




Research Software Engineering Skills

Jack Atkinson¹, Amy Pike¹, and Marion Weinzierl¹

¹ Institute of Computing for Climate Science, University of Cambridge, UK  Corresponding author

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Submitted: 26 September 2024

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

Summary

Research across many domains increasingly relies on software for simulation, data processing, analysis, visualisation, and more. This software is often written by individuals without a formal training in software engineering leading to inefficiencies, bugs, and poor reproducibility/re-usability.

The course described in this paper is designed to raise awareness software engineering principles and how they can be applied to research code. It illustrates, through a series of practical exercises, why these are important and introduced basic techniques to produce FAIR-er code (Barker et al., 2022). We hope that introducing a software engineering mindset, with simple tools and techniques to facilitate this will have long-term impact with participants continuing to explore ideas in their future work beyond this course.

Statement of need

Software forms a significant part of research today, with much of it being written and developed by academic researchers. Many of these individuals are self-taught or have minimal training, which will often focus on ‘coding’ rather than software development. As a result there is often a lack of awareness of software development principles amongst academic developers leading to software that is hard to use and maintain, presenting a barrier to research.

In our experience working with academic developers many of the bad practices and anti-patterns are a result of them:

- learning on the job from old code and emulating habits they observe
- never having been made aware of the available tools and techniques
- not keeping up to date with language/package developments
- not investing time into writing good code, often due to placing priorities elsewhere.

The intent of developing this short course, *RSE Skills (in Python)*, is to educate individuals in some basic software engineering principles that they can carry forward in their work. It is designed to introduce key tools and techniques to facilitate the writing of better code, but also to raise awareness of research software engineering and provide pointers to allow researchers to continue exploring and learning beyond this material.

In its intent our course is related to the work of Wilson et al. (2013), Wilson et al. (2017), and the corresponding course of Wallace (2022). While some of the content overlaps, our course is not as comprehensive but is intended as a shorter “What can I do better in my code right now?” training, which can be taught in 1.5–3 hours and adapted as needed. Another difference is that our course has a focus on demonstrating practical tools that can be used to help achieve the ‘good enough’ principles of Wallace (2022).

Learning objectives

The key learning objective for the workshop is to *Introduce key tools and concepts of research software engineering, and how they can be applied in everyday use to write higher-quality code.*

More specifically we cover:

- Why software engineering principles matter
- (Virtual) environments and dependencies
- Code standards and best practice (through PEP)
 - Code formatters and linting/static-analysis tools
- Documentation
 - Docstrings
 - Advice on commenting
 - The idea of self-documenting code and variable naming
 - READMEs and other useful files in your repository
 - Software licenses
- General principles for better code
 - Magic Numbers
 - Removal of hard-coded content to configuration files
 - use of f-strings in Python.

Teaching materials

All of the teaching materials for this course are [available online in a GitHub repository](#).

Slides

All of the key material is contained in a slide deck for the course [available online](#) and linked from the repository README. There is also a [recording of the workshop](#) available. The slides are written in [Quarto](#) markdown ([Allaire et al., 2022](#)) and rendered as [reveal.js](#) html. Source and instructions on how to render are included in the repository should others wish to tailor them to their specifications.

The slides are broken into separate files by section covering:

- Introduction to research software engineering, why it matters, and some examples,
- The use of virtual environments and specifying project dependencies,
- Code formatting for consistent readable layout,
- Naming principles for readable, re-useable code,
- Linting and static analysis of code,
- Writing docstrings and comments,
- READMEs and Licenses, and
- Other general principles including magic numbers, avoiding hard-coded values, etc.

This makes it easy to remove sections to tailor the course, or for additional topics to be added in future.

Exercises

The slides are interspersed with a series of exercises, each one applying techniques that have just been taught.

We set up a scenario where users have been provided a working, but poorly written, piece of code that they are now expected to re-use in their work. As they progress through the

exercises they apply their new skills to transform this into a much more readable and understandable piece of code.

The script we use is based on some of the code and data available in Irving (2019). This was chosen as it features typical applications we expect users to encounter (reading from a dataset, processing, and plotting). The course was originally designed to be taught to geoscientists, and then at a climate science summer school, and we wanted participants to be focussed on the changes to the code rather than worrying about the domain application or semantics.

Exercises roughly match the slide sections above:

- 01: Baseline code to examine, install dependencies, and run.
Users set up an environment and familiarise themselves with what the code does. They see why listing dependencies with a setup is useful, and make early observations about how the code is difficult to understand/use.
- 02: Apply a formatter to standardise code.
We introduce formatting through the black python formatter (Langa & contributors to Black, n.d.) to standardise code appearance and improve readability.
- 03: Improve code clarity with naming and source changes.
Users work through the code to find instances where they feel naming can improve clarity.
- 04: Linting and static analysis of code.
We introduce static analysis and linting through Pylint (Pylint contributors, n.d.) to catch errors and points for improvement.
- 05: Writing docstrings and best use of comments.
Users work through the code adding or improving docstrings.
- 06: General techniques for better code (magic numbers, string formatting).
Users work through the code to remove magic numbers and hard-coded values, and address other issues.
- 00: The end point of the workshop
This is provided as an example of an improved version of the code in exercise 01 that users can compare their work to.

Each exercises builds upon the results of the previous one, allowing participants to apply what they are learning in a continuous fashion to a code. However, each exercise has a dedicated subdirectory meaning that success in a previous exercise is not a pre-requisite for continuing to the next one. Two additional benefits to this approach are that the starting point for a subsequent exercise can be viewed as the solution to the current one, allowing participants to compare their work to an 'ideal' solution, and that exercises can easily be removed to tailor the course to different emphasis or time constraints.

