ESSAY

A how-to guide for code sharing in biology

Richard J. Abdill[1]*, **Emma Talarico**[2], **Laura Grieneisen**[2,3]

1 Section of Genetic Medicine, Department of Medicine, University of Chicago, Chicago, Illinois, United States of America, **2** Department of Biology, University of British Columbia—Okanagan Campus, Kelowna, British Columbia, Canada, **3** Okanagan Institute for Biodiversity, Resilience, and Ecosystem Services, University of British Columbia—Okanagan Campus, Kelowna, British Columbia, Canada

* rabdill@uchicago.edu

Abstract

In 2024, all biology is computational biology. Computer-aided analysis continues to spread into new fields, becoming more accessible to researchers trained in the wet lab who are eager to take advantage of growing datasets, falling costs, and novel assays that present new opportunities for discovery. It is currently much easier to find guidance for implementing these techniques than for reporting their use, leaving biologists to guess which details and files are relevant. In this essay, we review existing literature on the topic, summarize common tips, and link to additional resources for training. Following this overview, we then provide a set of recommendations for sharing code, with an eye toward guiding those who are comparatively new to applying open science principles to their computational work. Taken together, we provide a guide for biologists who seek to follow code sharing best practices but are unsure where to start.

OPEN ACCESS

Citation: Abdill RJ, Talarico E, Grieneisen L (2024) A how-to guide for code sharing in biology. PLoS Biol 22(9): e3002815. https://doi.org/10.1371/journal.pbio.3002815

Published: September 10, 2024

Funding: This study was supported by a National Science and Engineering Research Council of Canada Discovery Grant RGPIN-2023-06664 and a University of British Columbia ASPIRE award (LG). The funders had no role in study design, data collection and analysis, decision to publish, or preparation of the manuscript.

Competing interests: The authors have declared that no competing interests exist.

Introduction

Reproducible computational practices, and open science more broadly, are the subject of many discussions about competing priorities: Transparency is good, but its implementation is time intensive and poorly incentivized [1]. Open research is associated with more citations and more media coverage [2], but it can expose researchers to new avenues for harassment and suppression [3]. There are also many resources promoting particular approaches to performing computational work [4,5] and developing research software [6]. But amidst the discussions of how to perform computational research, where to publish it, and how to organize your files, there is a dearth of information on how to be transparent about work that's already been done, particularly for biologists who may not specialize in computational work. Many researchers are unsure how to share code or, as a recent Springer Nature survey found, even where to upload their data [7].

A complicating factor is the wide variation between fields in the standards for data and code sharing, as well as the types of data sets and code used. For example, data sharing standards are well reviewed in the ecological literature [8,9], and major journals in the field of ecology have extensive data sharing policies, such as Ecological Society of America's Open Research Policy "Definitions" page [10]. However, these practices are less standardized in other fields of biology. This means that research and journals at the intersection of multiple fields—such as microbiome science, which integrates medicine, ecology, and computational biology—may not have scientists trained in uniform standards of data and code sharing. Finally, many established papers on best practices focus on how to design a study from the outset to fit into a reproducible science framework. However, science is messy, and as projects and data sets are passed between lab members, the researchers who are assembling the final paper for submission often inherit files and code that were not created using best practices.

The goal of our paper, therefore, is to provide an integrative guide for sharing code, such that these practices can be implemented across biological subfields and stages of the research process. We focus on the implementation details particular to reproducible results—those that can be regenerated using the original data—as opposed to replicable results, in which another group is able to draw similar conclusions from new data [11,12].

We can all share code, if not because we want someone else to be able to use it in the future, then at least because we have already used it for the research being reported. The code used to analyze data, perform statistical tests, and build visualizations is no less critical to a project than reagents used at the bench and should be disclosed for the same reasons: In addition to providing critical information about the conditions under which the study was performed, sharing this information enables others to more easily validate computationally derived conclusions. It also reduces the effort required to apply similar methods in new projects by providing an example that allows others to avoid issues you may have already solved. A 2023 study across dozens of participants from 13 countries found broad support for publications that clearly state whether code was shared openly, with a persistent identifier (such as a digital object identifier (DOI)) and a clear license [13]. Sharing code also helps protect us against what Donoho and colleagues [14] called "the ubiquity of error" at the heart of the scientific method, which drives scientists to expend effort primarily "in recognizing and rooting out error": Even the most diligent scientists can miss a typo in a command or misunderstand parameters of a complex function in a package developed elsewhere. Sharing this code—essentially a fine-grained addendum to a methods section—can help bring these issues to the surface and help future researchers avoid them.

In this discussion, we use the term "code" as shorthand for any of the documents interpreted by a computer to generate information used in your manuscript. That covers software you've developed to perform analyses (a new algorithm implementation, for example), but it also includes commands used to perform statistical tests and the scripts used to generate figure panels. Most manuscripts don't come bundled with an entirely new software application, but many—especially those that include the analysis of genomic sequencing data—required code to get their results.

Below, we outline how to build a more easily reproducible project, share a set of resources for improving your coding skills, describe which pieces of your project are most important for others, and lay out how to get your work online in a practical format. Our recommendations were developed as guidance mainly to those who are preparing to share completed work. However, much of this will be easier—and more effective—if a project is started with reproducibility in mind. If you've already skipped some of these steps, or inherited a project that was set up in a different way, there is still plenty you can share to provide transparency and demonstrate your process to others.

Setting up a reproducible project

First things first: Your code is good enough to share [15]! It may be messy and disorganized and cobbled together by self-taught coders who are writing just enough code to get the job done, but you aren't the only person for whom that's true. If you trust the code enough that you've written a paper about its results, it's certainly worthy of sharing. Scientific code can be split into 2 broad categories: products that are intended to be reused by others (such as a new software package), and products that aren't. The preparation, packaging, and archiving of code differs greatly between these 2 categories, and this paper deals mostly with the latter—code shared to demonstrate the computational approach to a single paper, but that others should not expect to work like a broadly applicable tool with a friendly user interface. The recommendations here describe considerations that are helpful, but if some are not practical to implement, it's important to note that whatever you can share is almost always better than sharing nothing.

