Manuscript Title

This manuscript (permalink) was automatically generated from computer-aided-biotech/better-cb@96a44f4 on February 15, 2021.

Authors

- **Benjamín J. Sánchez**
  0000-0001-6093-4110 · BenjaSanchez · BenjaSanchez
  Department of Bioengineering, Technical University of Denmark, Kgs. Lyngby, 2800, Denmark

- **Daniela C. Soto**
  0000-0002-6292-655X · dcsoto
  Genome Center, MIND Institute, and Department of Biochemistry & Molecular Medicine, Davis, CA 95616,USA

Abstract

This should be the abstract.

Introduction

Since Margaret Dayhoff pioneered the field of bioinformatics back in the sixties, the application of computational tools to the field of biology has vastly grown in scope and impact. Nowadays, biotechnological and biomedical research are routinely fed by the insights arising from novel computational approaches, machine learning algorithms and mathematical models. The ever increasing amount of biological data and the exponential growth in computing power will amplify this trend in the years to come.

The use of computers to address biological matters encompasses a wide array of applications usually grouped under the terms of "computational biology" and "bioinformatics". Although distinct definitions have been delineated for each one [1,2], here we will consider both under the umbrella term "computational biology", alluding to any application that involves the intersection of computing and biological data. As such, a computational biologist can be a data analyst, a data engineer, a statistician, a mathematical modeler, a software developer, and many others. In praxis, the modern computational biologist will be a "scientist of many hats", taking on several of the duties listed above. But first and foremost, we will consider a computational biologist as a scientist whose ultimate goal is answering a biological question or addressing a need in the life sciences, by means of computation.

Scientific computing requires following specific practices to enable shareable, reproducible and sustainable outputs that stand the test of time. Computing-heavy disciplines such as software engineering and data science have already adopted practices addressing the need for collaboration, visualization, project management, and strengthening of online communities. However, as a highly interdisciplinary and evolving field, computational biology has yet to acquire a set of universal "best practices". As most computational biologists come from diverse backgrounds and oftentimes lack formal computational training, the absence of guidelines can lead to disparate and unsustainable practices that hinder reproducibility and collaboration, and slow down biomedical and biotechnological research.

Over the last decade, several researchers have published advice directed to bench scientists starting in either scientific computing [3,4] or computational biology [5]. The advice encompasses a wide range of topics ranging from programming to project organization to manuscript writing. Other works have adopted a different approach, fixating in one powerful tool and diving deeper into its scope and applications, such as the software development and version control cloud service GitHub [6] and the web-application Jupyter Notebooks [7]. Similarly, recent works have chosen to comprehensively address one specific need in computational biology such as workflow automation [8] and software library development [9]. Although this advice proves immensely helpful, several aspects of the computational biology "journey" remain uncovered. Specifically, there is a lack of a clear roadmap regarding best practices from fundamental to advanced topics, in addition to practical examples tackling the complexities of this kind of research.

We premise that best practices in computational biology lie within a continuum that traverses three "levels": the individual's internal practices, the collaborative practices of a team, and the practices that allow a broader scientific community to access and engage over time with the research (Figure 1). Each one of these levels has a different set of needs and challenges. Here, we compile a selected list of relevant practices and related tools advisable for each one of these levels, emphasizing their time and place in a computational biology research project. Finally, we illustrate the utility of these practices in three case studies covering the broader spectrum of computational biology research.

