: Hi, my name is Pawel and I have been teaching programming for almost a decade. I've always used Pair Programming in my teaching, and at times in my programming job before that. At the university I taught in Business School and Medical School, but also helped colleagues in many other departments. I love working with students on the beginning of their coding journey. They bring a lot of their own stories and thinking patterns to the table.

Brittany: Hello, my name is Brittany and I have been teaching programming and programming related topics for around 7 years in online and hybrid settings. More recently my teaching has focused on online masters courses and introductory courses. In my opinion, learning is a joyful experience and I am passionate about creating fun and joyful learning opportunities for my students, which pair programming definitely contributes to!

Kasia: Hello, my name is Kasia and I have been teaching programming for 5 years now. I also use pair programming in my courses, particularly those I teach in person. Join me over the next while as I interview my colleagues and ask them some of the questions that frequently come up when we introduce pair programming to other people.

5.2.2 What is Pair Programming?

Interviewer: Let's start with the basics: What is Pair programming (or pair-groupwork in general) and how does it differ from just working in a pair?

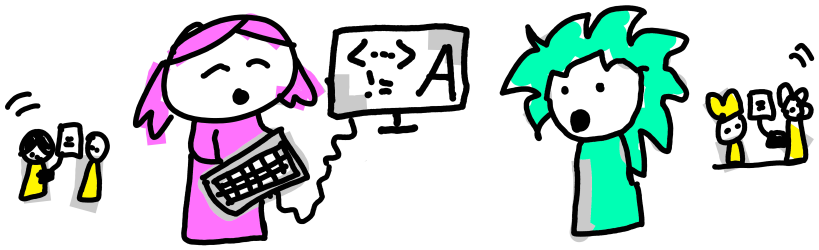

Figure 5.2: In pair programming two students work on one computer: Driver operates the computer, while Navigator helps them to find problems and offers advice.

Pawel: It's a new way to work on practical tasks, where students work in pairs and take turns. One of them is performing the active, leading role of a Driver, and the other one is supporting and guiding them as a Navigator. They switch after every task, or every 15 minutes, so both can experience driving and navigating. That's because there's a lot to learn from leading and from supporting. Communication is the key here, explaining what you're doing is a great way to learn.

Brittany: The role names come from the idea of driving a car - only one person is behind the wheel actively moving the vehicle, but the navigator in the passenger seat is crucial to making sure they reach the intended destination ("turn right here", "take that road"). In practice when pair programming, you would work on one computer, and pass the keyboard around, so that only the driver can 'DO' things like click or type. This means that when pair programming, you have to fully embrace your current role.

When it's time to switch roles you would just pass the keyboard around. Or a whiteboard marker, or whatever you are using to complete the task. So the core elements are: we take roles, we switch. But it's not like the navigator takes a break - both roles are active and have different responsibilities, they are asymmetrical.

Pawel: And what is really interesting is that this idea did not start in academia as a way of teach or learn. It comes from Agile Software Development practices, and programmers have been using it for a while when they work. Coding involves a lot of problem solving and that's easier to do when you can discuss it with someone. But when no-one is around, you talk to yourself, or even to an inanimate object like a rubber duck.

Interviewer: Oh, yeah, I've seen a lot of rubber ducks on the desks of programmers. There's that joke that it's good to explain your code to someone, but if there is no one around you can talk to the duck?

Brittany: Yes, they come from the same idea. I often tell my students that it is useful to talk out loud and sort through your ideas - talking is a way of thinking. In programming we have this idea of rubber-ducking - talking to an inanimate object like a rubberduck (I have a wee ghost on my desk not an actual rubber duck), my colleagues have them as laptop stickers, or

keyrings. With PP your rubber duck is a human who can talk back to you! Making it even more impactful, because most rubber ducks don't talk back! haha.

Interviewer: Ah, this sounds really interesting. And we said before that programmers in the software industry use this a lot. Has it been used in teaching? Have people found that pair programming can be beneficial in the classroom?

Pawel: It's quite a young technique in education, it's been gaining traction over the last decade or so. There is some recent research that using it in the classroom produces better code and more discussion between students, when comparing it to solo programming. Another team (Goel and Kathuria 2010) found that it is particularly beneficial for students with lower prior programming skills.

Brittany: We've been using it across our departments for about a decade, with about two thousands students in total. We've gathered a lot of expertise and are reaching a place where we're starting to look more analytically into what works and what doesn't. Especially what types of pair programming give people a better learning experience and better outcomes.

5.2.3 Why PP is good for learning

Interviewer: That's right, it is all about learning better after all. So how does pair programming fit into your courses? Where does it shine and where does it not?

Brittany: One thing's for sure: practical tasks is where working in pairs shines the most. When our students engage with a new creative task (like programming, problem solving or design) there's so much to think about. You're trying to work with the tools, but also keep in mind the context, and also incorporate whatever you learned recently. There is trial, and error, and exploration, and correcting your own mistakes. That's where working in pairs, as with a driver and navigator, works best.

Pawel: Imagine you're going on a quest, but not alone. There's always your trustworthy sidekick, your supporting character. Always on your flank, there to catch you if you stumble. It is easier to stay brave, and keep up the stamina when you know someone will swap you in a few minutes. Tasks where support matters, they work really well.

But yeah, some things do not work too well in a pair - things like reading paragraphs of text or watching videos. Any time when you're just passively consuming content. Maybe that's because everyone reads at a different speed, and takes notes differently (Orzechowski and Elaine Mowat 2026). We build our whole courses around this: students 'consume' content individually at home before the class. And then we meet all together for the practical exercises - those happen in pairs, in the classroom.

Interviewer: That's called a Flipped Classroom, right? Where instead of listening to the teacher together, and then practicing alone, the format is flipped. So students watch pre-recorded lecture videos alone, and then come to practice together with the teacher.

Pawel: Totally, it gets even better - you come to practice not just with the teacher but also with all your fellow students. Everyone becomes a little bit of a teacher, and a little bit of a student. It's like we do not have a classroom of 40 students and 1 teacher… It's more of a classroom with 41 teachers - everybody contributes! And that's where doing practical things works well - no one in the class has a full grasp of the concept yet, but we will all figure it out together.

Interviewer: That was meant to be my next question: some students will grasp the key concepts sooner than others, or some may have had previous exposure to the course content. Would doing pair programming with a group of mixed-skill students be more difficult? What if the two students in a pair have vastly different levels of experience?

Pawel: That's the really surprising thing, actually! It is just the opposite! Our students are here to learn, and the best way to learn something is to try to explain it to another human. When two people engage on that journey, they have to synchronise on what they believe is correct.

And yes, actual correctness is important too, you need quick ways to check if your solution works. That's why tutors are constantly visiting each group, throughout the class, and we also write our code so that it asserts its own correctness. But the bulk of the work happens with two students immersing themselves in a challenge as a team, trying to learn the most, to teach each other the most.

Interviewer: Ok, but doesn't this result in students working slower than they would on their own? You don't think that's a problem?

Brittany: We optimise for learning practical skills and being able to do it by yourself, a month later, with minimal support from notes or colleagues. But to get there, first everyone needs to build their muscle memory. And everyone is the operative word here - it's not just about YOU learning. You are a part of a cohort, and we are optimising for everyone in the group getting as much from the course as possible.

This type of community spirit, and empathy towards your peers, is really compatible with working in pairs. Members of the pair take responsibility for each other. But yes, there is some expectation management needed - you're not there to solve the puzzles as quickly as possible, you're there to learn and to help others learn and benefit from the opportunity of having another person to work with, rather than learning alone.

Interviewer: Oh, right, this reminds me of this approach in medicine, where they tell the students to "see one, do one, teach one". Young doctors would first watch a procedure being done; then they do it themselves under supervision, and then train others to do it. Only when you can guide someone else, you really understand it, right?

Pawel: Yeah, absolutely! When done well, pair programming puts you in all of those roles many times within each hour: you see, you do, and you teach. You get to see the problem from many angles, and also switch between learning in a passive, active, and reflective way. There's

nothing new in this idea, but what's new is how quickly we get to switch and learn. You're correct - overall the process, and the learning, is more valuable than the output.

Interviewer: Speaking of value, I totally understand why in academia we optimise for learning. But you mentioned that this practice comes from the industry. How do they justify using two people to do a job that one person could do, and potentially even faster?

Pawel: It appears to be puzzling at first, right? Counterproductive, using two people when one would do! But this false paradox comes from badly measuring what is good, what is efficient. The best writer or programmer is not the person who writes most book pages or most lines of code. Rather, the best writing fulfills its intended purpose and does it robustly. Extra points if it can be reused later for another purpose.

In creative industries, like coding, we can't think about efficiency like we do in manual labour, it's not like laying bricks when you build a house. A better metaphor would be that coding is like optimising a restaurant, so that food is tastier, and clients happier. It's about working smart, not working hard. Coding is not a repetitive, replicable task like peeling potatoes or assembling cars. Coding is about quality, efficiency, and internal communication. So two people can be justified if what they create is more future-proof and better integrated.

Brittany: And because each coding challenge is slightly different, and parts are so interconnected… it makes sense for two people to work together, and constantly check each other's work. The more something is interconnected, the more dramatic the impact of errors can be. We need code to be of super super high quality and that's worth every minute every person spent on it.

It depends on a task, but often another pair of eyes makes the code more solid, which is worth it in the long run. Additionally, pairing up and working with people across the team means more people have a deep understanding of what we're building, which makes for a more robust and agile team in the long run.

5.2.4 Students' reactions

Interviewer: Oh wow, I see how this approach would be useful, both in education and in industry. But do students get it? Do you have to spend a lot of time convincing them?

Pawel: Yeah they get it. They say that in a successful company 30% of the budget should be spent on marketing. We sort of do the same with our teaching methods. We spend a lot of time and energy explaining to students why pair programming is a great way to learn. Pretty much like we did to you, just now. Students are here to learn, and they pick up good learning practices very quickly.

Depending on the course, sometimes the benefits take a session or two to become obvious. But it always happens in the end. The persistence of the teaching team and reiterating the 'why' is needed at the beginning, but once the wheel is spinning, once everyone sees the benefit, it

just keeps giving. What you get quite soon is the excited buzz in the classroom as pairs code together. It's invigorating.

Interviewer: Have you ever encountered students who didn't want to be put in pairs, and insisted on working alone? What do you do then?

Brittany: Weirdly, this happens very rarely. Occasionally we put people in 'pairs' of 3... it is especially useful when someone didn't complete the pre-reading, or is anxious, or has technical troubles. And learning something new is quite often anxiety inducing, but having a familiar structure and clear expectations can be actually useful for calming the nerves.

Being at university, or even learning in general, is a social experience, so the fact that people are all different is great! This is one of the benefits of pair programming - interacting with people from different cultures, languages, backgrounds. Differences between you and your partner are a benefit, not a problem. The programmers would say "it's a feature, not a bug (haha)".

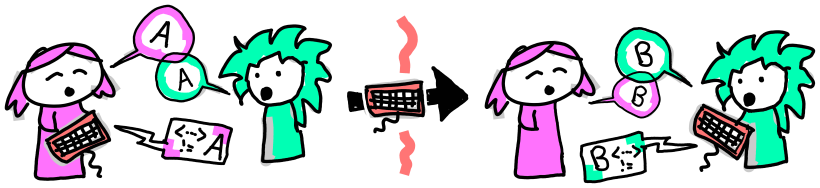

Figure 5.3: Students swap roles frequently to have experience of driving and navigating

5.3 Preparing for the session

Interviewer: Ok, this sounds really cool. Can we now talk a little bit about the practicalities? Walk me through how pair programming actually happens in the classroom.

5.3.1 How to run a PP session

Interviewer: For example, how long would your pair programming sessions be?

Brittany: In our teaching, each session is between 1 and 3 hours long. It should not be shorter than 1 hour for just coding, since students need time to 'settle into their chair', open the laptop, open software, say hi to their partner, etc. But it could be longer - it really depends on the number and types of tasks. It's also quite important to suggest to students that they should take breaks, particularly if the session is 2-3 hours long - we usually just tell pairs to manage their own time and take a break somewhere in the middle.

Interviewer: And what is the ideal class size for pair programming? How many pairs and how many teachers would you usually have?

Pawel: When we teach in-person, we are often limited by the room capacity, but online the only limitation is the size of a teaching team. My Business school course had 160 students. That's a lot, but I've split them into four groups things became very manageable. Each session had 40 students, me and 1 teaching assistant. It was a very good size. It was a bit taxing on the teaching team to run around for 2 hours, four times a week. But the benefits were enormous, and there were never too many hands in the air or a long queue of people waiting for help.

I would say 1 staff member can easily provide help to 10-15 pairs in person. It also really helps when the exercises we prepared for them are purpose-built for pair-programming. When students can investigate and solve their own problems, then there is much less staff support required.

Brittany: We initially thought that online would be quite different because trouble shooting takes longer and so you would need more staff, especially for beginner's courses. But after our experiences over the pandemic and our online teaching in the Medical School I'd say that one staff member per 5-10 pairs is a good proportion in an online classroom. And sometimes we are constantly (virtually) running between breakout rooms, but there are times when we just roam, and 'poke our head in' to see how they're doing. It's really nice when students say "since you're here, check out this cool code we've written!". haha

Interviewer: And how are the sessions structured? Is there an introduction or demo at the start of the session? Or a goodbye at the end?

Brittany: It really depends what the task is, and how it is anchored in the course structure. We try to follow the flipped classroom format, where 'talking at students' is discouraged. And instead we try to use the precious time together to work on practical tasks, in pairs.

Some lecturers start with a mini-lecture or coding demo, where they show students in 10-20 minutes how to solve the first task or give any course updates, and then students are assigned to their pairs and start by retracing the lecturer's steps. Same with gathering everyone for a debrief at the end - sometimes we do it, but often pairs are so immersed in their work that we do not want to interrupt them.

Pawel: And in terms of structure, once students are introduced to this format, after a few weeks, it just happens quite effortlessly. The teacher is still needed for some logistics, like creating the pairs, break-out rooms. But overall it's sort of self-organising, like, you do not need anyone to be in charge. And that frees us, teachers, to focus on helping students. There's a book I love about those self-organising activities which can be used in a classroom or a meeting, it's called "Liberating Structures", check it out.

But coming back to the pair programming, a good structure goes a long way. Forgive me a harsh example, but once I had a small bike accident on my way to work, and when I finally arrived in the classroom, what I saw stunned me. My students had already, all by themselves, split into pairs and just started working away on their tasks! It was great, I had that fantastic

feeling that you get as a teacher, that I am not really needed any more! That, once they understood the process, students just came to the sessions and get on with it.

Interviewer: Ok I understand the logistics a bit better now. And you mentioned that some tasks are better than others for pair programming?

Pawel: What works best is doing practical things. Especially when the exercises are separated into a series of small-ish tasks. Like I mentioned before, working together doesn't work too well for passively absorbing content, because we all process at different speed and or encode knowledge differently. So it is all doing small tasks, and if any reading is required, we instruct students to do it together and discuss what they've read.

But pair programming is really most suitable for active learning. It works when you have to together agree on a way forward, or when you together to decide the next small step that will take us there. It's like escaping a labyrinth or solving a puzzle together. For our students this process usually involves doodling, discussion and eventually writing some code.

Interviewer: This sounds like something that would be useful when you are already quite advanced. Would it also work for beginners?