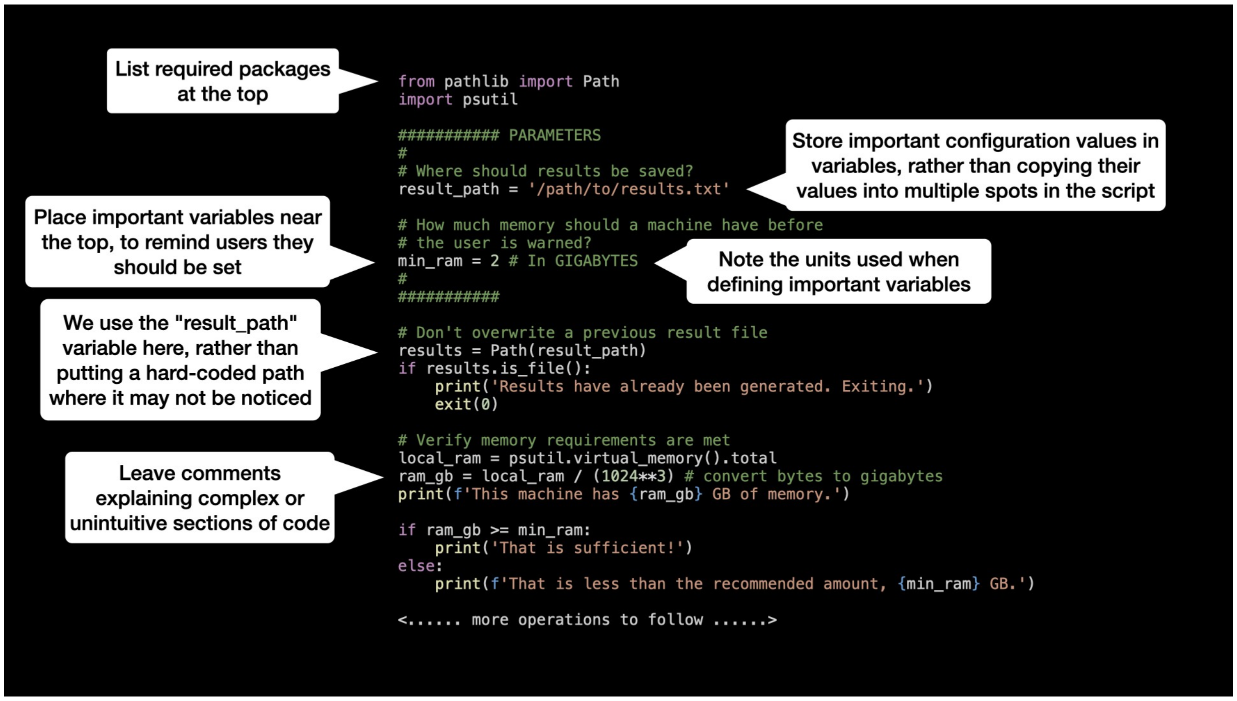
There are many guides for reproducible computational biology available online, both peer reviewed and independently published (for example, [9]), covering diverse topics such as telling stories with computational notebooks [16,17], organizational recommendations for large computational projects [18], and standards and checklists that provide very specific examples [19,20]. Rather than duplicate this effort, here we review common recommendations, going from general to specific. We highlight these as examples in the broad categories of accessibility, organization, and minimizing repeated work, with a focus on changes that are particularly impactful but that do not require significant new technical skills. Because this work is focused mostly on sharing projects that are already complete, we urge readers to investigate implementation details for their new projects in the papers and software documentation referenced below.

The first key to sharing your code is to **use code in the first place**. While many tools simplify their operation by enabling users to interact with them using graphics—such as icons, text boxes, and menus—this point-and-click approach can be difficult to document and even more difficult to replicate. Though it can be tempting to skip automation and programmatic approaches in favor of ad hoc point-and-click solutions, these shortcuts can backfire later when trying to repeat an analysis, recall who did what, which subset of the data was used for a figure panel, or exactly which version of a program was used for an important computational step. Understanding command-line tools and their parameterization can be a challenge, but running these operations a second time is much more straightforward than carefully following step-by-step instructions on which buttons to click in which order [21].

Keep humans in mind. Code style and organization is a recurring theme in the literature. In short, consider approaching your code as if it were intended to be read, rather than executed [22]. Give variables helpful names, rather than "foo" or "x," and leave plenty of comments to explain what different sections of code are doing and reasons for unconventional design decisions that users may be tempted to modify (for examples, see Fig 1) [21].

Use a consistent directory structure. Make it easy to distinguish raw data from intermediate files and final results [19,23,24]. For example, when manipulating your data specifically for a visualization, save the version of the data that's actually displayed [25]: This will be a more convenient file for readers to evaluate and makes it possible to rebuild your figure by simply reloading that file, rather than having to load the original dataset and performing all the processing steps again. Minimize the manual intervention required to run your scripts, and ideally organize them in a way that allows users to run them from start to finish [20]. Provide a "README" file that, at a minimum, walks a user through the intended execution of your code and documents the key steps [26].

Don't be afraid of the tedious work of automating your data cleaning. In particular, avoid manual editing of intermediate files [9,24,25,27]. For example, if script A processes raw data into a table of gene expression levels, and script B summarizes this table by pathway, we should be concerned if there is a step after script A that requires a user to open the table and edit fields by hand to prepare the data for script B. Such quick fixes can be tempting, but they also add risk: A researcher could easily add a typo, corrupt a file in unexpected ways [28], or forget the step altogether.