Content Delivery

The course has been designed to be flexible in terms of delivery, allowing it to be adapted to and reused in various environments. We have adopted a modular design for the course material – each subsection of the slides is contained in a separate file, and each associated exercise in a separate directory. This allows for easy inclusion or exclusion of particular sections in the workshop meaning the content can be adapted to the length and focus of the sessions and the skills of the audience.

The slides are interspersed with the exercises, rather than being separate from the presentation material, as they are an integral component of the course. Indeed, the main aspect we wish to emphasise in delivery is teaching in a “code-along” fashion. This helps with engagement, participation, and understanding (Barba et al., 2022) and is essential, we feel, to having a long-lasting benefit (Rubin, 2013). This approach slows those leading the course towards the rate at which the participants are working, and illustrates

through errors (whether intentional or not!) that even experienced coders are human and make mistakes. Such errors can illustrate common pitfalls and provide an opportunity to include the teaching of debugging approaches. More generally this approach helps emphasise RSE principles, as participants can see the live application of these ideas in practise.

We believe that there is sufficient guidance in the slides and repository documentation to follow the course alone, and we include a link to a [recording of the workshop](#). This is, however, no substitute to in-person delivery where participants can ask questions, and successive workshops are continually improving.

Teaching experience

One aspect of the material we have found useful is the ability to tailor the course to the workshop/audience. By breaking the slides into separate files by section and having standalone exercises for each section it is easy to add or remove material as desired.

As discussed in the introduction, the course is written in Python for the broadest appeal, but is intended to educate researchers more generally on software engineering principles. We found it useful during delivery to reference tools and techniques applicable for other languages and systems as we went along. For example, where we introduce environment and dependency management in Python using virtual environments, we also mention Spack as an analogous approach for those using HPC environments. We also discuss other language specific tools for formatting and linting/static analysis such as ClangFormat and Clang-Tidy for C++. We feel that this is important to help participants see past the python specifics and understand the broader ideas and how they can be deployed in their own projects.

We found that an effective way to increase long-term adoption of some of the tools and principles introduced in the talk was to show users how they can integrate them into their text editors or IDEs. Making these tools a seamless part of the workflow reduces the ‘activation energy’ required to use them so it is worth taking time to show how this is done/used. We also found that it was very useful to highlight the use of an IDE, or other tool such as vimdiff, to compare two files side-by-side. This was an invaluable teaching tool for showing the effects of changes made by formatters, or allowing participants to compare their work to the example solution for each exercise.

A more in-depth modification to the course would be to change the domain focus of the example code used in the exercises. Whilst the course could be taught to any researchers as-is, we felt that matching the code to the domain of the participants allowed them to focus on the content we were teaching rather than focussing on the code itself. To make this change would require a different example code and then working backwards through the exercises to introduce ‘bad’ code to be dealt with.

Finally, we encourage participants to feed experiences back into the project, either via a GitHub issue or pull request. This allows us to continually learn from delivery and improve the material for future participants, especially if making instructions clearer or providing solutions to previously unencountered problems.

Acknowledgments

We thank anyone who has made a contribution to these materials, however small, assisted in code review for us, or helped as demonstrators on the courses.

The [Institute of Computing for Climate Science](#) received support through [Schmidt Sciences](#).

References

- Allaire, J. J., Teague, C., Scheidegger, C., Xie, Y., & Dervieux, C. (2022). *Quarto* (Version 1.2). <https://doi.org/10.5281/zenodo.5960048>
- Barba, L. A., Barker, L. J., Blank, D. S., Brown, J., Downey, A., George, T., Heagy, L. J., Mandli, K., Moore, J. K., Lippert, D., Niemeyer, K., Watkins, R., West, R., Wickes, E., Willing, C., & Zingale, M. (2022). *Teaching and Learning with Jupyter*. <https://doi.org/10.6084/m9.figshare.19608801.v1>
- Barker, M., Chue Hong, N. P., Katz, D. S., Lamprecht, A.-L., Martinez-Ortiz, C., Psomopoulos, F., Harrow, J., Castro, L. J., Gruenpeter, M., Martinez, P. A., & others. (2022). Introducing the FAIR principles for research software. *Scientific Data*, 9(1), 622. <https://doi.org/10.1038/s41597-022-01710-x>
- Irving, D. (2019). Python for atmosphere and ocean scientists. *Journal of Open Source Education*, 2(16), 37. <https://doi.org/10.21105/jose.00037>
- Langa, E., & contributors to Black. (n.d.). *Black: The uncompromising Python code formatter*. <https://github.com/psf/black>
- Pylint contributors. (n.d.). *Pylint*. <https://github.com/pylint-dev/pylint>
- Rubin, M. J. (2013). The effectiveness of live-coding to teach introductory programming. *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, 651–656. <https://doi.org/10.1145/2445196.2445388>
- Wallace, M., E.W. J. (2022). *Good enough practices in scientific computing: A lesson (version 0.1.0)*. <https://doi.org/10.5281/zenodo.10783026>
- Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K., Mitchell, I. M., Plumbley, M., Waugh, B., White, E. P., & Wilson, P. (2013). Best practices for scientific computing. *PLoS Biology*, 12(1), e1001745+. <https://doi.org/10.1371/journal.pbio.1001745>
- Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6), e1005510+. <https://doi.org/10.1371/journal.pcbi.1005510>