Brittany: Ha, it's actually really great for beginners! It can be new or uncomfortable the very first time, because we're just not used to this type of honest and transparent problem solving or vulnerability of learning with someone else. For people who are just starting to learn something, pair programming is a safe and kind way to develop their ideas, their vocabulary, and so on. They also learn how to communicate about a new topic effectively with another person. You know how we have active vocabulary, but also a passive vocabulary - words we would recognise, but would never use in a sentence.

When students switch who's driving, they have the opportunity to work on both sides of their understanding: actively saying things, and understanding someone who says things. It challenges their misconceptions and blind spots. They say that you only understand something if you can explain it, in pair programming you constantly have to explain things. Back to that medical rule of: see one, do one, teach one.

Interviewer: We spoke about it briefly before, but what about mixed experience groups, when some people are further on their journey than others. Would they clash? Do the more novice students get intimidated, or do more senior ones get bored?

Brittany: Like we mentioned before, it really is about following the structure and switching. It can be humbling for a more experienced student to have the role of navigator and active switching enables both groups of students in different ways and it helps them in different ways. In fact, I recently got feedback from a more advanced student that they found pair programming really useful because it forced them to slow down and actually think about the code, what it is doing, and why they were making certain decisions.

But also we live in an imperfect world and it does happen that one of the partners breaks the rules. Sometimes people hog the keyboard, or talk over the other person, or dictate to the driver what to type, even if it's not their time to type.

Pawel: You need to be on the lookout as a teacher, occasionally check on the pairs to see if everything works well. There are many small ways to create an inclusive environment, or to give students opportunities to recover their flow when something goes wrong. There's a chapter led by Charlotte in our book, which describes those recovery mechanisms (Desvages et al. 2026).

To create an equitable world with diversity and inclusion in focus, we can't just care for students who are ahead, maybe because they had a leg up in their previous education. Our main focus is not to just teach the best students to be better, but rather to open the opportunities that coding provides to everyone (Guest and Forbes 2024).

Interviewer: Oh, I see how pair programming could be a great practice for inclusivity, and how we have an important mission here as teachers to make sure that it is in fact inclusive.

5.3.2 Logistics (room, equipment)

Interviewer: I have a few more questions about the practical side of things. When doing pair programming in person, what equipment do you need? Will any classroom do? Do you need special screens, tables, chairs?

Brittany: You actually need very little equipment - one computer for each pair is enough. If we work in a computer lab, each pair uses one computer and passes the keyboard around. It also works with passing one laptop around, but laptops usually have smaller screens, and if a 'pair' has three students it gets a bit crowded.

In terms of the room setup, as teachers we need the ability to walk around the groups to check in on them and to help them if needed. Which is why row-by-row lecture theaters do not work so well, because you cannot work between the rows of chairs. You also need an ability for students to sit together and see what is on the screen, which tends to be difficult for those mobile chairs on wheels with little pop-up tables.

Pawel: My preference is a good, old-fashioned classroom with desks and chairs. But the technique is very flexible. We've seen groups of 3 students connect to classroom screens to see better. I've seen instructors who put students in larger groups of 5, passing the keyboard around, gathered around a big screen.

But keep in mind that not everyone can see and hear well, so to include everyone's voice we need to have a good setup. We teach students how to adjust their computers so everyone can see well, hear each other, and be comfortable.

Interviewer: You mentioned pair programming online, so is this something that you can do in online courses? Or even in hybrid courses with some students online, and some in person?

Brittany: Over the last decade we've tried every possible combination. During the pandemic we had to suddenly start teaching online, including with pair programming, and it turned out to be very manageable, and in some ways even easier to run. For example, each group had its own breakout room, and the teacher could visit them with a click of a button. The students could also use the raise-hand feature to indicate they needed help. Instead of sharing a laptop, the driver would share their screen to show what they were doing.

In some way it made it easier to enact the roles, since there was no way the navigator would be tempted to touch the driver's keyboard from across an ocean. Among other benefits there's also the ease of switching between coding and doodling on a digital whiteboard (in-person students would doodle on pieces of paper). Overall, it was quite a manageable switch, with the student experience very similar to what we see in person. Now, years after the pandemic, we are still using pair programming in our online courses in the Edinburgh Medical School, with really good results!

Pawel: And indeed over the last few years in the Edinburgh Futures Institute we have been teaching large hybrid courses. Hybrid means that students in both modalities, some online and some in-person, pair-program together in real time. Imagine, a group of people in Edinburgh, sharing their screen and coding together with groups of people around the world. It was slightly chaotic, but also very inspiring and fun.

With my colleagues, Bea and Clare who wrote a book chapter about it (Beatrice Alex, Llewellyn, and Orzechowski 2026), we run our 30 person course in programming for Social Sciences like that. With our other colleagues I've run a similar course with 300 students. That's 300 people who would work in a hybrid way (some online, some in-person) with each other.

It requires a lot of prep, great knowledge of technology, and a really good team who can think on their feet. But it works! What makes it possible is the attention to small practical things like good earphones and microphones. When students can hear their partners without too much distraction in the background, all is well.

Interviewer: Wow, 300 students, half in person and half online, working together. What a thing to imagine! Amazing!

5.3.3 Tasks students work on

Interviewer: Ok, so it sounds like you're using pair programming a lot, in many different modalities. Tell me about the design of materials for pair programming sessions.

Brittany: What really works is if we have a set of small programming tasks. This way students can switch who's driving after each task. If tasks are larger, they could switch according to the clock (e.g. every 10 or 15 minutes). We try to have a few smaller warmup tasks at the beginning of the session, to give the pairs a bit of confidence. But once they're warmed up, the tasks can get more difficult and creative.

Interviewer: So in these tasks, what sort of code are you having the students work on? Do you give them a place to start, expected results, or tests? How detailed are the instructions?

Pawel: It really depends. In general we keep written instructions to minimum, since 'reading together' is not the best activity to do in a pair. Then we lean on the flexibility of pair programming, adjusting it to the level and topic of the course. For example, on a beginner course you may be given some code to get you started and more structure on what to do. Often you would get a worked example, maybe with something small to tweak in it.

On more advanced courses the tasks are often really open-ended, because students know how to navigate code, and also are used to working in pairs. The challenge is to provide enough guidance so that students are not lost, but also without making it a simple paint-by-numbers with no creative thinking or agency.

Interviewer: That makes sense. So I imagine each group will go at a different speed, choosing to focus on something else, or going deeper into another thing. What if they do not finish everything in the allocated session time?

Brittany: Oh, that is absolutely fine! Usually, it is the case that many pairs do not complete all tasks, or often they choose to skip some. But we don't let them get stressed out about it. Sample solutions are typically shared with students after the class, which can help to moderate the focus just getting a functioning answer without much thought or reflection.

Completing the tasks is not the main focus of the session, instead the real focus is on something else: to practice communicating and working with the human they are paired with. And developing a vocabulary for discussing new concepts and topics, and learning from each other. It does not matter how far they got into the materials, as long as they learn stuff really.

5.3.4 Creating Pairs

Interviewer: Ok, so what I'm getting here is that a lot of it hangs on the human relationship with your programming partner. How do you design the pairs? Do you do some kind of social engineering to create good combinations?

Brittany: It depends. We usually tell students who their pair will be at the beginning of the session. This makes things easier, and simplifies the chaos of 'where should I sit' and 'would you like to be in a pair with me'. And when creating those pairs we optimise for people having many different partners, preferably being with a new partner every time. This is great for building community, making friends, and the cross-pollination of skills.

But also for some people with social anxiety it can introduce uncertainty or worry, so there's always a way to tweak the pairs for those students or allow them to pick. Indeed our colleague Charlotte, from the math department, lets students pick their own partners so they can work with friends. From what Charlotte has shared, this can have a positive impact on attendance. It really depends what cohort of students you work with, and what are your constraints.

Pawel: And in terms of how exactly we come up with pairs… we use different strategies for that. Often it is just purely random, we shuffle a list of names and that decides who's with who. And sometimes it's quite fancy, and we can be very thoughtful about creating the most diverse pairs. Since we're all geeks, we even wrote a website which creates perfect pairs for you![2].

One way or another, since pairs are generally not repeated, we lean on the gods of randomness to just sort things out: if you had a 'meh' partner this week, then chances are you'll have a better one next time. But also you can't engineer everything and that's ok. There's a lot of 'letting go' that you need to run a class like that.

Brittany: Yeah, above everything else, the pairing strategy needs to be practical and quick, for example on courses where attendance is patchy or many people are late… we just put people into random groups as they appear. It always works out in the end.

Interviewer: What if you have an odd number of students? How do you split 13 students into pairs?

Brittany: It's really not a problem! Often we create some pairs of 3 people, where they have two navigators and one driver. They still would use just one laptop but pass it between 3 people. This flexible definition of a 'pair' comes quite handy in unexpected moments: quite often we end up with 'pairs of 3' as a contingency plan, for example when someone is unprepared, or anxious, or takes time to warm up to the method.

Same goes for any logistical issues, like patchy internet in online classes, or someone being very late - a group of 3 is our generic solution to any trouble. Indeed, on some of my courses internet problems are so common that the groups have 3 people as default.

Pawel: But also our colleague Umberto in Psychology has been experimenting with groups of 5. Where one student is driving, and others take various other roles, like Scribe, Strategist, Researcher. In the industry this type of method is called 'mob programming', a little bit like with "pitchforks and torches" type of mob.

It works really well when the driver is pitching the solution to the group and they approve of it, or suggest improvements. It seems to work well, especially when well integrated into the design of the whole course. Pair programming is a very flexible way to collaborate, but it's important to communicate the roles, the structure and the expectations to students very clearly.

5.4 Running a session

5.4.1 Before we start / Preparing the group

Interviewer: Right, that makes sense - the whole class should be on board and know exactly what they are doing. How do you achieve that? How do you onboard the class, especially the

[2] https://ddi-talent.shinyapps.io/pair-up/

first few times?

Brittany: We start each course with explaining why the method works, and how to get the most out of it. Students should know what the roles are, which we sometimes illustrate with a live demonstration. It is also important that they know 'who is who' immediately - we tell them that the first thing to do once they find their partner is to designate the first driver. Within minutes they will switch anyway, and both will get to be the driver, so we want them to start coding as soon as they get into their pair.

Pawel: Yeah, since this method really requires students to follow the roles, the script - we ensure that only one person types, and that they only have 1 computer open in front of them. Whenever we check-up on a pair, we ask those questions, like "who's the driver right now?" and "when was the last time you switched drivers?". But also we make sure that if they get stuck, they know how to call for help, or get themselves unstuck (Zarb and Hughes 2015).

Interviewer: So what happens in the very moment when two students are being put together in a pair? What does the first 30 seconds look like?

Pawel: Just before sending students to their pairs we reiterate first things they should do in their pair, like a simple code of conduct. It's all common sense: how to greet each other; how to choose who drives first; and how to create a kind and open environment for everyone. But knowing where to start makes everything much smoother.

Brittany: It's important to remember that negotiating accessibility features is a part of that initial hello. Agreeing on things like text size and colour scheme, or whatever else is needed so that both partners can meaningfully and equally contribute.

Interviewer: I am imagining myself as I pass the keyboard to someone else. But this is my code, they will change my work, I imagine having feelings! Have you seen this sort of attachment to one's work in your students?

Brittany: Ha, that's a good one. What we've seen is people writing simpler code because they no longer write it for just themselves. What you write as the driver, has to be understood and 'accepted' by the navigator. The navigator needs to fully understand it, because in a minute they will continue from where you left off. A simple example is that if you use meaningful names for things, there is less confusion about what you meant, which also means less bugs in your code.

Pawel: It's quite eyeopening and creates a lot of moments of pride! You stop accepting quick, shoddy code… or just thinking "oh, I'll fix this later". Instead we switch to long-term, kind thinking like "I need to write this code clearly and simply enough, so when we switch drivers, the next person can continue without much drama".

5.4.2 Once pairs are at work / Roles of instructors and students

Interviewer: And once the pairs are at work, what do the instructors do?

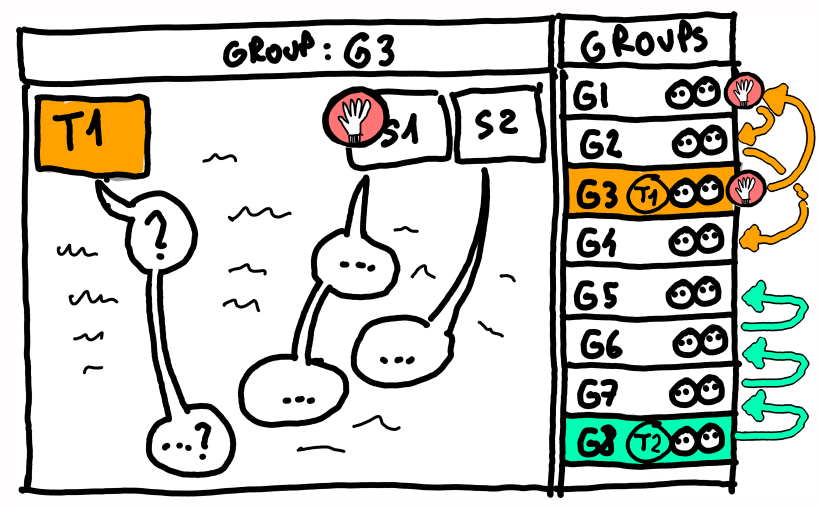

Figure 5.4: In online pair programming instructors continuously roam between breakout rooms, prioritising those with raised hands. Instructions take 'ownership' of parts of 'the room'. When instructor enters a breakout room, they first ask students to describe the problem and what was already tried. After that instructor will gives help students.

Brittany: We roam around the room and help the pairs. As a teaching team we take on two roles, really: being the facilitators and being the teachers. As facilitators we keep checking if students are doing the pair programming right: are they switching drivers, are they using just one computer, and such.

But really most of our time is spent being the teachers: checking if they are doing OK with the tasks, if they need any help getting unblocked or unstuck. Whenever someone raises their hand, we go and help them. And then once they are unstuck, we resume roaming. Usually in each class we would check in on each pair quite a few times.

Interviewer: So, as an instructor, when do you give help and how? Do you give your students the answers, or hints, or just ask leading questions? How stuck is too stuck? Do you ever just give them a leg up so they can start the next task?

Pawel: Totally, there's a subtle art of being a good tutor: knowing how and when to help. Not too early, but also not too later. Since the goal of class is to learn, not to just plow through the tasks, there's little value in just giving students the answer. It is ok to let students struggle a bit, because then they forge new ways to think.

But if it takes too long and is missing context, it can become frustration which is not good.It's sort of like lifting weights at the gym, you need some perseverance and effort, for things to stay in your head. We guide students and unblock them, usually just enough so that they can find the answer themselves. But we do help students a lot and in many ways, since not all struggles are good struggles.

As a tutor you tune into each individual situation and ask 'are we approaching a learning moment here, or are we just lost'. There are times when it's more beneficial to just unblock the pair, so they can move on to the next task. Especially when more learning waits for them over there.

Brittany: Getting unstuck or unblocked often includes asking students to explain what they have tried already and why they think it did not work. We also often switch to a whiteboard or a piece of paper to "doodle ourselves out of trouble". We try to use drawings to identify the solution first, and agree between partners what it could be. And only then they try to type the code which represents that doodled solution. At its core, solving problems is what we teach. Typing code is just the final step of that process.