List required packages at the top
Store important configuration values in variables, rather than copying their values into multiple spots in the script
Place important variables near the top, to remind users they should be set
Note the units used when defining important variables
We use the "result_path" variable here, rather than putting a hard-coded path where it may not be noticed
Leave comments explaining complex or unintuitive sections of code

```python
from pathlib import Path
import psutil

########### PARAMETERS
#
# Where should results be saved?
result_path = '/path/to/results.txt'

# How much memory should a machine have before
# the user is warned?
min_ram = 2 # In GIGABYTES
#
###########

# Don't overwrite a previous result file
results = Path(result_path)
if results.is_file():
    print('Results have already been generated. Exiting.')
    exit(0)

# Verify memory requirements are met
local_ram = psutil.virtual_memory().total
ram_gb = local_ram / (1024**3) # convert bytes to gigabytes
print(f'This machine has {ram_gb} GB of memory.')

if ram_gb >= min_ram:
    print('That is sufficient!')
else:
    print(f'That is less than the recommended amount, {min_ram} GB.')

<...... more operations to follow ......>
```

Fig 1. Example code for a reproducible project. The first lines of a longer analysis script written in Python 3 with examples of practices that make the code easier for other users to understand.

https://doi.org/10.1371/journal.pbio.3002815.g001

Minimize cut-and-paste errors. Use custom functions to do repeated operations [9], and store important values in prominently commented variables, rather than hard-coding (Glossary) them somewhere deep in the script where a future user may not notice it (Fig 1). For example, if you write out a multistep process that prepares data for a particular visualization but find out later that you need to perform the same steps for a different subset of data in a new figure, avoid copying that code and pasting it farther down in your analysis script. If you later find an error in this code or simply modify it to change a threshold or reorganize the output, it's easy to forget to scroll back down and change it in two (or more!) places. Repetitive code is one example of "code smell": code that may work as intended but is suggestive of a larger design flaw that may cause hard-to-find bugs or make it more complicated to modify the program in the future [29]. Organizations and open-source (Glossary) communities have published "style guides" in many popular languages including R (https://style.tidyverse.org) and Python (https://peps.python.org/pep-0008/) that may help avoid some of these issues, but eventually you will develop a "nose" for intuiting when you may be wandering down an ill-advised path.

Glossary

Application Programming Interface (API) keys

Authentication parameters generally used in a similar manner as passwords when 2 software applications communicate with each other, typically over the internet.

Command-line utilities

Computer applications that are guided using text entered into a terminal. Implicitly refers to programs running on the Linux operating system.

Dependencies

In the context of software packages or scripts, dependencies are libraries and packages of third-party code that must be present for a given package or script to function.

Hard-coding

When the value of a variable or process is manually specified in a way that is not easily modified by someone who wants to run the code. A "hard-coded" file path would only look in one specific directory for a file, rather than exposing a way for the path to be modified via configuration files or command-line options.

Linux

Generally, this refers to the family of operating systems that used the Linux kernel. Ubuntu is one such operating system (or "distribution"), as is Rocky Linux, Debian, and Android. High-performance computing clusters rely heavily on nodes running Linux operating systems in the same way many desktop computers run Windows.

Open-source

"Source" is a reference to a given software's "source code," or the text documents written in languages such as Python or Java that are then prepared and interpreted by the computer for execution. In a general sense, code that is "open" is freely available for inspection, but open-source advocates frequently incorporate additional licensing considerations when deciding whether software is "open," such as free redistribution of the software [30].

Software container

A self-contained computing environment that can be launched with a predefined set of files and software. Similar to virtual machines, containers can be useful in situations where an analysis requires software that is complicated to install or requires very specific system specifications [31].

These are relatively uncontroversial recommendations to make your project easier to manage and adhere more closely to programming best practices, but there are many other opportunities to improve. A 2014 survey showed that software developers reviewing code from computational biology papers were shocked at its content, confusing structure, and lack of documentation [32]. Still, optimizing processes past a point of practical reproducibility may not be worthwhile [33], particularly when competing priorities leave researchers with few professional incentives to tackle the time-consuming work of sharing digital materials.

There are many other recommendations that appear in the literature even without expanding your search beyond papers focused on computational biology: Version control and code

Table 1. Programming resources.

Program	Description	URL
The Carpentries workshops	Software Carpentry workshops are held all over the world and online. Data Carpentry workshops are less frequent but may be more applicable.	https://carpentries.org
The Carpentries online resources	Carpentries volunteers, many of them full-time researchers, have also built interactive lessons that can be taken at your own pace.	https://carpentries-lab.github.io/good-enough-practices/index.html https://datacarpentry.org/semester-biology/
Glittr	This website organizes hundreds of free, open-source training courses in bioinformatics, from foundational "Python for data analysis" materials to more specific courses in machine learning packages. See the "Reproducibility" topic category in particular.	https://glittr.org
Stack Overflow	A question-and-answer website about computer code. If you have a specific question about anything code related, someone has probably already asked and answered it on Stack Overflow.	https://stackoverflow.com
Biostar	A bioinformatics-focused forum with more than a decade of archived discussions [41].	https://biostars.com
Nextflow and Snakemake	Examples of popular workflow management tools with curated collections of pipelines that may provide the functionality you need, or at least provide a sophisticated example of officially endorsed implementations that you can modify.	Nextflow: https://nf-co.re [42] Snakemake: https://snakemake.github.io/snakemake-workflow-catalog/
GitHub Skills	The "Introduction to GitHub" course is a useful introduction to version control on the most popular platform for open-source code. Other courses cover more advanced concepts, such as pull requests and automation.	https://skills.github.com

https://doi.org/10.1371/journal.pbio.3002815.t001

review are frequent topics of discussion [9,21,24,34–39], as is "defensive programming" [22], that is, performing what may feel like excessive validation of the inputs and outputs of functions to make sure unexpected states (such as a negative quantity of items) are detected before they can cause problems. For particularly complicated operations, automated testing of sections of code with known inputs and expected outputs is another practice that can make your code more robust, to catch scenarios where a small change has unintended effects elsewhere [40]. These are all complex processes with an intimidating learning curve, but there have never been more resources available for those looking to implement them. Below is a list of valuable websites offering lessons in software development practices applicable to computational biology (Table 1).

Files to share

When your project is complete and you're preparing to share your computational work, even the best-organized projects can be a tangled web of intermediate files and quick fixes. In short, you should try to share anything you created that would be necessary to reproduce your calculations. More specifically:

1. **Scripts for data-cleaning and analysis.** Nothing is too mundane! These steps make it easier for others to reproduce your work and can clarify exactly what was done and, crucially, in what order. For example, it may seem trivial to share the exact Python statement you used to perform a straightforward logistic regression, but the "LogisticRegression" function from the popular Python package scikit-learn defaults to a technique that penalizes coefficients in large models (known as L2 regularization) [43,44], while R's built-in "glm" function doesn't include similar penalties even as an option [45]. These penalties can dramatically alter the results of a regression, which is why it's critical for users to understand which options are being used by their statistical libraries. This is just one of many potentially important details about model development [46] that may not be obvious from looking at the outcome but can be tracked down using the original code.

