Bridging Languages: Teaching C to Python Novices

Sam Skipsey

School of Physics and Astronomy, University of Glasgow
Samuel.Skipsey@glasgow.ac.uk

Gordon Stewart

School of Physics and Astronomy, University of Glasgow
Gordon.Stewart@glasgow.ac.uk

Jeremy Singer

School of Computing Science, University of Glasgow
Jeremy.Singer@glasgow.ac.uk

David Cutting

School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast
d.cutting@qub.ac.uk

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[90] to C[91]), and secondly, the development environment shifts from a browser-based notebook to a terminal-based text editor (i.e. from Jupyter[92] 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.

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.

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 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[93].

As a consequence of arguments similar to those described in (Balreira, Silveira, and Wickboldt 2022) 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

[90] https://www.python.org
[91] https://9p.io/cm/cs/who/dmr/chist.html
[92] https://jupyter.org
[93] https://github.com/features/codespaces

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.

"P2T" – A Specialised Course in C for Physicists

In the second semester of year two, Physics offers the course "P2T[94]: C Programming Under Linux", which introduces lower-level programming in the C language[95], 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 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.

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

[94] 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".

[95] 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.

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.

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).

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.)[96]. 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.

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 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

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

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.

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.

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. 2014) 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 2022), 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
Rational"
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.

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.

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>".

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 the 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!

References

Balreira, Dennis G., Thiago L. T. da Silveira, and Juliano A. Wickboldt. 2022. “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.

Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Shah. Viral B. 2014. “Julia: A Fresh Approach to Numerical Computing.” *SIAM Review* 59: 65--98. https://doi.org/10.1137/141000671.

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.

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

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.

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*, 180–86. ACM. 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 of Conferences*, 337:01266. EDP Sciences. https://doi.org/10.1051/epjconf/202533701266.

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*, 1–10. ACM. https://doi.org/10.1145/3488042.3488050.