Interviewer: So there is a lot of teacher skill there, but also a lot of students' self-regulation. For example, when should they decide that they are 'stuck enough' to ask for help? Do you teach students how they should approach problems?

Pawel: We do. Indeed, that's at the core of pair programming. Students try to solve the problem first with their partner before they ask the teacher for help. In primary schools kids are taught to escalate problems with this mnemonic "brain, book, buddy, boss". Your brain is your first point of call when you face a problem. Then your books and your buddies. Only once those failed you would approach the teacher.

When a tutor approaches a pair who asked for help, the first thing they would ask is "what did you try already?". That's important because solving things yourself creates learning opportunities. What's valuable here is that Brain, Book and Buddy is not a passive experience of asking or trying to remember something. We encourage students to actively experiment. In the context of programming, it means creating little code experiments, small 'sandbox' solutions, to tease out the answer. It is often faster than googling.

Brittany: Exactly! Running code and seeing what happens is the best way to go. I often tell my students "You won't break your computer, you might break the code but we can fix it". Let me give you a geeky example: two students are wondering what happens if you add two words to each other, like "pen" and "apple". Would Python just glue them making a "penapple" (all one word) or would it do what a human would and put a space in between them making a "pen apple" (as 2 words). Or maybe it will freak out and error! Students could spend 3 minutes or more googling for the solution, but they might also take 10 seconds writing a one line of code which adds "pen" and "apple" to each other, and just see what happens.

Pawel: Those tiny experiments are the very essence of what programming is.

Brittany: Indeed! Creating them does much more than just give you the answer - the process teaches you how to investigate problems, and how to understand the topics deeply. It's practicing the skill of problem decomposition - there's a saying that "there are no difficult problems in programming". If it seems difficult, you just need to decompose it into smaller and smaller problems, until they are simple to solve. (haha) and yes, it's that decomposition that is the real programming skill. And this is what students learn in pair programming.

Pawel: And coming back to the logistics of asking for help, students who need it would just raise their hand. Some classes use the Software Carpentries[3] model where students can stick a red or green post-it note to their screen, so teachers know how they are doing, and if they need help.

In big classes, like our EFI course with 300 students, our Teaching Assistants would act as the avant garde - they go to help students first. And if the problem is very challenging, then they escalate it further and call over the Lecturer. Really it is just more of that "brain book buddy boss" philosophy. To learn problem solving, as a whole class we need to practice problem solving.

5.4.3 End of the session

Interviewer: I see. So the focus in a pair programming session is to get more practical experience. And how do the sessions end? Is there a structure similar to how you start the session?

Brittany: Sometimes, the time runs out and we all just go home. But it is also nice to have a moment at the end of the class where the pairs come back together to debrief. People share how it went and what they learned. It usually turns out that there were common issues and themes - it's good to feel this community vibe at the end (haha).

But quite often pairs are so deep into their work that they do not want to come back to debrief with the class. Students go "just 5 more minutes, we're doing so well!". And that's a great sign! Having too much fun in the classroom is never a bad thing.

Interviewer: Speaking of ending the pair programming sessions, I think we'll need to wrap up our chat soon as well. We've discussed a lot of fascinating stuff. I am intrigued and now I want to try Pair Programming in my own teaching. What's your one piece of advice for people like me, who want to try it in their classroom?

Brittany: I'd say: find a way to experience it by yourself. Since Pair Programming is such an experiential, immersive thing to do, you will really get the point of it once you have done it yourself. When you have had to drive, and navigate, and pass the keyboard to someone. Experience it to understand it.

[3] https://carpentries.org/

All you'll need to start is: a willing colleague or friend; two hours of time; and a simple programming task (maybe a challenge from Tidy Tuesday, or Advent of Code). You could even come to one of our taster sessions which we run at conferences and such, where we pair you up with another participant, we give you something to do and let you Pair Program! You can find instructions on how to run such a workshop by yourself on our website[4].

Pawel: And my biggest suggestion is, once you've tried it, go full steam ahead, without half-measures. It might be tempting to 'simplify' the method and remove some parts of it, but you might end up throwing out the "baby with the bathwater".

For example if you do not have clear roles (who's the driver, and who's the navigator) it might fall into chaos or into working separately. If you do not switch, the person with more confidence or experience can end up hogging the keyboard. Or if you create bigger groups, people might disengage and not take ownership of the exercise. I'd say, start with the simple 'vanilla' off-the-shelf pair programming, and then find what works best in your context.

Interviewer: This sounds like very useful advice and first steps! And if people want to know more, they can find out about it on the website of our community, `pairprogramming.ed.ac.uk`[5]! Thanks for talking to me today. I am Dr Kasia Banas, and I've been talking today with Dr Brittany Blankinship and Dr Pawel Orzechowski from the Edinburgh Medical School at the University of Edinburgh.

Brittany: Thanks for having us!

Pawel: Thank you!

[4] https://github.com/teaching-programming/pair-programming-r-workshop
[5] https://pairprogramming.ed.ac.uk

6 Removing Barriers by Programming Without Computers

6.1 Introduction

Computational thinking (CT) has increasingly been recognised as a key competence for learners in the 21st century (Wing 2006). It is not limited to computer scientists but has been identified as a "universally applicable attitude and skill set" (Wing 2006). CT involves problem-solving processes that enable individuals to approach complex challenges by decomposing them into smaller parts, designing algorithms, recognising patterns, abstracting irrelevant details, and applying logical reasoning (Aho 2012). These skills are foundational for programming but are also valuable in wider contexts such as mathematics, engineering, social sciences, and everyday problem-solving (Tedre and Denning 2016).

Despite its importance, the teaching and learning of programming and CT is challenging. For many learners, the initial encounter with programming involves exposure to a programming language and development environment. While these tools are essential for producing executable code, they can impose significant barriers to entry. Syntax errors, cryptic error messages, and the abstract nature of code execution often discourage beginners (Robins, Rountree, and Rountree 2003) and many students focus narrowly on the mechanics of a programming language, rather than grasping the underlying concepts of problem-solving and algorithmic thinking (Koulouri, Lauria, and Macredie 2014).

These difficulties are not confined to school-aged learners. Lifelong learners, professionals seeking to re-skill, and students from non-computing disciplines also experience barriers when first attempting to program. Studies have highlighted that learners without prior computing exposure often report low confidence, high anxiety, and a perception that programming is inaccessible or only suitable for "technical people" (Lye and Koh 2014). Such perceptions contribute to underrepresentation of many groups in computing, particularly women, mature learners, and students from non-traditional backgrounds (Sentance and Csizmadia 2017).

Another barrier is the mismatch between learners' expectations and the nature of programming tasks. In formal education, assessments often emphasise correctness of code rather than exploration of ideas. Learners may therefore see programming as a high-stakes activity where failure is punished rather than as an opportunity for experimentation and iterative problem-solving (Watson and Li 2014). This can lead to disengagement, particularly when feedback comes in the form of obscure compiler errors rather than supportive guidance.

Alternative methods for introducing CT have been explored over the last two decades. The "Computer Science Unplugged" movement has shown that unplugged activities—teaching computing concepts without computers—can successfully build intuition and engagement (Bell et al. 2009). By using games, puzzles, and physical activities, learners can experiment with computational structures in a low-stakes environment. Similarly, tangible resources such as cards, tiles, and worksheets have been used to teach sequencing, logic, and algorithms in ways that reduce dependence on technology infrastructure (Curzon et al. 2014). These approaches are especially valuable in contexts where computer access is limited, such as in under-resourced schools or outreach programmes (United Nations Educational, Scientific and Cultural Organization (UNESCO) 2023).

This chapter discusses two approaches to teaching programming and CT without the use of computers. The first is the **ProgBoard**, a printable tool that enables learners to explore sequence, selection, and iteration. The second is a **play-kit for first-year computing students**, developed to foster computational thinking through playful and tangible activities. The chapter then draws connections between these approaches and the wider challenges of accessibility and inclusion in programming education, concluding with a reflection on the potential of such methods for diverse learner groups.

6.2 Unplugged Programming with ProgBoard

The **ProgBoard**, developed as part of the "Think Like a Computer" initiative (Cutting 2025) and available from thinklikeacomputer.org[1], provides learners with a structured way to engage with programming fundamentals in a tangible form. It is a printable teaching tool designed to introduce the three fundamental programming constructs of **sequence**, **selection**, **iteration**, and **variables** all without requiring a computer.

The board itself consists of a grid on which tokens representing instructions can be placed. Problems are posed in the form of challenges, such as navigating a path, following rules, or producing repeated patterns. Learners construct solutions by arranging instruction tokens in order, creating simple algorithms that can be "executed" by following the board step by step.

[1] https://thinklikeacomputer.org

Figure 6.1: A ProgBoard being used to break the ice with during international programming teaching in China.

6.2.1 Sequence

Sequence is taught by requiring learners to arrange steps in a specific order. For example, if the task is to move an object from one side of the board to another, instructions such as "move forward", "turn left", and "pick up" must be placed in the correct sequence. Learners can then manually follow the sequence to check whether the task is achieved. This provides an embodied experience of algorithm design, reinforcing the importance of ordering in programming.

6.2.2 Selection

Selection is introduced by conditional markers. For instance, learners may encounter a task where an object must only be picked up "if it is red". Tokens representing conditional statements are placed on the board, and learners must branch their instructions based on the condition. This tangible representation of branching logic helps learners to understand one of the most challenging concepts in programming: making decisions based on conditions.

6.2.3 Iteration

Iteration is represented by tokens that indicate repetition. For example, a loop marker may allow a sequence of steps to be repeated a specified number of times. Learners may be tasked

with drawing a square by repeating a set of instructions four times. Through physically repeating steps, they grasp the efficiency and power of loops compared to writing out repeated instructions.

6.2.4 Variables

Variables are introduced in the form of counters ("keep count of how many…") initially and then as **named variables**, for example "x is three, move forward x squares and each time your counter is in a red square, add one to x". The use of variables in combination with selection for branching logic and iteration allows learners to see how simple algebra aligns with variables in programming and also introduced the concept of state.

6.2.5 Benefits of the ProgBoard

The ProgBoard reduces cognitive load by focusing on concepts rather than syntax. By externalising algorithms into a physical form, learners can see and manipulate their thought processes. Errors become visible as misplaced tokens or illogical sequences, which can be corrected collaboratively. This is in contrast to the intimidating error messages encountered in programming environments.

The tool also supports **collaborative learning**. Students can work in groups, discuss solutions, and challenge each other to design efficient algorithms. Peer explanation reinforces understanding, as learners must articulate the reasoning behind their token placements.

Research on unplugged activities has shown that such approaches are particularly effective for learners with limited prior exposure to computing (Denning 2017). By engaging learners in hands-on tasks, the ProgBoard provides a stepping stone into programming that lowers anxiety and builds confidence.

6.3 The Play-Kit Approach for First-Year Students

A second approach to programming without computers has been developed through the design of a **play-kit for incoming first-year university students**. This initiative, derived from the work undertaken at Maynooth University on their primary school CT workbook, aimed to introduce learners to computational thinking through playful, tangible interactions (Anderson et al. 2025) so is therefore widely applicable and not limited to students studying computer science.

The project emerged from the recognition that incoming students often have diverse levels of experience in computing. Some may have extensive programming backgrounds, while others may never have written a line of code. A one-size-fits-all approach risks alienating both groups.

The play-kit therefore sought to provide accessible, engaging activities that introduce CT concepts in a non-intimidating way.

6.3.1 Design Principles

The play-kit was co-designed by students and academic staff, drawing on resources originally developed for primary school learners but adapted to suit a university-level audience. Several guiding principles shaped its design:

1. **Accessibility** - reliance on colour was reduced to support learners with colour vision deficiencies.
2. **Visual instructions** - activities emphasised pictorial guidance rather than dense text, reducing language barriers.
3. **Playfulness** - tasks were framed as puzzles or games, encouraging creativity and engagement.
4. **Breadth of CT skills** - activities covered decomposition, algorithms, pattern recognition, logic, representation, and abstraction.

6.3.2 Examples of Activities

- **Bracelet Patterns**: Students extend and complete sequences of beads arranged in patterns, developing skills in pattern recognition, decomposition, and logic.
- **Code Breaker**: A symbolic substitution exercise where learners decode messages using simple ciphers. This fosters skills in representation and abstraction.
- **Escape the Grid**: A maze-based challenge where characters must move according to specified rules. This introduces conditional logic and algorithmic pathfinding.
- **Castles and Wyverns**: A dice-based game where outcomes depend on nested conditions, encouraging learners to apply probabilistic reasoning and selection.
- **Marble Race**: Learners simulate the timing of marbles released into tracks, exploring concurrency, sequencing, and time-based reasoning.

These activities allow learners to explore computational concepts in a playful environment. Unlike programming assignments that require precise syntax, the play-kit enables safe exploration. Mistakes are opportunities for discussion and learning rather than points lost.

6.3.3 Preliminary Evaluation

Early feedback from workshops suggested that the play-kit was effective in lowering barriers for learners who felt intimidated by programming. Students reported increased confidence, greater willingness to collaborate, and enjoyment of the playful format (Anderson et al. 2025). The

tangible, hands-on nature of the activities was particularly valued, as it contrasted with the abstractness of code.

6.4 Discussion

The two approaches explored in this chapter—the ProgBoard and the play-kit demonstrate the potential of introducing computational thinking (CT) and programming concepts without direct reliance on computers. While they differ in context, design, and target audience, they share a common aim: to make programming more approachable by lowering barriers and foregrounding concepts before syntax. By situating programming in a tangible and playful space, both approaches challenge the assumption that programming must begin with a computer and instead suggest that thinking skills can, and perhaps should, be nurtured in more accessible ways.

One of the most significant contributions of unplugged approaches is the way they reduce **cognitive load** for beginners. Research has consistently shown that novices often struggle with the dual burden of understanding abstract programming concepts and mastering the mechanics of a programming language at the same time (Sweller 1988). For instance, a student may conceptually understand that a loop repeats a set of actions, but may be unable to express this understanding in Python, Java, or C without encountering frustrating syntax errors. By externalising programming logic into physical forms, as seen in the ProgBoard tokens or play-kit puzzles, learners can focus on what the computer is being asked to do, rather than how the instructions are formally expressed. This separation of concerns provides a clearer path to mastery: concepts first, formal languages later.

Equally important is the potential for these activities to support **diverse learner groups**. Traditional programming environments tend to privilege students who are already confident with computers or who have prior experience in coding. This creates inequities, with learners from non-computing backgrounds often left behind. Lifelong learners, interdisciplinary students, and those with limited digital literacy can find the initial learning curve intimidating, reinforcing the stereotype that programming is only for "tech experts." By contrast, both the ProgBoard and the play-kit offer accessible, low-barrier entry points where learners can engage in computational problem-solving without needing technical fluency. This is especially relevant in outreach contexts, professional development programmes, and in regions where reliable access to computers is limited.

The **affective dimension of learning** should not be overlooked. Fear of failure is a well-documented barrier in programming education, where small mistakes in syntax can lead to discouraging error messages and wasted time (Watson and Li 2014). In unplugged contexts, however, mistakes become opportunities for dialogue and playful exploration. If a sequence of tokens fails to solve a ProgBoard puzzle, learners can quickly rearrange the steps and try again. If a group misinterprets the rules of a play-kit game, the result is not an intimidating compiler error but a chance to reflect, discuss, and iterate. This transformation of errors into

constructive learning moments fosters resilience and helps to cultivate a growth mindset around programming.