2. **Data visualization code.** Sharing the code you used to generate your figure panels can help people who want to visualize their data in a similar way and clarify finer points of the figure

that have been omitted from the legend, intentionally or not. The code can also show exactly how data were filtered and modified before visualization. This may enable readers to explore your results using "living figures" they can modify to look at different subsets of your data, or, in some cases, even add data of their own [47]. It may also serve as valuable documentation for your own future reference.

3. **Parameters used to configure and launch command-line utilities.** Many computational biology tools are executed from the Linux command line (Glossary), with relevant parameters included directly in the command. These parameters may specify the location of input and output files, for example, or set other configuration values such as thresholds or file formats. Ideally, a single script could be executed to run each command and perform your entire analysis process [20]. But even if that isn't how you executed these operations, including a list of the commands used may be useful for those trying to evaluate minor implementation details, either to reproduce a paper's findings or to apply a similar process to their own data. (It's also worth noting that running commands using scripts is highly preferable to attempting to reconstruct these commands after the fact—see "Setting up a reproducible project," above, for other techniques to keep in mind.)

4. **Pipeline specifications and configuration files.** The files defining a series of data-processing steps (sometimes called "pipeline code") are valuable resources, regardless of how much sophisticated automation they use. Workflow automation tools such as Snakemake [48] or Nextflow [49] can streamline your own bioinformatics work and make it easier to reproduce, but the Bash and Perl scripts used by many still provide valuable documentation of the process, even in situations where the files are written to work only for your data or to run only on a specific machine. Automating the installation and configuration of your tools with workflow managers, package managers (for example, Conda, used for the coordination of installing dependencies), and software container platforms (for example, Apptainer/Singularity [50], Docker) can make things easier for you to manage, but they also make it easier for interested parties to learn about your environment even if they can't execute the exact code. If you did not use a workflow automation tool, it would still be useful to include a brief summary of all the steps performed to generate your results and figures.

5. **A list of dependencies.** Providing a very specific list of all software dependencies (Glossary) in your pipeline may make a critical difference in how reliably your work can be reproduced. For example, when version 1.16.0 of the popular Python package NumPy—which can be imported into Python scripts to perform many linear algebra operations—was released, the functionality of its matrix multiplication function was unintentionally dramatically changed. This wasn't fixed until the release of version 1.16.6 nearly a year later [51,52]. There have been dozens of releases since, but a script that worked one way in 2019 may work very differently now—unless you record the version numbers. Placing a call to "sessionInfo()" within R scripts should print all package versions in the output of the script. In Python, running "pip freeze" (or its equivalent, if you're using a different tool such as Conda for installing packages) will print out the versions of the packages installed in your environment. A software container (Glossary), such as those on the Docker and Singularity platforms, with everything already installed would provide a more complete account of dependencies, but even a list of library versions will cover most contingencies that don't involve low-level factors such as drivers and differences in hardware [53].

6. **A list of almost-dependencies.** Though a software "dependency" is generally a reference to a package or library that must be locally available for the script to run, there are likely other things your code also depends on. Components of your pipeline may not be "dependencies"

per se, but they can be critical to reproducing your work. Reference databases, for example, don't need to be shared with your code (assuming they are publicly available), but noting the version in your methods section is important because different reference databases can result in disparate results [54], even between minor versions of the same database [55]. Similarly, it is also helpful to note the operating system on which the code was run, particularly if you're using command-line utilities. Tools can behave differently when moved between platforms, and commands that work on a Linux machine may fail on Windows (or on slightly different distributions of Linux). Even worse, they may finish "successfully" and return different results. One example of this is a popular text-manipulation tool called sed, useful for performing repetitive text substitutions in large files. Multiple versions of sed have been developed for various operating systems, with different approaches for users to define which strings should be altered and in what ways [56]. A script may work as intended on a Linux computer with a specific version of sed [57], but running the same script on a macOS computer may not find the same strings for replacement, or even recognize the same command-line options [58].

7. **New software applications.** If you wrote a whole new program to perform your work, it's essential to be as transparent as possible about how it works. The editorial staff at the journal should provide guidance about how to handle software that you don't intend to make open source. Best practices for tool development have been well covered elsewhere [59–64] and are outside the scope of this review, but if you've developed a useful tool that is not the primary focus of your work, it may be helpful to submit a separate software paper or "application note" about your program to a computational journal [65], such as the Journal of Open Source Software (https://joss.theoj.org).

8. **A copyright notice and license.** If you are publishing a new computational tool or software package, choosing a license that describes what rights you've reserved may play a critical role in its wider adoption because it enables potential users to know which ways of using and sharing your code are legally acceptable. If you have specific needs around allowing (or controlling) reuse, then it would be beneficial to speak with an expert from your institution's library or office of legal counsel. Even if you are simply sharing the scripts you wrote to clean your data or process images, you can avoid potential headaches, such as emails from individual users asking for permission to modify your code for their work, if you include a "LICENSE.txt" file specifying the copyright holder (possibly your university) and the terms under which others can use, modify and share the code. This is one area in which even prominent companies with legal teams opt for commonly used [66], well-documented licenses such as MIT and Apache 2.0. Journals may have specific recommendations as well, and guides are available from organizations such as the Open Source Initiative (https://opensource.org/licenses) to help you decide.

Preparing the code

Source code is necessary, derivative files aren't. The most important files to include are the source code files themselves—the files with extensions such as ".py" and ".R" that were written by those performing the analyses. Byproducts of these scripts or their dependencies are not necessary to share—files such as those ending in ".pyc" and those created by the installation of packages, for example, will be regenerated by the user doing their own installations. A simple exercise for distinguishing these files is to try rerunning your code on a machine that was uninvolved in the original analysis: Some files will be regenerated on the new machine, others will need to be downloaded instead, and the ones that you need to move manually are likely the ones most important to share.