Furthermore, the **playful nature** of these activities contributes to learner motivation and engagement. Play has long been recognised as a powerful mode of learning (Papert 2020; Whitton 2022). It encourages experimentation, risk-taking, and persistence, qualities that are central to effective problem-solving in computing. By embedding CT tasks into puzzles, games, and tangible challenges, the ProgBoard and play-kit avoid framing programming as a purely technical skill. Instead, they emphasise creativity, exploration, and fun—qualities that can be especially important for engaging students who might not initially see themselves as "coders." For example, the fantasy-themed "Castles and Wyverns" dice game from the play-kit illustrates how algorithmic logic can be learned through an imaginative context, rather than an abstract lecture on conditional statements.

The **transferability of skills** developed in these unplugged settings is also crucial. While the activities do not produce working code, they cultivate habits of mind—such as decomposition, abstraction, and algorithmic reasoning—that can be readily applied in formal programming environments. In this way, unplugged activities function as a **bridge**. Once learners have gained confidence with the underlying concepts, they are better equipped to face the complexity of programming languages and development environments. This scaffolding effect has been linked to improved retention and success rates in introductory programming courses (Luxton-Reilly et al. 2018).

Finally, both approaches point to the **broader potential of programming without computers**. Their application need not be limited to early stages of computer science education. They could be adapted for interdisciplinary teaching, allowing students in fields as varied as biology, business, or the arts to engage with computational thinking in a way that feels relevant to their domain. They could also be used in informal education settings—libraries, community workshops, or online learning platforms—to broaden participation in computing. For educators, unplugged methods provide flexible, low-cost tools that can be adapted to different curricula, learner profiles, and institutional contexts.

Taken together, these observations suggest that unplugged approaches should not be viewed merely as supplementary activities or temporary scaffolds, but as integral components of a broader strategy to democratise computational education. By making programming more tangible, playful, and inclusive, they help dismantle barriers that have historically excluded many learners from engaging with computing. In doing so, they lay the groundwork for more diverse, resilient, and conceptually grounded cohorts of future programmers.

6.5 Conclusion

Programming without computers represents an effective and inclusive approach to developing computational thinking skills. By focusing on concepts rather than syntax, learners are able to

engage with programming fundamentals in an accessible, playful, and low-stakes environment.

The ProgBoard demonstrates how simple, printable resources can teach sequence, selection, and iteration. The play-kit for first-year students illustrates how playful, tangible activities can introduce decomposition, algorithms, logic, and abstraction in ways that reduce anxiety and increase motivation.

Together, these approaches highlight the value of unplugged activities as a means of lowering barriers, supporting diverse learners, and broadening participation in computing. They serve as stepping stones to computer-based programming, ensuring that learners enter digital environments with confidence and conceptual clarity.

It is concluded that the continued development and evaluation of unplugged programming approaches has significant potential for education at all levels. By reimagining how programming is introduced, educators can make computational thinking accessible to all learners, regardless of background or prior experience.

7 Learning Together Across Modes: Online and On-site Pair Programming in a Fusion Course

7.1 Introduction

"Sarah, can you scroll down a bit?" The voice appeared suddenly in Sophie's headphones, causing her to jump slightly in her chair. Just moments before, she'd been chatting face-to-face with Dr Llewellyn about Python syntax errors. Now her disembodied voice was helping her online partner navigate their shared code, while she remained physically present in the classroom just three feet away. Welcome to fusion teaching, where instructors become friendly ghosts and the boundaries between digital and physical learning dissolve in unexpectedly delightful ways.

In this chapter, we want to share our experience of developing and teaching an experimental and innovative pair programming course. This course is non-traditional; we teach text mining to master's level students in fusion mode (both online and on-site students at the same time) in a two-day intensive session. We knew this would be challenging, but we also knew that this would be a great learning experience and a chance to do things differently. We present here an honest reflection of the trials, tribulations, and wins in what was ultimately a rewarding and satisfying experience. We didn't do everything perfectly, but we learned a lot, and we believe the students did too. Please read on and hopefully enjoy hearing about our experience.

Pair programming (Orzechowski, Blankinship, and Banas 2026; Desvages et al. 2026) is central to our course, but pairing students effectively in a fusion setting presents its challenges. It was important to us for our entire cohort to feel a sense of parity; this could not be an in-person course with people watching online. We wanted to develop fluidity between the modes. We worked hard at this and broke down the barrier between the room and online. We created a classroom in which both teachers and students can be vulnerable and open, so that we can grow from each other's mistakes, and communicate with respect.

When we first designed and ran *Text Mining for Social Research*[1], the Edinburgh Futures Institute's first fusion course at the University of Edinburgh, we knew we were stepping into uncharted territory (B. Alex et al. 2021). Fusion courses integrate both in-person and online students, requiring thoughtful interaction strategies to create a cohesive learning experience.

[1]This was a two-day intensive bootcamp course for 30 postgraduate students (15 on-site, 15 online). Each day consisted of four 1.5-hour-long sessions, with each session starting with a 15-minute introduction, an hour of coding in pairs and a 15-minute Q&A session.

While hybrid teaching models have been explored extensively (Beatty 2019; Raes et al. 2020) and so has the value of peer instruction (Porter et al. 2011), our fusion approach to intensive pair programming represents a novel application of these principles to collaborative coding education. With a team of four (three lecturers and one teaching assistant), we had the flexibility to experiment with innovative teaching strategies that bridged the gap between online and on-site students. This chapter presents our experiences and lessons learned when experimenting with three key interaction methods: *Negotiating Fusion-Pairs*, *the Ghost Helper Effect* and *Micro-interactions*.

7.2 Making Fusion Pairs Work

A reflection on what needs to be considered when pair programming in a fusion classroom

Pair programming is central to our course (L. Williams et al. 2000; Hannay et al. 2009; Hanks et al. 2011), but pairing students effectively in a fusion setting presented its challenges. We experimented with assigning students in same-mode pairs (on-site-on-site or online-online) and mixed-mode pairs (on-site-online), adjusting pairs every hour to help students feel part of a single, integrated cohort. Each new pairing required students to renegotiate key aspects of their collaboration: how they worked together, who took the lead, and how they interacted with instructors. This constant reshuffling reinforced the sense of a shared learning experience across both modes.

Figure 7.1: Figure 1: Different modes of paired students working together on programming exercises.

We encountered several challenges in this type of paired learning approach (audio issues, cognitive load, context switching). This could especially affect students when accessibility issues or other special circumstances are compounded or clash with the learning environment.

While students generally preferred mixed-mode pairs (see Figure 2) and reported feeling more connected to the broader cohort, this arrangement proved exhausting to both online and on-site participants. The background noise in the classroom, combined with the additional cognitive

load students experienced as a result of this noise, can leave them feeling drained by the end of a session.

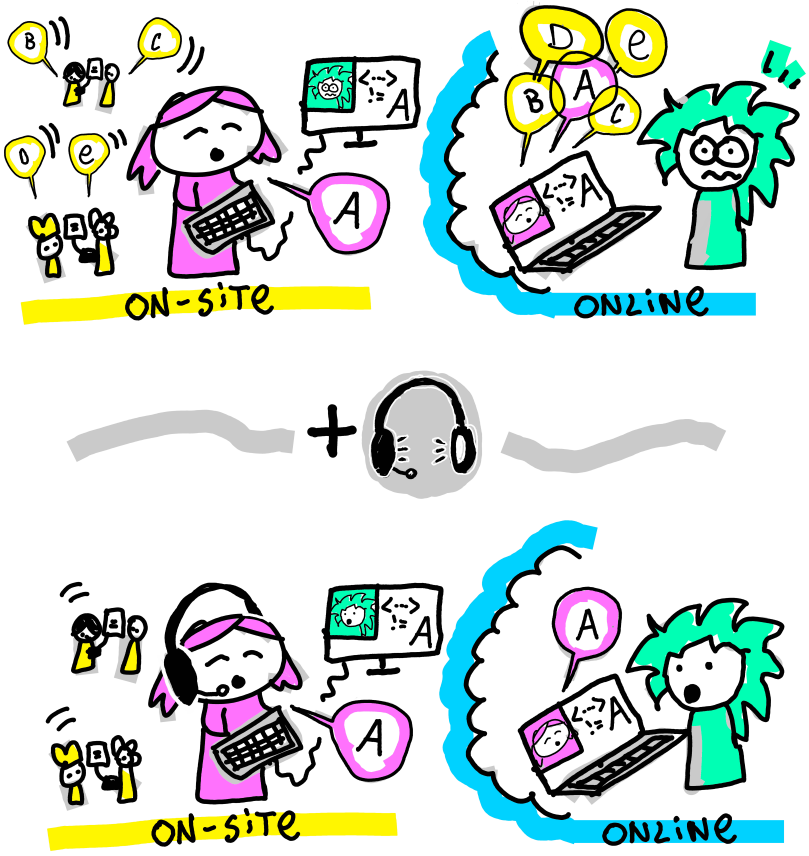

Figure 7.2: Figure 2: Mixed-mode pair programming in action with noise in background.

The size and acoustics of the classroom are key factors to consider when using pair programming in teaching. Background noise not only causes "Zoom fatigue" for online students (Bailenson 2021), especially in longer events like ours, but also affects the on-site students who wear headphones to connect to their mixed-mode buddy. Such overload issues disproportionately affect neurodiverse learners who may struggle with auditory processing or filtering competing stimuli. In our case, additional room facilities or even a quiet area outside of the classroom proved handy for such interactions. As a teaching team, we hadn't originally anticipated this need. We are known to improvise and have once used a "broom cupboard" as an overflow space to support mixed-mode pairs.

Unlike typical online interactions, where participants can selectively mute themselves, paired programming also demands constant verbal communication as students alternate between navigator and driver roles throughout exercises. This requirement for continuous dialogue

means that small audio cues (*"aha," "umm"*) become important elements that would make the experience worse if they were missing. The downside of not muting ourselves is that background noises cannot be avoided.

As teachers, we had to learn what the experience of an online participant is. When joining students online, we noticed how important the camera angles are. Online students like to see teachers close up but also what's happening in the classroom. When teaching hearing-impaired students online, they like to see the face of the lecturer close up for lip reading. When looking at code on a screen, the font size needs to be sufficiently big, and it is important to scroll in a controlled way, as otherwise it becomes difficult for the partner to follow. Often, the first few minutes for each pair's collaboration were taken up by making all those audio/visual adjustments, and then they were ready to go.

With several mixed-mode pairs in one room, there might also be a feedback loop (which sounds like a high-pitched squeak when a microphone picks up the sound from the speakers). This is something that needs to be managed carefully, and, again, additional physical space will help with that. In an hour-long session, such adjustments can take up another few minutes to sort out.

More broadly related to fusion teaching, addressing and talking to online students explicitly during the introduction of the course and throughout, and not just the ones, is vital to make them feel part of the group, as it is very easy to forget about them otherwise. Having personally participated in hybrid courses that prioritised those in the physical classroom, the online experience can feel isolating. Inertia and gravity pull teachers towards excluding online participants, if there are active efforts and structures to include them. For our text mining course, we rehearsed all possible scenarios of interacting with on-site and online students in the bigger group or in pairs (e.g. how do I interact with a student pair that is mixed-mode, on-site-only or online-only) and really thought carefully about what is important in each situation. We use an advocate for the online students in the room, a person with specific responsibility, who could be another teacher, a teaching assistant, or a student, to draw attention to any comments or raised hands from an online student that the teacher may have missed. This has the dual effect of including the online students in that moment but also reminding the teacher to check in the future. Making space to ask the online students for their comments is important; they become more active participants, and the students in the room feel like they are part of the cohort.

After several years of practice teaching using pair programming in this and similar courses, we are tending more towards online-only and on-site-only pairs, given that some of the challenges involved (e.g. classroom setup) are out of our control and are not something we can easily adjust during the course. In some ways, this feels like a failure; we really wanted to completely cross that boundary, but if this makes it harder for the students, we must adapt. We found that running the mixed-mode sessions only in the morning, when everyone is fresh, allows us to feel like one group and reduces the likelihood of Zoom fatigue later in the day.

7.2.1 Ghost Helper Effect and the Fluid Divide

A discussion on a fluid divide and not siloing students into two groups

In a traditional classroom, students can simply raise a hand and ask the teacher for help. However, in a fusion course environment, mixed-mode pairs introduced a new dynamic. We quickly learned that if online students needed help, support also had to come online. Otherwise, teachers ended up only communicating with people in the room. To address this, all teaching staff were online during pair programming sessions, creating what we called the "ghost helper effect". This allowed us to speak to students in both modes simultaneously, whether through direct conversation in the room or via webcams and headsets to ensure an equal experience.

This led to a surprising phenomenon for on-site students in mixed-mode pairs, suddenly hearing a voice in their headphones from one of the teaching team they had just interacted with in person a few minutes earlier, like a friendly ghost. At first, this unexpected presence was disorientating, diverting attention and requiring a cognitive shift (see Figure 3). To normalise the experience, we consciously used online helpers for all mixed-mode pair work. While initially some students were spooked by the fluidity of the "ghost" seeming to be everywhere at once, they quickly found this both friendly and helpful.

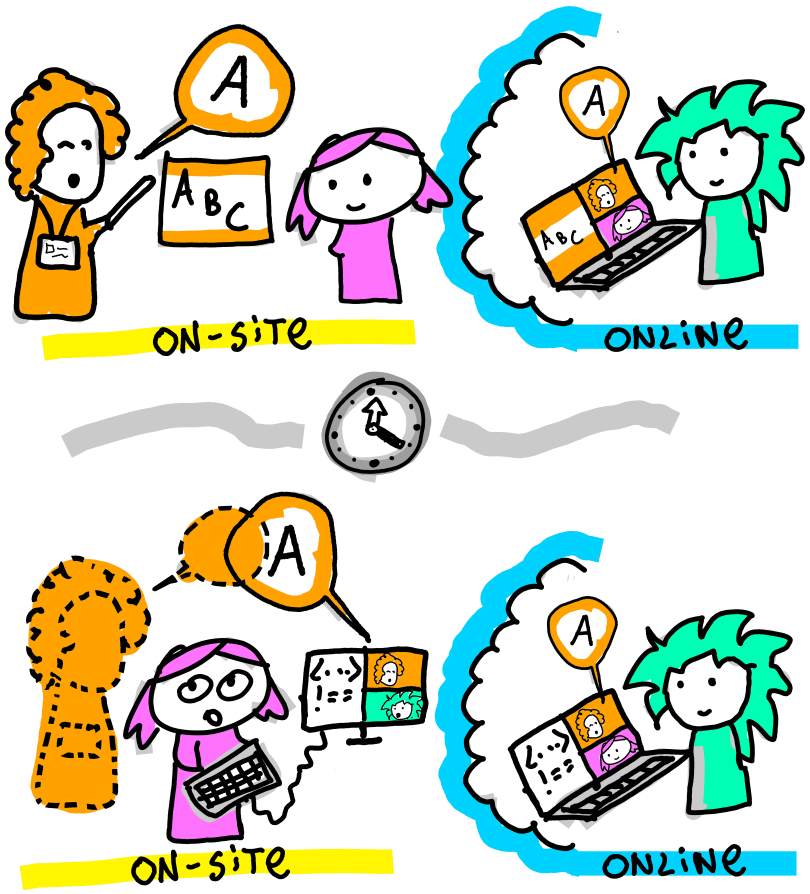

Figure 7.3: Figure 3: The ghost helper effect.

In taking this approach, it was important for students to feel a sense of parity in how the course was being delivered and to develop a fluidity between online and on-site. With staff being online and visible through their own cameras, they could speak to both groups directly through the same method. Rather than speaking to an online student through an on-site partner's webcam or relaying information to be passed on, this ensured that all information was delivered simultaneously and created a sense of fluidity between students and staff.

This was strengthened by staff at times leaving the main classroom and speaking to students in different locations, for example, a smaller classroom. They effectively became 'online' as well, working to break down the divide between the two modes, while still communicating with students together. In blurring the boundaries, neither group felt staff were giving preference to one side or the other, and the "ghost helper" approach allowed us to support students seamlessly regardless of location.

For students, this fluidity created a sense of seamlessness between the two media, and while

much thought went into facilitating this, for the students, it felt effortless. Some even shifted between modes during the course, starting the day online before coming on-site or the other way around. Allowing this flexibility showed that students felt at ease in either mode and that, while both had specific issues, neither was better than the other. For instance, at one point, electrical issues on-site meant that students in the room were unable to access their workbooks, while those online were unaffected. Therefore, while both modes could present challenges, in creating flexibility between them, students were able to adapt with little disruption.

7.3 Micro-interactions

Micro-interactions are how we bring the learning into the wider classroom

When you teach a course in an intensive mode, you are all together for two very busy days. This helps with a feeling of connection and that we are all in this together. It exacerbates feelings of joy but also of frustration. Learning happens in the midst of these feelings, and we know we need to connect the course with those small moments of joy and frustration and allow them to be voiced. We wanted to encourage, celebrate, and capture these moments of learning and cohort camaraderie, and quickly fix mistakes we were making and habits that were annoying.

Below we will explain strategies we employed to make the classroom a place of sharing, exchange and growing together. Feedback was gathered not only for quality control, as with the typical "How did you like the course?" questions, but rather to continuously synchronise with the energy and mood of all the students on the course; we call this 'relentless feedback'. It has to be quick and fun and not distract the class from the core learning. We gamified these small moments of interaction, with methods for ensuring everyone has an equal and anonymous voice, avoiding some common techniques which can be quite exclusionary, such as when a teacher picks the student who raised their hand first, it is often the same student. We asked students to put crosses on a whiteboard of questions to show where their mood was, and we asked them to fire answers to verbal questions in the chat function of the online room; we call this a 'Chat Blast'. If a student picked up a mistake in the code, we asked them to circulate it to the room. If they had a good comment or insight during their learning, we asked them to bring it back up in the discussion sessions when we came back together as a whole group after pair programming. We adapted sessions when people were frustrated or tired, we voiced that these feelings were valid and uncomfortable but can be part of learning, and we upped our energy when the energy in the room started to fade. We created a classroom in which both teachers and students can be vulnerable and open, so that we can grow from each other's mistakes, and communicate with respect (amplification of learning).

7.3.1 1) Relentless Feedback

We wanted to gather feedback as we went to have a quick and real-time reading on how the course is going and to be able to adjust our approach if needed. There is a trade-off between asking feedback on the spot or later: asking during the learning process can be quite disruptive, while if we ask later, students might not take the time to provide the feedback in the first place or might not remember how they felt earlier. This is especially important if we care about granularity and would like to know if, for example, a particular part of the course needs to be improved.

We envisaged a two-way feedback channel, where students say how they are feeling, but they also see how everyone else in the room is doing. We were also aware that anonymity will lead to more honest and useful answers. Another part was that we wanted it to be playful and not very constrained by the digital tool, where the little artefacts like handwriting, colour choice or exact position on the screen can convey personality and individuality.

We needed a quick and unobtrusive way to gather feedback, which meant that we could use it multiple times during the day. We also wanted to ask meaningful questions, which will help us guide the course delivery (not just "Do you like it") since we were aware of the complex interaction between whether a course is 'hard' and 'useful' (where 'hard' is fine, as long as it is 'useful').

We wondered whether we could create a feedback mechanism which is easier to complete than to dismiss/avoid, sort of like 'accept cookies' popups on websites make it easier to accept than to customise. So, we created a multiple-slider feedback mechanism, where students grade on a Likert scale multiple aspects of their learning experience, and all results are fully open but anonymous.

For our fusion course, we could not find a tool which would deliver exactly what we needed at the time, so we used a shared digital whiteboard. The whiteboard would always start with several empty scales, and each student would be invited to type the letter X in the appropriate place on the whiteboard (see Figure 4). Since we gathered this feedback every 1-2 hours of the course (after each badge), we had very granular data about the two days of our course. In the animation below, you can see the way the mood in the room changed as the course progressed, which was very useful for us to see and respond to.

Figure 7.4: Figure 4: Feedback collected at the end of each badge.

For completeness, this technique is not limited only to online or hybrid environments, as we also tried it on in-person bootcamp courses as a means of gathering and sharing feedback in the room (Figure 5).

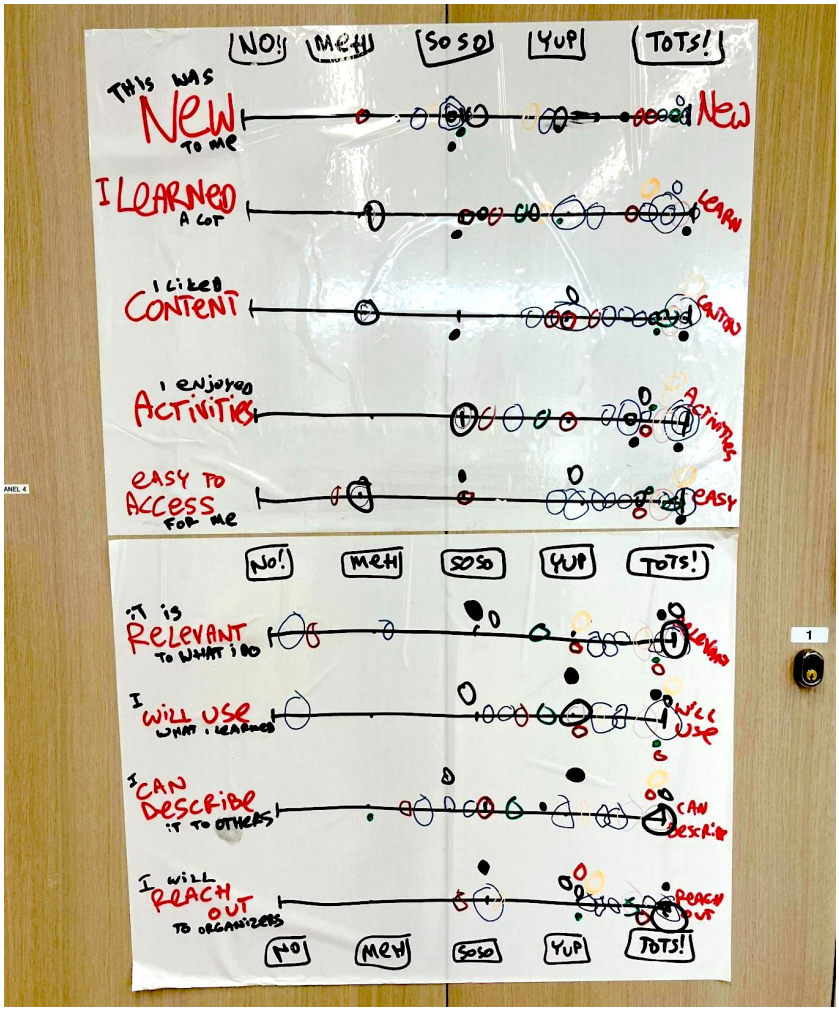

Figure 7.5: Figure 5: Feedback collected during a one-day intensive course on Hands-on Data Visualisation within Edinburgh Medical School at the University of Edinburgh.

7.3.2 2) Chat Blast and Other Micro-contributions

Following the same philosophy as in the feedback above, we wanted to give students opportunities to check in with their own learning and the group and provide constructive feedback to the teaching team. Throughout the course, there were moments where we asked them to contribute questions that we would then discuss in the session after each pair programming exercise. This enabled the entire cohort to benefit from the discussion and answers provided. We also ensured that all student questions were repeated aloud during debrief sessions, while also encouraging students to share their own solutions via the chat.

At specific points in the course, we also asked all students to type a quick, micro contribution

into the chat all at the same time, i.e. a Chat Blast (see Figure 6), to encourage engagement and shared participation. We took the Three Stars and A Wish approach, which is used by teachers in primary schools to help pupils focus on what is working and what needs improving (Larson, Trees, and Weaver 2008). In our case, we flipped this around and asked students to provide us with "Three stars and a wish" about the course in the chat of the video call. This enabled them to share their thoughts not only on what they liked about the course but also gave them permission to point out things that could be improved or things they struggled with.

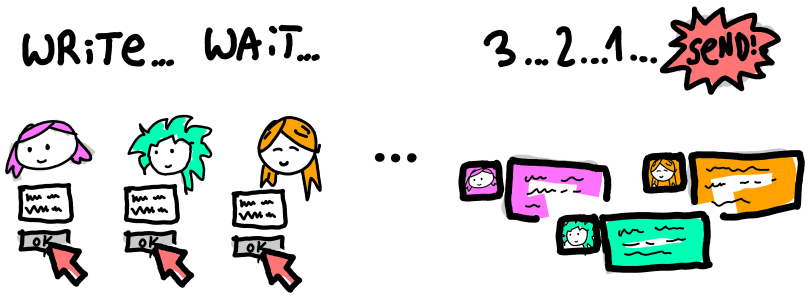

Figure 7.6: Figure 6: Chat blast with feedback using the "Three stars and a wish" method.

Using the Chat Blast method, we exposed the fact that we were learning from the students as well. We were constantly adjusting the course in an agile way according to the feedback received from students when possible. We also exposed the fact that not everything goes right all the time, that we are human beings who can make mistakes and are learning how to adapt to each new group of students in a fusion teaching environment.

7.4 A Practical Guide for Fusion Educators

After two intensive days in the trenches of fusion teaching, we emerged with battle-tested insights that we wish we had known before we started. Here are our concrete recommendations for educators considering teaching pair programming in a fusion (or hybrid) setting:

Before You Begin:

- **Rehearse every scenario.** Practice switching between online and on-site interactions until it becomes second nature. We literally role-played "How do I help a mixed-mode pair when one student is confused?"

- **Audio is everything.** Invest in quality headsets and microphones for all teaching staff and test for feedback loops before the course begins. Poor audio will sink your fusion

dreams faster than any pedagogical mistake.

- **Designate an online advocate.** Assign one person the specific job of watching for raised hands, chat messages and overlooked online participants. Without this, online students become invisible.

During the Course:

- **Start mixed, then adapt.** Begin with mixed-mode pairs when energy is highest, but don't be afraid to switch to same-mode pairs if fatigue sets in. Flexibility is not failure; it is responsive teaching.

- **Feedback must be quick and visual.** Our slider system took 30 seconds to complete and showed results immediately. Long surveys kill momentum.

- **Normalise the weird moments.** When your voice appears "ghost-like" in someone's headphones, acknowledge it with humour. Students need permission to find fusion teaching delightfully strange.

Technical Essentials:

- **Font size matters.** What looks readable to you will be too small for online partners watching code.

- **Avoid unnecessary scrolling.** Every rapid scroll or sudden page jump becomes disorienting for online partners trying to follow along. Move deliberately through content.

- **Camera angles are crucial.** Online students want to see both your face and the classroom; position cameras accordingly.

- **Have a backup space ready.** Whether it is a smaller room, or yes, even a "broom cupboard," you'll need overflow space for mixed-mode pairs.

The hard truth is that some days, same-mode pairs will work better than mixed-mode ones. Accept this not as defeat, but as responsive teaching. The goal is not perfect fusion; it is creating a cohort that learns together, regardless of location.

Our most important lesson? Fusion teaching requires the same vulnerability we ask of our students. When we admitted our mistakes in real-time and adjusted together, that's when the real learning, for all of us, began.

7.5 Conclusions

Fusion teaching challenges traditional classroom dynamics, but with thoughtful design, it can create an engaging and collaborative learning experience. By experimenting with different interaction strategies, we found ways to bridge the physical and digital divide between online and in-person students, creating an interactive and integrated cohort. Our lessons learned and practical insights are timely and valuable for educators designing hybrid courses, particularly in the post-pandemic education landscape.

Fusion teaching challenges traditional classroom dynamics, but with thoughtful design, it can create an engaging and collaborative learning experience. By experimenting with different interaction strategies, we found ways to bridge the physical and digital divide between online and in-person students, enabling effective pair programming partnerships across both modalities. The mixed-mode pairing approach, while challenging to implement in typical classroom environments due to background noise and cognitive load issues, demonstrated the potential for bridging physical and digital collaboration in programming education.

When logistical constraints made mixed-mode pairing difficult to sustain, our experience still provided valuable insights into preparing students for the increasingly common reality of distributed software development teams. Our lessons learned and practical insights are timely and valuable for educators designing hybrid programming courses, particularly in the post-pandemic education landscape, where graduates from various disciplines increasingly need coding and remote collaboration skills in their professional work.

8 Overcoming coding anxiety: learnings and strategies

8.1 Introduction

Programming or coding anxiety is defined as 'a psychological state engendered when a student experiences or expects to lose self-esteem in confronting a computing situation' (Connolly, Murphy, and Moore 2009). Coding anxiety may be driven, in part, by other anxieties such as computer anxiety (Chua, Chen, and Wong 1999; Meinhardt-Injac and Skowronek 2022) or statistics anxiety ((Zeidner 1991), amongst others. One may assume that "digital natives" (Bennett, Maton, and Kervin 2008) experience a lower level of computer anxiety; this assumption may not naturally translate to coding anxiety (Nolan and Bergin 2016). Indeed, computer anxiety could be attributed to, for example, variable exposure to coding and an over-reliance on connected digital ecosystems during childhood (*How's Life for Children in the Digital Age?* 2025). The attribution of coding anxiety, however, is a more nuanced issue. Different levels of coding anxiety for mathematics and biomedical undergraduate students exist (Miller and Pyper 2024), with the former finding coding to be more enjoyable, and hence having lower anxiety levels, than the latter. More broadly, student identity and background can also contribute to anxiety-linked barriers to learning, such as gender (Forrester et al. 2022) or learning in a non-native language (Kaur and Newell 2024).

Given that coding anxiety can be attributed to a variety of sources (PAN and Harun 2025), here we draw on our collective experience of teaching coding, provide perspectives on coding anxiety, and offer strategies to mitigate it. We focus specifically on students for whom coding is not their core academic focus, but rather a necessary tool encountered when taking classes involving statistics, data analysis, experimental design, and modelling. It is possible that these students have had no prior exposure to coding, and their initial interactions will define their longer-term relationship with coding, and as a consequence, how they navigate an increasingly data-centric world where coding is an attractive, liberating, and increasingly essential skill.