Test your reproducibility, or ask a friend. Switching workstations can also help highlight aspects of your code that will get in the way of others trying to run it themselves. Removing things like passwords and Application Programming Interface (API; Glossary) keys is critical, but other workstation-specific settings can also cause confusion: Hard-coded file paths will likely only work on a machine connected to that exact file system, even when pointing at common utilities. Storing strings like these in variables declared at the top of your script will provide clues to future users that this is a value they need to specify themselves, or an important variable that is used in multiple places. Showing your work to a colleague may also highlight other sources of confusion in the materials you're sharing: A person less familiar with the data may not know how to distinguish case samples from controls, for example, or your column labels may not be as intuitive as they seem. There are numerous resources available discussing practical considerations in data curation and sharing [67–72].

Rerun inherited code when practical. Another potential reproducibility headache is hiding in code written by people who are no longer contributing to the project—they may not have documented quick fixes or special cases, or they may have used older versions of libraries that have since changed their functionality. If possible, it can be helpful to make sure these scripts still run as expected before sharing. This can also work as an informal form of code review: While you're working through this code, you may get a better understanding of the data or, in some cases, catch long-forgotten errors.

Build bridges over black boxes. You may also encounter situations where there are computational steps that do not have any executable scripts associated with them: Perhaps a critical tool requires a commercial license, or it lacks a programmatic interface, making the "point-and-click" approach using menus and buttons as the only option. In situations like this, document whatever you can. Proprietary sections create a "black box" for future users of your code, in that they can see your input and output of the proprietary section but may lack the resources to run it themselves. It can be helpful to write out the computational steps both before and after the proprietary step and provide a bridge between sections of usable code by indicating where the difficult step is located. Providing users with the output of these tricky steps will enable them to skip over that step and pick up where you left off.

Provide documentation. A brief "README" text file should be the minimum descriptor of your code. Providing even a cursory description of how to follow the pipeline could make a big difference in how useful your files are to others, doubly so if you provide more detail about things like important parameters, installing complex dependencies, and where to place input files [26]. Consider someone who has read your paper and is trying to learn more from looking at your code—where should they start? If they want to reproduce an analysis, which script should they run, and which files will the script be looking for? Does anything need to be installed first? This is the file where you can explain in plain language how someone can use your code to answer their questions.

Deposit your files somewhere public and permanent. Repository services such as GitHub, GitLab, and Sourceforge are reliable tools for collaboration and distribution [73,74], particularly for ongoing software projects. However, many organizations such as university libraries [75] and groups like the Software Sustainability Institute [76] are skeptical of relying exclusively on commercial enterprises that haven't explicitly stated no-cost perpetual hosting of digital artifacts as one of their goals. In contrast, repositories such as Zenodo are free, publicly funded projects with decades-long retention plans [77]. Depositing files in a repository that mints DOIs makes it simpler for users to cite the work (and its version) as specifically as possible and makes links less likely to break over time by providing a canonical URL through the DOI resolution service available at doi.org. Deposits with these services are intended to be immutable artifacts, however, so it is preferable to wait until you have a "final" version before sharing there. If you are reluctant to share a permanent version of your data and code prior to publication, many journals (and repositories, such as Zenodo [78] and the Sequence Read Archive [79]) have mechanisms for sharing these resources with reviewers privately first. In addition, Zenodo allows new "versions" of archives to be uploaded, which provides a streamlined way to make corrections or additions if necessary, both before and after publication. There is also a process for linking a GitHub repository to Zenodo [80], if you're already using GitHub and would like "snapshots" saved in a more citable format.

Conclusions

Here, we have reviewed almost 3 dozen articles about reproducible research practices to summarize their recommendations. We hope this summary will be useful for those seeking to share their code to comply with requirements from journals or funders, or simply because they are enthusiastic about participating in the open science ecosystem [24]. The factors involved in the sharing of data and code—and in open science issues more broadly—are complex, both for researchers and research subjects, particularly in relation to the equitable participation of marginalized communities [81–84]. If you've decided to share your code, these recommendations will provide a starting point for the effort and connections to more detailed sources. Finally, we note that code sharing best practices should be taught early in the science curriculum alongside other Open Science approaches [85–89].

Acknowledgments

We thank Casey S. Greene, Katja Della Libera, and Elizabeth Gibbons for helpful feedback.

Author Contributions

Conceptualization: Richard J. Abdill, Laura Grieneisen.

Investigation: Richard J. Abdill, Emma Talarico.

Supervision: Laura Grieneisen.

Writing – original draft: Richard J. Abdill, Laura Grieneisen.

Writing – review & editing: Laura Grieneisen.

References

1. Allen C, Mehler DMA. Open science challenges, benefits and tips in early career and beyond. PLoS Biol. 2019; 17:e3000246. https://doi.org/10.1371/journal.pbio.3000246 PMID: 31042704
2. McKiernan EC, Bourne PE, Brown CT, Buck S, Kenall A, Lin J, et al. How open science helps researchers succeed. Elife. 2016; 5:e16800. https://doi.org/10.7554/eLife.16800 PMID: 27387362
3. Lewandowsky S, Bishop D. Research integrity: Don't let transparency damage science. Nature 2016; 529:459–461. https://doi.org/10.1038/529459a PMID: 26819029
4. Toelch U, Ostwald D. Digital open science-Teaching digital tools for reproducible and transparent research. PLoS Biol. 2018; 16:e2006022. https://doi.org/10.1371/journal.pbio.2006022 PMID: 30048447
5. Stodden V, Ferrini V, Gabanyi M, Lehnert K, Morton J, Berman H. Open access to research artifacts: Implementing the next generation data management plan. Proc Assoc Inf Sci Technol. 2019; 56:481–485. https://doi.org/10.1002/pra2.51
6. Koch T, Gläser D, Seeland A, Roy S, Schulze K, Weishaupt K, et al. A sustainable infrastructure concept for improved accessibility, reusability, and archival of research software. arXiv [csSE]. 2023. https://doi.org/10.48550/ARXIV.2301.12830

7. Stuart D, Baynes G, Hrynaszkiewicz I, Allin K, Penny D, Lucraft M, et al. Whitepaper: Practical challenges for researchers in data sharing. figshare; 2018. https://doi.org/10.6084/M9.FIGSHARE.5975011.V1

8. Powers SM, Hampton SE. Open science, reproducibility, and transparency in ecology. Ecol Appl. 2019; 29:e01822. https://doi.org/10.1002/eap.1822 PMID: 30362295

9. Cooper N, Hsing P-Y, editors. Reproducible Code. British Ecological Society; 2017. Available from: https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf

10. Open Research Policy. [cited 2024 Jan 4]. Ecological Society of America [Internet]. Available from: https://www.esa.org/publications/data-policy/.