8.2 Recognising and Refocusing Anxiety

Anxiety sustains itself through a self-perpetuating cycle (Mkrtchian et al. 2017). It is believed to start with a trigger (e.g., a bad experience), which leads one to avoid future associated

triggers in an attempt to feel some short-term relief (Hofmann and Hay 2018). However, such avoidance can lead to heightened anxiety in the long term (Mkrtchian et al. 2017). Since the brain learns that avoidance temporarily decreases anxiety, even if only momentarily, it may eventually prevent an individual from engaging with the trigger entirely. If the trigger is computer code, then coding becomes something to avoid, and hence coding anxiety is born. In contrast, confronting the trigger can build confidence and reduce anxiety over time (Kampmann, Emmelkamp, and Morina 2018).

We believe that recognising learners' coding anxieties upfront is an important first step towards demystifying it. Once students realise that this is an identified problem common to many, rather than something affecting them alone, they are one step closer to overcoming it. Humour, including self-deprecating jokes, describing how, despite now being the teacher, you once suffered from coding anxiety, can go a long way towards making students feel better. Cartoons may also be an effective way of making anxiety itself less daunting (Figure 1). If existing online content is not enough, and making cartoons is not one's gift, AI-generated images could easily be created for that purpose - just make sure people have 5 fingers!

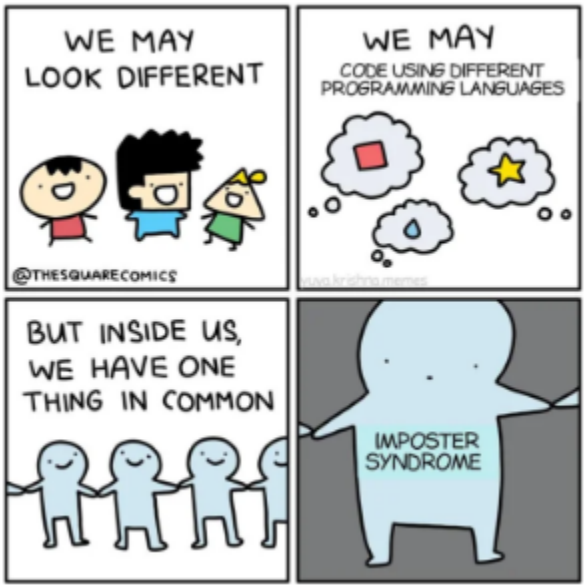

Figure 8.1: Figure 1. An example of a humorous cartoon that normalises coding anxiety. (https://www.thesquarecomics.com/) TODO: attribute this properly and add alt-text

Given anxiety's negative feedback loop, the best way to combat anxiety might be to break this vicious cycle (Figure 2). As an instructor or teacher, this can be achieved by providing positive experiences with coding. Importantly, after acknowledging that coding anxiety is real, the solutions should be presented along with how, as a teacher, overcoming their anxiety will be supported. For example, reflecting on their personal experience, the teacher could describe

how they became at ease with coding. Such a role model could serve as a powerful catalyst for students to realise they can overcome their own coding anxiety too.

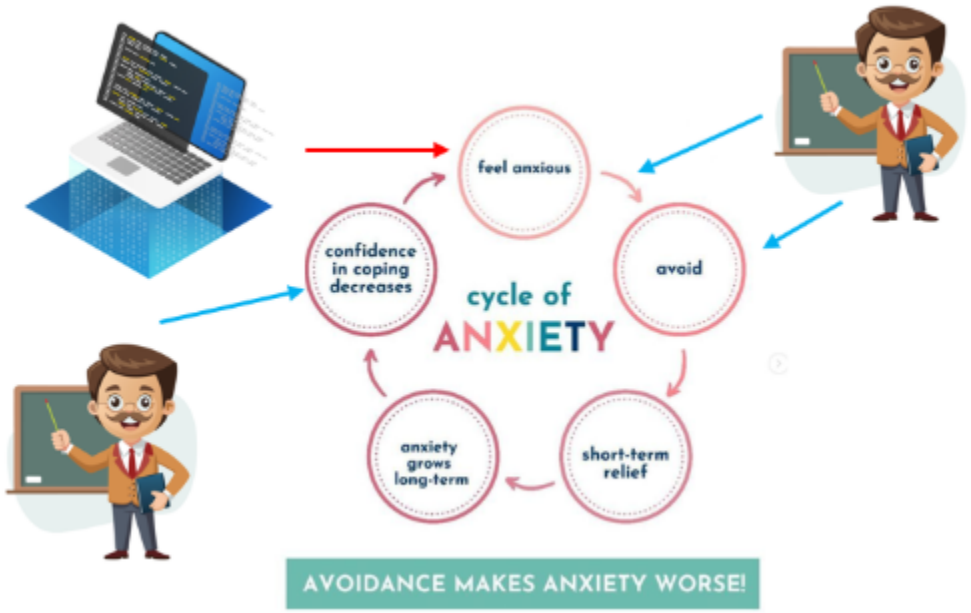

Figure 8.2: Figure 2. The cycle of anxiety. Coding is identified as the trigger and the teacher serves as a diffuser to intervene in multiple places and in multiple ways to break the cycle. Adapted from https://www.mentalyc.com/ TODO: Attribute this properly and add alt text

Gauging the class's perception towards coding should prove useful in lowering overall anxiety. Collecting information about a priori fears or causes of anxiety related to one's course, including - but not exclusively - coding anxiety, as well as course expectations, provides clues to what might be good strategies to adopt. In-class feedback tools like Mentimenter[1], allow students to provide feedback, anonymously and in real time, to make the course more accessible in an informal, friendly, and interactive way. This could generate useful ideas, including specific suggestions, for how to reduce anxiety. Additionally, having students reflect on how anxious they are about coding might allow them to identify shared problems and anxieties across the class.

[1] www.mentimenter.com

8.3 Building Confidence Through Active Learning

Early hands-on activities involving code, but not necessarily creating their own code, could prove beneficial by focusing on the end product rather than the means to get there. Even exercises where students simply copy-paste code, without knowing why it does what it does (but being told that later they will be able to code it themselves), could go a long way in reducing anxiety. Such exercises allow students to see how useful coding is for tasks they would like to do, but which would be impossible to do manually. Focusing on the outcomes students are interested in and can relate to makes them less reluctant to learn programming than if they just see it as a barrier. For example, one could take a couple of large datasets linked by a common variable, and ask students to examine the files in a text editor and "understand" the data. Then the students can be given code to produce a captivating visualisation (Figure 3) with relevant insights for the subject matter of their course. Focusing on the reward, rather than the trigger, can lower anxiety (Xiao et al. 2022) and hopefully inspire students to create similar outputs themselves.

TODO: missing image? please keep in mind naming convention

Fig. 3. Example of a beautiful visualisation created using R (a map of elephant movements through Kruger National Park). Produced by Dr Pratik Gupte (Imperial College London) and reused here with his permission.

Having fun reduces anxiety in a wide variety of contexts, so why not try it for coding anxiety, too? Coding need not be boring! There are many resources that one can use to showcase how coding can be fun. Having fun coding will mean that when coding becomes required, students' receptivity to it is increased. R packages like `memer` and `mermery` allow meme creation from within R, while generative art packages can produce abstract paintings from code (Figure 4). The R package fun makes games and activities accessible from within the R environment. As an example, the classic game Minesweeper can be played within R via a single line of code defining the size of the area and the number of bombs. All the above can be used to introduce, in an informal setting, the concepts of functions and arguments. At the end of a long coding session, or even during one, package praise can also come in handy. A call to its single function `praise()` returns positive feedback like "You are wonderful!" or "You are supreme!". These simple statements often put a smile on students' faces, and based on our experience, that is halfway towards reducing anxiety. Finally, if students just can't code but want to "fake it till they make it", the R function `look_busy()` in the `player` package will produce output on the console that will make anyone else believe students are working hard. We have used several of these approaches and provide them as a starting point, but given R is a dynamic and ever-evolving environment, other fun and engaging options must live out there waiting to be used by teachers to help reduce code anxiety.

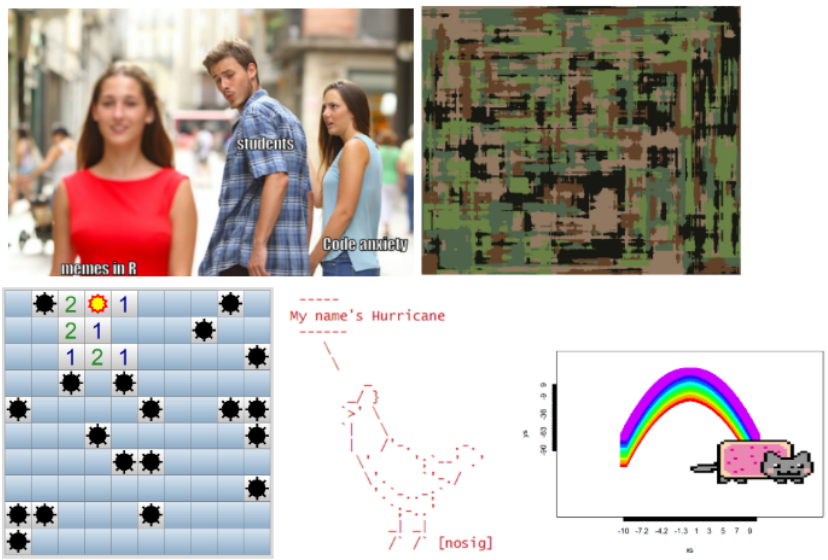

Figure 8.3: Figure 4. Examples of how R coding can be fun. Top-left: A meme showcasing how memes can reduce code anxiety; this meme can be recreated with 2 lines of code using package "memer". Top-right: A painting created in "aRtsy", a generative art R package. Bottom-left: An instance of a minesweeper game created in package "fun", demonstrating the authors inability to play it. Bottom-centre: An example of an animal that can be drawn in R syntax using the "cowsay" package. Bottom-right: The "CatterPlots" package can be used to generate scatterplots that include cats.

Live coding can be one of the most effective tools to demystify programming and reduce coding anxiety. When teachers write and run code in real time, narrating their thought process as they go (and getting error messages), they model both the practice and the mindset of coding. Students witness that even experienced coders make mistakes, encounter errors, and debug them systematically. This transparency transforms coding from a mysterious, high-stakes activity into a visible process of exploration, reasoning, and iteration.

Importantly, the approach taken during live coding matters as much as the content. Speaking aloud while typing, eg, explaining why particular commands are chosen, predicting outputs before running them, and pausing to invite student predictions, turns what might otherwise be a passive demonstration into an interactive learning experience. When an error appears, embrace it as an opportunity to model problem-solving, showing that mistakes are normal, and even useful. Seeing a tutor respond to an error message with curiosity rather than panic can profoundly reframe students' emotional response to failure.

To lower anxiety further, live coding should begin with simple, visually rewarding tasks. Plot a small dataset, calculate a mean, or clean a short table. This way, students can follow the logic without feeling overwhelmed by syntax. Gradually, the exercises can build in complexity, moving from replication to modification and finally to creative application. When possible,

students can be invited to suggest the next line of code, predict what will happen if a parameter is changed, or even take over the keyboard for short stretches.

Finally, recording or providing annotated versions of live coding sessions allows students to revisit material at their own pace. This ensures that the spontaneity of live teaching is complemented by the reassurance of reviewable, structured notes. By combining openness, interaction, and a healthy acceptance of mistakes, live coding transforms what might otherwise be a source of anxiety into a collective, confidence-building experience.

There can be multiple approaches to teaching code and statistical data analysis, and some of these might improve or contribute to coding anxiety (link to chapter Sequential vs. Simultaneous: Approaches to Learning Programming and Statistics). Common wisdom also says that the first step is the hardest for any journey. In our experience, if students start with small, manageable, and clearly described tasks, their initial experience with coding will be positive, which will decrease anxiety levels when facing harder tasks. Therefore, we suggest that students are first given easy-to-follow tasks with step-by-step instructions, allowing confidence to grow. This is consistent with the strategy suggested by Auker and Barthelmess (2020). Later classes or assignments may contain less detailed guidance or be more open-ended, allowing students to explore their newfound confidence in their coding abilities at their own pace.

8.4 Support and Collaboration Beyond the Classroom

Coding confidence often grows most effectively outside formal teaching time. Low-pressure practice, whether collaborative or individual, helps students reinforce skills and approach coding with curiosity rather than fear. Teachers can support this by structuring opportunities for independent study and peer interactions that sustain progress between classes. One simple but effective technique is to encourage students to maintain a short 'coding diary', allowing for normalisation of occasional frustration and celebration of key achievements.

The fear of revealing a lack of coding ability can be a powerful barrier, discouraging students from even attempting new exercises and progressing in their learning. Independent work outside the classroom, supported by additional materials, can help overcome this. Providing a list of such resources will help students realise that they can learn on their own and that it will give them a head start. In our experience, a particularly interesting resource comes via the R package `swirl`[2]. Swirl "teaches you R programming and data science interactively, at your own pace, and right in the R console!". Contents can be explored independently, or by following the prompts of a 'virtual tutor' (link to generative AI chapter). Students are amused by a virtual tutor calling them by their name, meaning that they engage with the learning experience in a positive mood.

[2] https://swirlstats.com/

Sharing the learning journey with others can be equally powerful. The mantra "a problem shared is a problem halved" can be operationalised while teaching programming via pair-programming (Figure 5 + link to other chapters). Pair-programming might be a useful strategy to lower coding anxiety levels **TODO missing REF Brougham et al. 2020**. Nonetheless, these are not universal findings; see (Krizsan and Lambic 2024) for a case where performance was apparently improved, but without a decrease in anxiety levels. Online pair-programming might be less intimidating for some students, although it is not clear whether the benefits associated with live pair-programming apply (Hafeez et al. 2023).

AI-assisted pair-programming, where a coder is paired with an AI agent, has become an accessible alternative. It could be used to the same effect as human-human pair-programming (Fan et al. 2025). This might be easier for students who would otherwise be reluctant to expose their difficulties to a colleague. Nonetheless, while AI-assisted pair-programming has been shown to enhance motivation, reduce anxiety, and even improve performance, it does not fully match the collaborative depth and social presence achieved through human-human pairing (Fan et al. 2025). **TODO: is this meant to be the same reference?** Regardless of the pair-programming mode considered, we believe that if well thought out and prepared, pair-programming activities can be used to lower coding anxiety levels in a range of students.

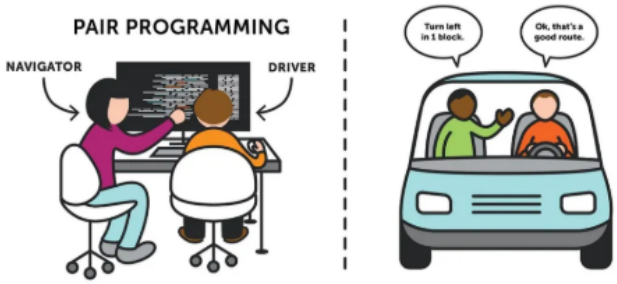

Figure 8.4: Figure 5. Pair-programming can help lower one's coding anxiety.

Students can be encouraged to further their learning journey by 'teaming up' with peers to continue to pair-programme outside of the classroom environment. It has been observed that this can assist with coding anxiety both through additional assistance in a familiar mode and also through the mechanism of shared endurance and a head-on approach to the anxiety barrier.

Ultimately, supporting learners beyond the classroom is about fostering autonomy and connection in equal measure. When students know they can access help, whether from peers, teachers, lecturers, or AI tools, they are less likely to interpret difficulties as personal failings. Instead, they begin to view coding as a shared and iterative process that need not come with coding anxiety but natural failure and success routines.

8.5 Key Takeaways

Coding anxiety is real, especially for students for whom coding is a core part of their disciplinary focus. This barrier should be recognised, and we have a responsibility and opportunity to normalise it in teaching settings and explain why and how the anxiety cycle can be broken. Overcoming coding anxiety should be presented as a personal endeavour, but one that instructors and peers, along with a growing number of resources, will support. As a teacher, effective methods for breaking the anxiety cycle are as pedagogically important as content creation. Moreover, you will become an anxiety diffuser, starting by exploring ways to provide rewards from coding activities. In particular, activities allowing students to see the potential of coding to achieve tasks they are inherently interested in should be considered. Providing encouragement and additional resources for learning how to code, and promoting additional high-rewarding coding activities, are likely to help further. Strategies to share the burden, like pair-programming, could further boost self-confidence in coding. Overall, different audiences and topics will require slightly different approaches, so engaging with students and intervening using strategies discussed above will be fundamental for reducing coding anxiety and improving learning.

9 Structured group work with assigned asymmetrical roles and switching - Lessons from Pair Programming across disciplines

9.1 Introduction

TODO: keep this as introduction, or ditch it?

In this chapter, we share our experiences with pair programming as a structured group work format that we believe fosters student engagement, well-being, and transferable skills. We argue that the same pedagogical benefits can also be realised when used for non-programming tasks.

9.2 What is "pair programming"? (and why should you care?)

In the software industry, people often work in pairs, which evolved over the years into a standard collaborative practice called "pair programming" (L. A. Williams 2010). Partners take turns playing two different roles: driver and navigator. The driver has exclusive control of the shared computer, and the navigator guides the driver by acting as their sounding board. Being in each role allows them to gain and contribute unique perspectives on the task at hand. Switching roles frequently allows both partners to get the most out of the experience.

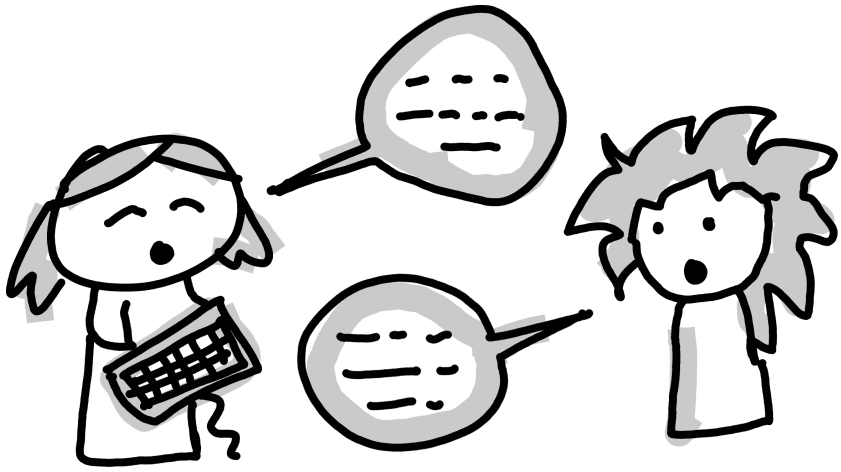

Figure 9.1: “I think your idea will work, let me type it in!”,” I see a bracket missing on line 7”, “Let’s run this code and see what happens.”, “Yay, we did it! It works!”

The purpose of the exercise is not only to produce better output (e.g., code) but also to: share knowledge and practice (cross-pollinate ideas); help each other to grow; create friendships and community; and foster a culture of being able to doubt and ask for help (you’re never struggling alone).

In the professional world, pair programming benefits the product, the individuals, and the team. When implemented in the classroom, students can reap those same benefits, with evidence of positive impact on learning experience and outcomes (Hanks et al. 2011).

We are a team of UoE educators who have been using pair programming as a group work protocol in our teaching for several years. We have done so across many subject areas (psychology, mathematics, statistics, data science, medicine, epidemiology, business, EFI); levels of study (UG and PGT); and delivery modalities (in person, online, hybrid). We have consistently seen the positive impact of this practice on our students and teaching teams.

In our view, the benefits come from the structured roles and the process, which are transferable to other production tasks beyond just writing code (Saltz and Heckman 2020). Indeed, any task where a group of people create an artifact (e.g., writing, weaving, drawing, problem-solving) could apply the same collaboration principles.

9.3 Structured roles for active learning

Group work can be beneficial to learning, but navigating group dynamics requires effort, which can significantly worsen the cognitive load of learning something new. When roles are vague (or non-existent), communicating and negotiating responsibilities between group members can become an extra task in itself. For instance, when no-one in a team feels individually

responsible, everyone might wait for the others to take initiative, and nothing gets done; or, one student might assume a leadership role, leaving others who are less confident to follow passively or to disengage entirely.

As a structured way to work together (a "collaboration script" (Weinberger 2011)), pair programming gives groups a roadmap with prescribed roles for each student, and specific activities associated with each role. With the burden of inefficient group work lifted, students have more room to focus on learning. Furthermore, when clear roles are assigned, everyone has to take ownership of their part and actively contribute.

> "Making sure everyone takes turns being the driver means everyone needs to contribute." - UG, Psychology

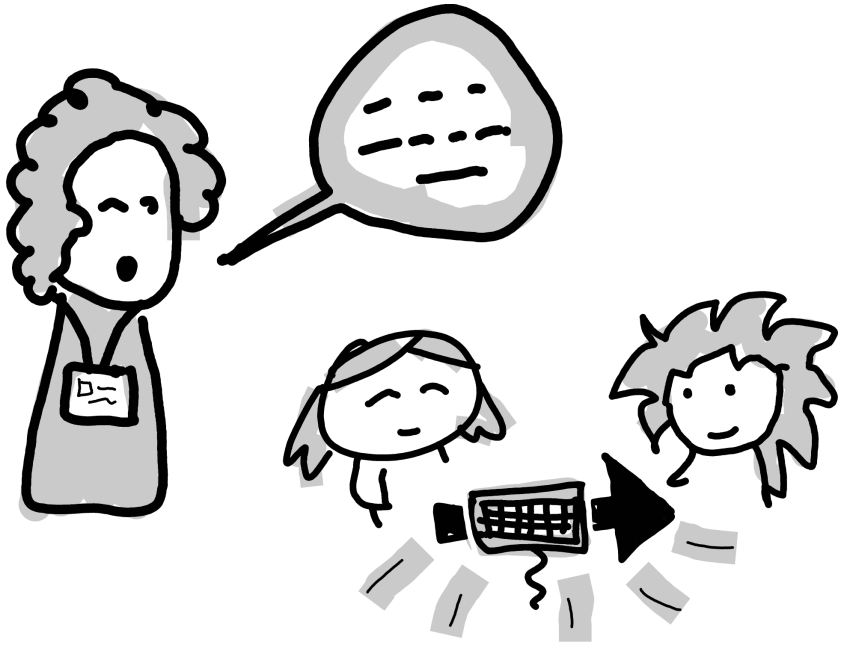

Figure 9.2: "Hello! Which of you is driving now?", "Is it time to switch drivers?", "Do you need help getting unblocked or are you just cracking on?"

As a result, we see students engage more. They take responsibility for their group, attend more sessions, and come better prepared to not let their teammates down.

> "Students were not engaging in the tutorial sessions at the beginning, which made them less useful. But it improved after having pair programming" - PGT, Medical School

Having defined roles doesn't leave room for any one student to take over the task or speak over others, as can sometimes happen in unstructured group work. This is particularly important

as many of our students have experienced marginalisation, and hence feel less safe putting themselves forward.

> "This is sort of just a problem with sexism, but I have really disliked group working in the past because I've been put with people (boys) who completely disrespect my work. This is why I'm opposed to the forced group working. It's better to work alone than be treated like that." - UG, Mathematics

Although the structured roles are not a magic bullet to prevent these issues, they can provide tools for both students and tutors to maintain a safer learning environment. One useful strategy is for tutors to casually check in with the groups, leading with the question: "who is the driver?". This does two things: first, it is a temperature check of whether the collaboration is working well; second, it reinforces the expectation that students should stick to the script at all times, even after the tutor leaves.

Figure 9.3: "What if I end up in a pair with someone who is mean, or much more advanced than me?", "I'm uncomfortable in socially complex situations. Will I know what to say and do in a group…", "Should I just always work with the same friend, or would I miss out on meeting new people, and learning new coding styles"

If a student is undermined by their partner, they can fall back on the script to reassert themselves as a valued contributor, or to communicate the issue with a tutor. This can reduce the emotional cost required to flag or confront bad behaviour directly, by giving both students and tutors a tool to defuse the situation.

9.4 Fostering peer learning and community

Pair programming provides plenty of opportunities for peer learning. The driver has to verbalise their specific ideas and problems, discuss them with the navigator, and synthesise the outcomes of their discussion to "put things on paper" and advance the task. The navigator must be an active listener, observer, and helper, and can take the leading role in higher-level strategic thinking. Both students are exposed to each other's approach, and learn from each other by cross-pollination.

> "The most useful part of [pair programming] is being able to collaborate with other classmates, which allows me to discover aspects I hadn't noticed in my own work through the questions raised by others." - PGT, Mathematics

> [A core learning outcome from the course was] Experience in pair programming and trying to explain concepts to another student. Sometimes it's quite easy to "see" how you'd do something yourself, but actually explaining this instead of doing it is more difficult, so the online sessions were very useful to practice this . I also found it very valuable to see how others work and how they understand code differently." - PGT, Medical School

Students also appreciate how discussing with a peer can generate new ideas. A pair programming environment is highly conducive to discussion, creating a free flow of ideas where the outcome is more than the sum of its parts. Since only one person can write things down, students must verbally negotiate a shared understanding of the agreed way forward.

> "We have different ideas, and co-operate with each other could generate more ideas." - UG, Psychology

> "Group discussions often have unexpected results." - UG, Psychology

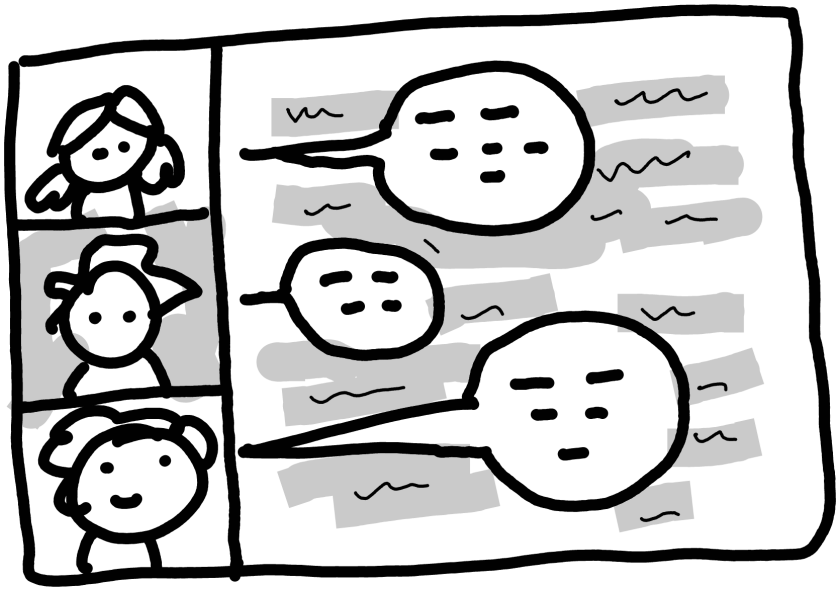

Figure 9.4: “So the way I understand it, is …”, “I see another way to solve this!”, “That’s cool! I never thought about it this way”

Although the roles are asymmetrical at any particular point in time, taking turns ensures that all students benefit equitably from playing both roles over the course of a session or a semester.

> “The practice of being a driver and a navigator is very useful. These two different roles provide me with distinct perspectives on the code. When I’m the driver, I focus on how to complete the code, and when I’m the navigator, I gain insights into how my fellow students understand the code, which helps me learn new things.” - PGT, Mathematics

Beyond direct benefits in the classroom, we have also seen pair programming help build a sense of community among students and enrich their personal life at university. Students build a network of peers that they can turn to for support later. This is especially critical in distance learning, where students do not have the opportunity to naturally meet peers (e.g., outside of class).

> “The group work is very helpful. Not only have I made new friends, but I now also have people that I can rely on when I need help. Group work has also made the individual workload seem much lighter and we get things done faster.” - UG, Psychology

> “[It] was nice to get some 1-1 interaction through the pair programming in what could have been an otherwise solitary course.” - PGT, Mathematics (delivered online in 2020/21)

There are many ways to assign students into pairs, which might influence the classroom experience and community building in different ways. For example, many students are more likely to attend class if they can work with the same friend each time, but this might mean that they miss out on meeting new people. Swapping partners each week will create a network of student-to-student connections which contributes to cohort building. Finally, stretching the definition of "pair" programming to groups of 3 or more (with a single driver) can also work well. Allowing for more navigators can create flexibility in terms of logistics (odd number of students), technological problems (connection issues in online teaching), student preferences, or accessibility.

> "It gives you the perfect opportunity to work with other people and make potential new friends." - UG, Psychology

> "[about using PP on 4 different courses] Thank you for the interactive pair programming sessions [across the academic year], they made a big difference in creating a community of online learners." - PGT, Medical School

9.5 Tackling issues as a community of practice

Many of our students have experience of dysfunctional group work, where different group members have different priorities, expectations, or skills. Pair programming can exacerbate this issue, because each student is expected to contribute actively to the joint work, whether they are driver or navigator. Inevitably, pairs will occasionally be incompatible, in a way which can undermine the effectiveness of peer learning. For instance, a very wide gap in technical proficiency or level of preparation can make for a frustrating experience for both students, leading to disengagement and even resentment.

> "I fully agree with the idea of group working, but in practise it usually results in people being left behind by those who understand better [...] and unfortunately in maths way too many people are mean and make the beginners feel really bad." - UG, Mathematics

Although the strategies discussed above can minimise this, there is no single correct solution. Whatever way you decide to implement groupwork, you will encounter issues, and your approach will need to be adapted to your discipline, classroom, or even cohort.

We have come together as a community of practice for mutual support. We have found it incredibly valuable as educators to discuss experiences, tackle challenges, and share examples of good practice.

9.6 This chapter was pair-programmed

Pair programming has worked in our programming classes, but we very much see the value of this kind of structured group work outwith a programming context. For instance, we "pair-programmed" the writing and editing of this very chapter over two sunny afternoons. We took turns being the driver typing on a laptop, while everyone else navigated, contributing their ideas and wording. We believe that the principles we have discussed are widely applicable across disciplines, and we look forward to finding out how colleagues adapt it in their classrooms.

Bibliography

Aho, Alfred V. 2012. "Computation and Computational Thinking." *The Computer Journal* 55 (7): 832–35.

Alex, Beatrice, Clare Llewellyn, and Pawel Orzechowski. 2026. "Teaching Programming Across Disciplines." In.