11. Mesirov JP. Accessible Reproducible Research. Science 2010; 327:415–416. https://doi.org/10.1126/science.1179653 PMID: 20093459

12. Peng RD. Reproducible research in computational science. Science. 2011; 334:1226–1227. https://doi.org/10.1126/science.1213847 PMID: 22144613

13. Cobey KD, Haustein S, Brehaut J, Dirnagl U, Franzen DL, Hemkens LG, et al. Community consensus on core open science practices to monitor in biomedicine. PLoS Biol. 2023; 21:e3001949. https://doi.org/10.1371/journal.pbio.3001949 PMID: 36693044

14. Donoho DL, Maleki A, Rahman IU, Shahram M, Stodden V. Reproducible Research in Computational Harmonic Analysis. Comput Sci Eng. 2009; 11:8–18. https://doi.org/10.1109/MCSE.2009.15

15. Barnes N. Publish your computer code: it is good enough. Nature. 2010; 467:753. https://doi.org/10.1038/467753a PMID: 20944687

16. Rule A, Birmingham A, Zuniga C, Altintas I, Huang S-C, Knight R, et al. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. PLoS Comput Biol. 2019; 15:e1007007. https://doi.org/10.1371/journal.pcbi.1007007 PMID: 31344036

17. Figueiredo L, Scherer C, Cabral JS. A simple kit to use computational notebooks for more openness, reproducibility, and productivity in research. PLoS Comput Biol. 2022; 18:e1010356. https://doi.org/10.1371/journal.pcbi.1010356 PMID: 36107931

18. Lowndes JSS, Best BD, Scarborough C, Afflerbach JC, Frazier MR, O'Hara CC, et al. Our path to better science in less time using open data science tools. Nat Ecol Evol. 2017; 1:1–7. https://doi.org/10.1038/s41559-017-0160 PMID: 28812630

19. Sawchuk SL, Khair S. Computational reproducibility: A practical framework for data curators. J eSci Librariansh. 2021; 10. https://doi.org/10.7191/jeslib.2021.1206

20. Heil BJ, Hoffman MM, Markowetz F, Lee S-I, Greene CS, Hicks SC. Reproducibility standards for machine learning in the life sciences. Nat Methods. 2021; 18:1132–1135. https://doi.org/10.1038/s41592-021-01256-7 PMID: 34462593

21. Preeyanon L, Pyrkosz AB, Brown CT. Reproducible Bioinformatics Research for Biologists. In: Stodden V, Leisch F, Peng RD, editors. Implementing Reproducible Research. CRC Press; 2014.

22. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. Best practices for scientific computing. PLoS Biol. 2014; 12:e1001745. https://doi.org/10.1371/journal.pbio.1001745 PMID: 24415924

23. Kelly D, Hook D, Sanders R. Five Recommended Practices for Computational Scientists Who Write Software. Comput Sci Eng. 2009; 11:48–53. https://doi.org/10.1109/MCSE.2009.139

24. Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK. Good enough practices in scientific computing. PLoS Comput Biol. 2017; 13:e1005510. https://doi.org/10.1371/journal.pcbi.1005510 PMID: 28640806

25. Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten simple rules for reproducible computational research. PLoS Comput Biol. 2013; 9:e1003285. https://doi.org/10.1371/journal.pcbi.1003285 PMID: 24204232

26. Eglen SJ, Marwick B, Halchenko YO, Hanke M, Sufi S, Gleeson P, et al. Toward standard practices for sharing computer code and programs in neuroscience. Nat Neurosci. 2017; 20:770–773. https://doi.org/10.1038/nn.4550 PMID: 28542156

27. Noble WS. A quick guide to organizing computational biology projects. PLoS Comput Biol. 2009; 5: e1000424. https://doi.org/10.1371/journal.pcbi.1000424 PMID: 19649301

28. Ziemann M, Eren Y, El-Osta A. Gene name errors are widespread in the scientific literature. Genome Biol. 2016; 17:177. https://doi.org/10.1186/s13059-016-1044-7 PMID: 27552985

29. Alfadel M, Aljasser K, Alshayeb M. Empirical study of the relationship between design patterns and code smells. PLoS ONE. 2020; 15:e0231731. https://doi.org/10.1371/journal.pone.0231731 PMID: 32298360

30. Open Source Initiative. The Open Source Definition v1.9. 2007 Mar 22 [cited 2024 Jun 24]. Open Source Initiative [Internet]. Available from: https://opensource.org/osd.

31. Alser M, Lawlor B, Abdill RJ, Waymost S, Ayyala R, Rajkumar N, et al. Packaging and containerization of computational methods. Nat Protoc. 2024. https://doi.org/10.1038/s41596-024-00986-0 PMID: 38565959

32. Petre M, Wilson G. Code Review For and By Scientists. arXiv [csSE]. 2014. https://doi.org/10.48550/ARXIV.1407.5648

33. Raj A. From over-reproducibility to a reproducibility wish-list. 2016 Mar 3 [cited 2023 Aug 25]. RajLab [Internet]. Available from: https://rajlaboratory.blogspot.com/2016/03/from-over-reproducibility-to.html.

34. Ram K. Git can facilitate greater reproducibility and increased transparency in science. Source Code Biol Med. 2013; 8:7. https://doi.org/10.1186/1751-0473-8-7 PMID: 23448176

35. Blischak JD, Davenport ER, Wilson G. A Quick Introduction to Version Control with Git and GitHub. PLoS Comput Biol 2016; 12:e1004668. https://doi.org/10.1371/journal.pcbi.1004668 PMID: 26785377

36. Millman KJ, Pérez F. Developing open source scientific practice. In: Stodden V, Leisch F, Peng RD, editors. Implementing Reproducible Research. CRC Press; 2014.