Alex, B., C. Llewellyn, P. M. Orzechowski, and M. Boutchkova. 2021. "The Online Pivot: Lessons Learned from Teaching a Text and Data Mining Course in Lockdown, Enhancing Online Teaching with Pair Programming and Digital Badges." In *NAACL-HLT 2021*, 138.

Anderson, Neil, Maria Angela Ferrario, Aidan McGowan, Matthew Collins, Jonathan W Browning, Leo Galway, Philip Hanna, David Cutting, and Darryl Stewart. 2025. "Learning to 'Think' Through Playful Interactions: A Play-Kit for Incoming First-Year Computing Students." In *2025 IEEE Global Engineering Education Conference (EDUCON)*, 1–3. IEEE.

Auker, Linda A., and Erika L. Barthelmess. 2020. "Teaching r in the Undergraduate Ecology Classroom: Approaches, Lessons Learned, and Recommendations." *Ecosphere* 11 (4). https://doi.org/10.1002/ecs2.3060.

Bailenson, J. N. 2021. *Nonverbal Overload: A Theoretical Argument for the Causes of Zoom Fatigue, Technology, Mind, and Behavior*. American Psychological Association.

Balreira, Dennis G., Thiago L. T. da Silveira, and Juliano A. Wickboldt. 2023. "Investigating the Impact of Adopting Python and C Languages for Introductory Engineering Programming Courses." *Computer Applications in Engineering Education* 31 (1): 47–62. https://doi.org/10.1002/cae.22570.

Beatty, B. 2019. *Hybrid-Flexible Course Design*. London, UK: EdTech Books.

Bell, Tim, Jason Alexander, Isaac Freeman, and Mick Grimley. 2009. "Computer Science Unplugged: School Students Doing Real Computing Without Computers." *New Zealand Journal of Applied Computing and Information Technology* 13 (1): 20–29.

Bennett, Sue, Karl Maton, and Lisa Kervin. 2008. "The 'Digital Natives' Debate: A Critical Review of the Evidence." *British Journal of Educational Technology* 39 (5): 775–86. https://doi.org/10.1111/j.1467-8535.2007.00793.x.

Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Shah. Viral B. 2017. "Julia: A Fresh Approach to Numerical Computing." 59: 65--98. https://doi.org/10.1137/141000671.

Bromage, Adrian, Sarah Pierce, Tom Reader, and Lindsey Compton. 2022. "Teaching Statistics to Non-Specialists: Challenges and Strategies for Success." *Journal of Further and Higher Education* 46 (1): 46–61.

Chin, Monica. n.d. "File Not Found: A Generation That Grew up with Google Is Forcing Professors to Rethink Their Lesson Plans (Internet Archive Link)." The Verge.

https://web.archive.org/web/20210922124725/https://www.theverge.com/22684730/students-file-folder-directory-structure-education-gen-z.

Chua, Siew Lian, Der-Thanq Chen, and Angela FL Wong. 1999. "Computer Anxiety and Its Correlates: A Meta-Analysis." *Computers in Human Behavior* 15 (5): 609–23.

Connolly, Cornelia, Eamonn Murphy, and Sarah Moore. 2009. "Programming Anxiety Amongst Computing Students—a Key in the Retention Debate?" *IEEE Transactions on Education* 52 (1): 52–56. https://doi.org/10.1109/te.2008.917193.

Curzon, Paul, Peter W McOwan, Nicola Plant, and Laura R Meagher. 2014. "Introducing Teachers to Computational Thinking Using Unplugged Storytelling." In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, 89–92.

Cutting, D. 2025. "The ProgBoard." https://thinklikeacomputer.org.

Denning, Peter J. 2017. "Remaining Trouble Spots with Computational Thinking." *Communications of the ACM* 60 (6): 33–39.

Desvages, Charlotte, Brittany Blankinship, Umberto Noè, and Pawel Orzechowski. 2026. "Teaching Programming Across Disciplines." In.

Fan, Guangrui, Dandan Liu, Rui Zhang, and Lihu Pan. 2025. "The Impact of AI-Assisted Pair Programming on Student Motivation, Programming Anxiety, Collaborative Learning, and Programming Performance: A Comparative Study with Traditional Pair Programming and Individual Approaches." *International Journal of STEM Education* 12 (1). https://doi.org/10.1186/s40594-025-00537-3.

Fearns, Josh, Lydia Harriss, and Clare Lally. 2023. "Data Science Skills in the UK Workforce."

Forrester, Chiara, Shane Schwikert, James Foster, and Lisa Corwin. 2022. "Undergraduate r Programming Anxiety in Ecology: Persistent Gender Gaps and Coping Strategies." *CBE—Life Sciences Education* 21 (2): ar29.

Gimenez, Olivier, Fitsum Abadi, Jean-Yves Barnagaud, Laetitia Blanc, Mathieu Buoro, Sarah Cubaynes, Marine Desprez, et al. 2013. "How Can Quantitative Ecology Be Attractive to Young Scientists? Balancing Computer/Desk Work with Fieldwork." *Animal Conservation* 16 (2): 134–36.

Goel, Sanjay, and Vanshi Kathuria. 2010. "A Novel Approach for Collaborative Pair Programming." *Journal of Information Technology Education: Research* 9 (1): 183–96.

Grizzle, Jessy. n.d. "Notes for Computational Linear Algebra." GitHub. https://github.com/michiganrobotics/ROB-101-Textbook-Computational-Linear-Algebra/tree/main.

Guest, Olivia, and Samuel H Forbes. 2024. "Teaching Coding Inclusively: If This, Then What?" *Tijdschrift Voor Genderstudies* 27 (2/3): 196–217.

Hafeez, Mustafa, Anand Karki, Yara Radwan, Anis Saha, and Angela Zavaleta Bernuy. 2023. "Evaluating the Efficacy and Impacts of Remote Pair Programming for Introductory Computer Science Students." In *Proceedings of the 25th Western Canadian Conference on Computing Education*, 1–7. WCCCE '23. ACM. https://doi.org/10.1145/3593342.3593351.

Hanks, Brian, Sue Fitzgerald, Renée McCauley, Laurie Murphy, and Carol Zander. 2011. "Pair Programming in Education: A Literature Review." *Computer Science Education* 21 (2): 135–73.

Hannay, J. E., T. Dybå, E. Arisholm, and D. I. Sjøberg. 2009. "The Effectiveness of Pair Programming: A Meta-Analysis." *Information and Software Technology* 51 (7): 1110–22.

Hofmann, Stefan G, and Aleena C Hay. 2018. "Rethinking Avoidance: Toward a Balanced Approach to Avoidance in Treating Anxiety Disorders." *Journal of Anxiety Disorders* 55: 14–21.

How's Life for Children in the Digital Age? 2025. OECD Publishing. https://doi.org/10.1787/0854b900-en.

Johnson, Fionnuala, Stephen McQuistin, John O'Donnell, and Quintin Cutts. 2022. "Experience Report: Identifying Unexpected Programming Misconceptions with a Computer Systems Approach." In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*, 325–30. https://doi.org/10.1145/3502718.3524775.

Kampmann, Isabel L., Paul M. G. Emmelkamp, and Nexhmedin Morina. 2018. "Does Exposure Therapy Lead to Changes in Attention Bias and Approach-Avoidance Bias in Patients with Social Anxiety Disorder?" *Cognitive Therapy and Research* 42 (6): 856–66. https://doi.org/10.1007/s10608-018-9934-5.

Kaur, Tarandeep, and Samantha Newell. 2024. "The Silent Struggle: Experiences of Non-Native English-Speaking Psychology Students." *Australian Journal of Psychology* 76 (1): 2360983.

Kelly, Martyn. 1992. "Teaching Statistics to Biologists." *Journal of Biological Education* 26 (3): 200–203.

Koulouri, Theodora, Stanislao Lauria, and Robert D Macredie. 2014. "Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches." *ACM Transactions on Computing Education (TOCE)* 14 (4): 1–28.

Krizsan, Tibor, and Dragan Lambic. 2024. "Examining the Impact of Pair Programming on Efficiency, Motivation, and Stress Among Students of Different Skills and Abilities in Lower Grades in Elementary Schools." *Education and Information Technologies* 29 (18): 25257–80. https://doi.org/10.1007/s10639-024-12859-w.

Larson, K. A., F. P. Trees, and D. S. Weaver. 2008. "Continuous Feedback Pedagogical Patterns." In *Proceedings of the 15th Conference on Pattern Languages of Programs*, 1–14.

Luxton-Reilly, Andrew, Simon, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, et al. 2018. "Introductory Programming: A Systematic Literature Review." In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, 55–106.

Lye, Sze Yee, and Joyce Hwee Ling Koh. 2014. "Review on Teaching and Learning of Computational Thinking Through Programming: What Is Next for k-12?" *Computers in Human Behavior* 41: 51–61.

Meinhardt-Injac, Bozana, and Carina Skowronek. 2022. "Computer Self-Efficacy and Computer Anxiety in Social Work Students: Implications for Social Work Education." *Nordic Social Work Research* 12 (3): 392–405.

Metz, Anneke M. 2008. "Teaching Statistics in Biology: Using Inquiry-Based Learning to Strengthen Understanding of Statistical Analysis in Biology Laboratory Courses." *CBE—Life Sciences Education* 7 (3): 317–26.

Miller, Ainsley, and Kate Pyper. 2024. "Anxiety Around Learning r in First Year Undergraduate Students: Mathematics Versus Biomedical Sciences Students." *Journal of Statistics and*

Data Science Education 32 (1): 47–53.

Mkrtchian, Anahit, Jessica Aylward, Peter Dayan, Jonathan P Roiser, and Oliver J Robinson. 2017. "Modeling Avoidance in Mood and Anxiety Disorders Using Reinforcement Learning." *Biological Psychiatry* 82 (7): 532–39.

Mustafa, R Yilmaz. 1996. "The Challenge of Teaching Statistics to Non-Specialists." *Journal of Statistics Education* 4 (1).

Nolan, Keith, and Susan Bergin. 2016. "The Role of Anxiety When Learning to Program: A Systematic Review of the Literature." In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 61–70.

O'Hara, Robert B. 2016. "On Teaching Ecologists Contemporary Methods in Statistics." Wiley Online Library.

Orzechowski, Pawel, Brittany Blankinship, and Kasia Banas. 2026. "Teaching Programming Across Disciplines." In.

Orzechowski, Pawel, and Bea Alex Elaine Mowat Clare Llewellyn. 2026. "Teaching Programming Across Disciplines." In.

PAN, YING, and Jamalludin Harun. 2025. "Conquering Coding Fears: A Systematic Review of Programming Anxiety in Higher Education." *Journal of Information Technology Education: Research* 24: 020.

Papert, Seymour A. 2020. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic books.

Pletzer, Belinda, Guilherme Wood, Korbinian Moeller, Hans-Christoph Nuerk, and Hubert H Kerschbaum. 2010. "Predictors of Performance in a Real-Life Statistics Examination Depend on the Individual Cortisol Profile." *Biological Psychology* 85 (3): 410–16.

Porter, L., C. Bailey Lee, B. Simon, Q. Cutts, and D. Zingaro. 2011. "Experience Report: A Multi-Classroom Report on the Value of Peer Instruction." In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, 138–42.

Raes, A., L. Detienne, I. Windey, and F. Depaepe. 2020. "A Systematic Literature Review on Synchronous Hybrid Learning: Gaps Identified." *Learning Environments Research* 23 (3): 269–90.

Robins, Anthony, Janet Rountree, and Nathan Rountree. 2003. "Learning and Teaching Programming: A Review and Discussion." *Computer Science Education* 13 (2): 137–72.

Saltz, Jeffrey, and Robert Heckman. 2020. "Using Structured Pair Activities in a Distributed Online Breakout Room." *Online Learning* 24 (1): 227–44.

Sentance, Sue, and Andrew Csizmadia. 2017. "Computing in the Curriculum: Challenges and Strategies from a Teacher's Perspective." *Education and Information Technologies* 22 (2): 469–95.

Sharifi, Serveh, Ruini Qu, and Stuart King. 2026. "Teaching Programming Across Disciplines." In.

Singer, Jeremy. 2020. "Notes on Notebooks: Is Jupyter the Bringer of Jollity?" In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. https://doi.org/10.1145/3426428.3426924.

Singer, Jeremy, and Steve Draper. 2025. "Let's Take Esoteric Programming Languages

Seriously." In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 213–26. Onward! '25. ACM. https://doi.org/10.1145/3759429.3762632.

Stewart, Graeme Andrew, Moreno Briceño, Alexander, Gras, Philippe, Hegner, Benedikt, Hernandez Acosta, Uwe, Gal, Tamas, Ling, Jerry, et al. 2025. "Julia in HEP." In *EPJ Web Conf.*, 337:01266. https://doi.org/10.1051/epjconf/202533701266.

Sweller, John. 1988. "Cognitive Load During Problem Solving: Effects on Learning." *Cognitive Science* 12 (2): 257–85.

———. 2018. "Instructional Design." In *Encyclopedia of Evolutionary Psychological Science*, 1–5. Springer.

Tedre, Matti, and Peter J Denning. 2016. "The Long Quest for Computational Thinking." In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research*, 120–29.

Tshukudu, Ethel, Quintin Cutts, and Mary Ellen Foster. 2021. "Evaluating a Pedagogy for Improving Conceptual Transfer and Understanding in a Second Programming Language Learning Context." In *Proceedings of the 21st Koli Calling International Conference on Computing Education Research*. https://doi.org/10.1145/3488042.3488050.

United Nations Educational, Scientific and Cultural Organization (UNESCO). 2023. "Global Education Monitoring Report 2023: Technology in Education – a Tool on Whose Terms?" Paris, France: UNESCO. https://www.unesco.org/gem-report/en/publication/technology.

Watson, Christopher, and Frederick WB Li. 2014. "Failure Rates in Introductory Programming Revisited." In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 39–44.

Weinberger, Armin. 2011. "Principles of Transactive Computer-Supported Collaboration Scripts." *Nordic Journal of Digital Literacy* 6 (3): 189–202.

Whitton, Nicola. 2022. *Play and Learning in Adulthood: Reimagining Pedagogy and the Politics of Education*. Springer Nature.

Williams, Laurie A. 2010. "Pair Programming." *Encyclopedia of Software Engineering* 2.

Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. 2000. "Strengthening the Case for Pair Programming." *IEEE Software* 17 (4): 19–25.

Wing, Jeannette M. 2006. "Computational Thinking." *Communications of the ACM* 49 (3): 33–35.

Xiao, Pei, Liang Chen, Xiaoqin Dong, Zhiya Zhao, Jincong Yu, Dongming Wang, and Wenzhen Li. 2022. "Anxiety, Depression, and Satisfaction with Life Among College Students in China: Nine Months After Initiation of the Outbreak of COVID-19." *Frontiers in Psychiatry* 12 (January). https://doi.org/10.3389/fpsyt.2021.777190.

Zarb, Mark, and Janet Hughes. 2015. "Breaking the Communication Barrier: Guidelines to Aid Communication Within Pair Programming." *Computer Science Education* 25 (2): 120–51.

Zeidner, Moshe. 1991. "Statistics and Mathematics Anxiety in Social Science Students: Some Interesting Parallels." *British Journal of Educational Psychology* 61 (3): 319–28. https://doi.org/10.1111/j.2044-8279.1991.tb00989.x.