37. Perez-Riverol Y, Gatto L, Wang R, Sachsenberg T, Uszkoreit J, Leprevost F da V, et al. Ten Simple Rules for Taking Advantage of Git and GitHub. PLoS Comput Biol. 2016; 12:e1004947. https://doi.org/10.1371/journal.pcbi.1004947 PMID: 27415786

38. Braga PHP, Hébert K, Hudgins EJ, Scott ER, Edwards BPM, Sánchez Reyes LL, et al. Not just for programmers: How GitHub can accelerate collaborative and reproducible research in ecology and evolution. Methods Ecol Evol. 2023; 14:1364–1380. https://doi.org/10.1111/2041-210x.14108

39. Ivimey-Cook ER, Pick JL, Bairos-Novak KR, Culina A, Gould E, Grainger M, et al. Implementing code review in the scientific workflow: Insights from ecology and evolutionary biology. J Evol Biol. 2023; 36:1347–1356. https://doi.org/10.1111/jeb.14230 PMID: 37812156

40. Balaban G, Grytten I, Rand KD, Scheffer L, Sandve GK. Ten simple rules for quick and dirty scientific programming. PLoS Comput Biol. 2021; 17:e1008549. https://doi.org/10.1371/journal.pcbi.1008549 PMID: 33705383

41. Parnell LD, Lindenbaum P, Shameer K, Dall'Olio GM, Swan DC, Jensen LJ, et al. BioStar: an online question & answer resource for the bioinformatics community. PLoS Comput Biol. 2011; 7:e1002216. https://doi.org/10.1371/journal.pcbi.1002216 PMID: 22046109

42. Ewels PA, Peltzer A, Fillinger S, Patel H, Alneberg J, Wilm A, et al. The nf-core framework for community-curated bioinformatics pipelines. Nat Biotechnol. 2020; 38:276–278. https://doi.org/10.1038/s41587-020-0439-x PMID: 32055031

43. scikit-learn. LogisticRegression. 2023 Jan [cited 2024 Aug 13]. scikit-learn [Internet]. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html.

44. Lipton Z. X post from 2019 Aug 30. X.com [Internet]. 2019 Aug 30 [cited 2023 Jan 25]. Available from: https://x.com/zacharylipton/status/1167298276686589953.

45. The R Core Team. R: A Language and Environment for Statistical Computing, Reference Index. Vienna, Austria: R Foundation for Statistical Computing; 2023 Oct. Available from: https://cran.r-project.org/doc/manuals/r-release/fullrefman.pdf.

46. Crawford J, Chikina M, Greene CS. Optimizer's dilemma: optimization strongly influences model selection in transcriptomic prediction. bioRxiv. 2023. p. 2023.06.26.546586. https://doi.org/10.1101/2023.06.26.546586

47. Brito JJ, Li J, Moore JH, Greene CS, Nogoy NA, Garmire LX, et al. Recommendations to enhance rigor and reproducibility in biomedical research. Gigascience. 2020;9. https://doi.org/10.1093/gigascience/giaa056 PMID: 32479592

48. Köster J, Rahmann S. Snakemake—a scalable bioinformatics workflow engine. Bioinformatics. 2012; 28:2520–2522. https://doi.org/10.1093/bioinformatics/bts480 PMID: 22908215

49. Di Tommaso P, Chatzou M, Floden EW, Barja PP, Palumbo E, Notredame C. Nextflow enables reproducible computational workflows. Nat Biotechnol. 2017; 35:316–319. https://doi.org/10.1038/nbt.3820 PMID: 28398311

50. Kurtzer GM, Sochat V, Bauer MW. Singularity: Scientific containers for mobility of compute. PloS One. 2017; 12: e0177459. https://doi.org/10.1371/journal.pone.0177459 PMID: 28494014

51. NumPy 1.16.6 release notes. NumPy manual [Internet]. 2019 Dec 29 [cited 2023 Oct 26]. Available from: https://numpy.org/doc/stable/release/1.16.6-notes.html.

52. NumPy v1.16.6 release. GitHub [Internet]. 2019 Dec 29 [cited 2023 Oct 26]. Available from: https://github.com/numpy/numpy/releases/tag/v1.16.6.

53. Perkel JM. Challenge to scientists: does your ten-year-old code still run? Nature. 2020; 584:656–658. https://doi.org/10.1038/d41586-020-02462-7 PMID: 32839567

54. Ramakodi MP. Influence of 16S rRNA reference databases in amplicon-based environmental microbiome research. Biotechnol Lett. 2022; 44:523–533. https://doi.org/10.1007/s10529-022-03233-2 PMID: 35122569

55. McLaren MR, Callahan BJ. Silva 138.1 prokaryotic SSU taxonomic training data formatted for DADA2. Zenodo. 2021. https://doi.org/10.5281/ZENODO.4587955

56. Differences between sed on Mac OSX and other "standard" sed? 2014 May 25 [cited 2024 Jun 20]. Unix & Linux Stack Exchange [Internet]. Available from: https://unix.stackexchange.com/a/131940/84206.

57. Free Software Foundation. sed, a stream editor. 2020 [cited 2024 Jun 20]. GNU Operating System [Internet]. Available from: https://www.gnu.org/software/sed/manual/sed.html.

58. FreeBSD Manual Pages. [cited 2024 Jun 20]. Available from: https://man.freebsd.org/cgi/man.cgi?sed.

59. Artaza H, Chue Hong N, Corpas M, Corpuz A, Hooft R, Jimenez RC, et al. Top 10 metrics for life science software good practices. F1000Res. 2016;5. https://doi.org/10.12688/f1000research.9206.1 PMID: 27635232

60. Ramakrishnan L, Gunter D. Ten Principles for Creating Usable Software for Science. 2017 IEEE 13th International Conference on e-Science (e-Science). IEEE. 2017. pp. 210–218. doi: 10.1109/eScience.2017.34

61. Jiménez RC, Kuzak M, Alhamdoosh M, Barker M, Batut B, Borg M, et al. Four simple recommendations to encourage best practices in research software. F1000Res. 2017; 6. https://doi.org/10.12688/f1000research.11407.1 PMID: 28751965

62. Queiroz F, Silva R, Miller J, Brockhauser S, Fangohr H. Good Usability Practices in Scientific Software Development. arXiv [csHC]. 2017. https://doi.org/10.48550/ARXIV.1709.00111

63. Hunter-Zinck H, de Siqueira AF, Vásquez VN, Barnes R, Martinez CC. Ten simple rules on writing clean and reliable open-source scientific software. PLoS Comput Biol. 2021; 17:e1009481. https://doi.org/10.1371/journal.pcbi.1009481 PMID: 34762641

64. Saia SM, Nelson NG, Young SN, Parham S, Vandegrift M. Ten simple rules for researchers who want to develop web apps. PLoS Comput Biol. 2022; 18:e1009663. https://doi.org/10.1371/journal.pcbi.1009663 PMID: 34990469

65. Romano JD, Moore JH. Ten simple rules for writing a paper about scientific software. PLoS Comput Biol. 2020; 16:e1008390. https://doi.org/10.1371/journal.pcbi.1008390 PMID: 33180774

66. Vidal N. The most popular licenses for each language in 2023. 2023 Dec 7 [cited 2024 Jun 20]. Open Source Initiative [Internet]. Available from: https://opensource.org/blog/the-most-popular-licenses-for-each-language-2023.

67. Fouad K, Vavrek R, Surles-Zeigler MC, Huie JR, Radabaugh HL, Gurkoff GG, et al. A practical guide to data management and sharing for biomedical laboratory researchers. Exp Neurol. 2024; 378:114815. https://doi.org/10.1016/j.expneurol.2024.114815 PMID: 38762093

68. Wilson SL, Way GP, Bittremieux W, Armache J-P, Haendel MA, Hoffman MM. Sharing biological data: why, when, and how. FEBS Lett. 2021; 595:847–863. https://doi.org/10.1002/1873-3468.14067 PMID: 33843054

69. Wilkinson MD, Dumontier M, Aalbersberg IJJ, Appleton G, Axton M, Baak A, et al. The FAIR Guiding Principles for scientific data management and stewardship. Sci Data. 2016; 3:160018. https://doi.org/10.1038/sdata.2016.18 PMID: 26978244

70. Tang YA, Pichler K, Füllgrabe A, Lomax J, Malone J, Munoz-Torres MC, et al. Ten quick tips for biocuration. PLoS Comput Biol. 2019; 15:e1006906. https://doi.org/10.1371/journal.pcbi.1006906 PMID: 31048830

71. Meyer MN. Practical Tips for Ethical Data Sharing. Adv Methods Pract Psychol Sci. 2018; 1:131–144. https://doi.org/10.1177/2515245917747656

72. Levenstein MC, Lyle JA. Data: Sharing Is Caring. Adv Methods Pract Psychol Sci 2018; 1:95–103. https://doi.org/10.1177/2515245918758319

73. Perkel J. Democratic databases: science on GitHub. Nature. 2016; 538:127–128. https://doi.org/10.1038/538127a PMID: 27708327

74. Mangul S, Mosqueiro T, Abdill RJ, Duong D, Mitchell K, Sarwal V, et al. Challenges and recommendations to improve the installability and archival stability of omics computational tools. PLoS Biol. 2019; 17:e3000333. https://doi.org/10.1371/journal.pbio.3000333 PMID: 31220077

75. Share & Preserve Code. [cited 2023 Jan 26]. University of Iowa Libraries [Internet]. Available from: https://www.lib.uiowa.edu/data/share-and-preserve-your-code/.

76. Potter M, Smith T. Making code citable with Zenodo and GitHub. 2016 Sep 26 [cited 2023 Jan 26]. Software Sustainability Institute [Internet]. Available from: https://www.software.ac.uk/blog/2016-09-26-making-code-citable-zenodo-and-github.

77. General Policies v1.0. [cited 2023 Oct 25]. Zenodo [Internet]. Available from: https://about.zenodo.org/policies/.

78. General Policies. 2017 [cited 2024 Aug 20]. Zenodo [Internet]. Available from: https://about.zenodo.org/policies/.

79. SRA FAQ. [cited 2024 Jun 24]. NCBI [Internet]. Available from: https://submit.ncbi.nlm.nih.gov/about/sra/.

80. Referencing and citing content. [cited 2024 Aug 20]. GitHub [Internet]. Available from: https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content.

81. Penfold NC, Polka JK. Technical and social issues influencing the adoption of preprints in the life sciences. PLoS Genet. 2020; 16:e1008565. https://doi.org/10.1371/journal.pgen.1008565 PMID: 32310942

82. Gewin V. How to include Indigenous researchers and their knowledge. Nature. 2021; 589:315–317. https://doi.org/10.1038/d41586-021-00022-1 PMID: 33437060

83. Tsosie KS, Fox K, Yracheta JM. Genomics data: the broken promise is to Indigenous people. Nature. 2021:529. https://doi.org/10.1038/d41586-021-00758-w PMID: 33742179

84. CARE Principles. [cited 2023 Dec 21]. Global Indigenous Data Alliance [Internet]. Available from: https://www.gida-global.org/care.

85. Watson M. When will "open science" become simply "science"? Genome Biol 2015; 16:101. https://doi.org/10.1186/s13059-015-0669-2 PMID: 25986601

86. Wilson G. Software Carpentry: lessons learned. F1000Res. 2014; 3:62. https://doi.org/10.12688/f1000research.3-62.v2 PMID: 24715981

87. Munafò MR, Nosek BA, Bishop DVM, Button KS, Chambers CD, du Sert NP, et al. A manifesto for reproducible science. Nat Hum Behav. 2017; 1:0021. https://doi.org/10.1038/s41562-016-0021 PMID: 33954258

88. Wolkovich EM, Regetz J, O'Connor MI. Advances in global change research require open science by individual researchers. Glob Chang Biol. 2012; 18:2102–2110. https://doi.org/10.1111/j.1365-2486.2012.02693.x

89. Emery N, Crispo E, Supp SR, Kerkhoff AJ, Farrell KJ, Bledsoe EK, et al. Training Data: How can we best prepare instructors to teach data science in undergraduate biology and environmental science courses? bioRxiv [Preprint]. 2021. p. 2021.01.25.428169. https://doi.org/10.1101/2021.01.25.428169