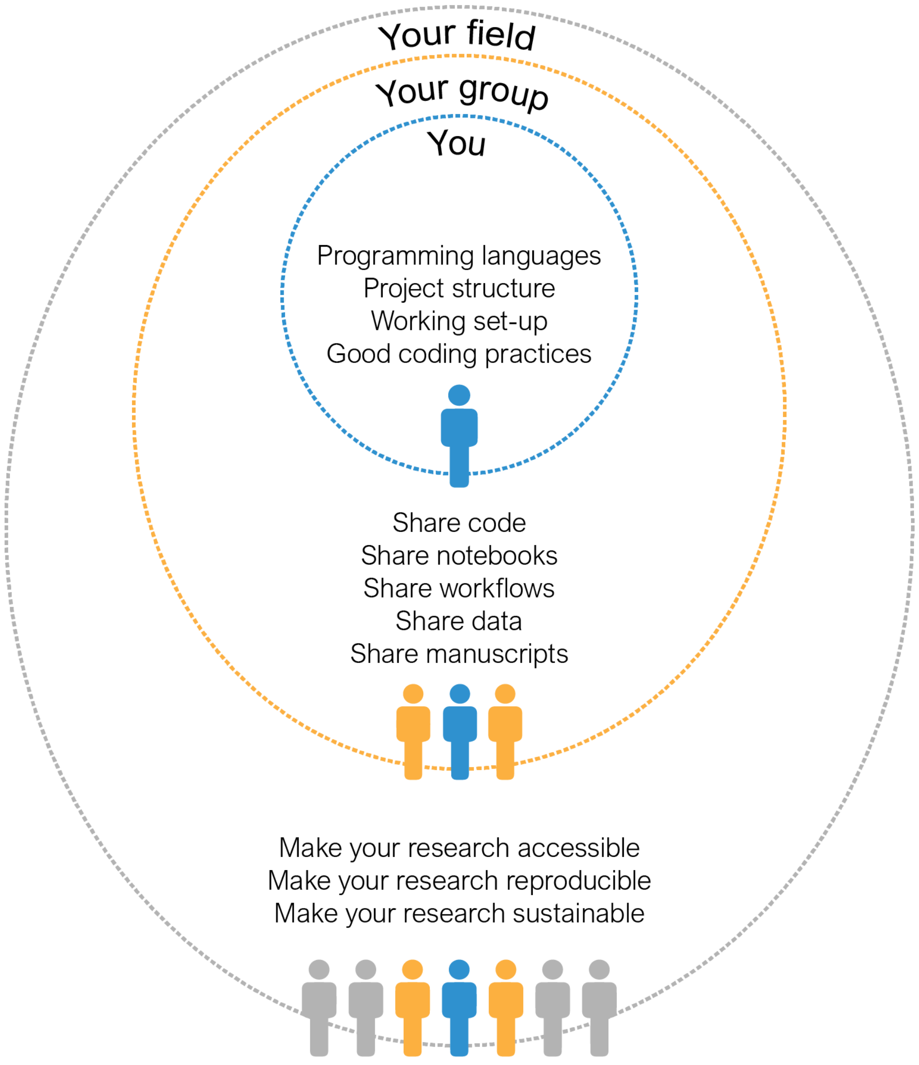

Figure 1: The three "levels" of computational biology include your personal research, your group and collaborators, and your scientific field.

Level 1: Personal Research

The computational biology "journey" begins with you—specifically, with the set of skills, tools and practices that you have put in place to conduct your research. Taking the time to optimally establish these building blocks will have high payoffs later, when you need to check your previous research (which will certainly happen). Consider that your most important collaborator is your future self, either of tomorrow or ten years from now. To consider all the aspects that make up a solid ground for any computational biology project, we devised a framework involving four main sequential steps (Table 1).

Step 1: Choose your programming languages

Different programming languages serve distinctive purposes and have different idiosyncrasies. As such, choosing the right programming language for a project depends on your research goals and personal preference or skill. Additionally, communities usually favor the usage and training of some programming languages over others; utilizing such languages may facilitate integrating your work within the existing ecosystem.

As computational biology becomes a data intensive discipline, interacting with high-performance computing (HPC) operating systems has become a hallmark of the field. HPC infrastructures commonly use Unix/Linux distributions as their operating system. To interact with these platforms, you need to use a the command line interpreter known as the “Unix shell”. There are multiple versions of Unix shells, being Bash one of the most widely adopted. Besides providing an “user interface”, the shell is also a scripting language that allows manipulating files and executing programs through “shell scripts”. Unix/Linux operating systems have other interesting perks, such as powerful and fast commands for searching words and manipulating files (e.g. `sed`, `grep` or `join`) as well as the language AWK, that can perform quick text processing, including arithmetic operations.

One of the most common task of any computational biologist is data analysis. The job of a data analyst involves data cleaning, exploration, manipulation, and visualization. Currently, Python is the most widely used programming language for data analysis worldwide [10,11]. Computational biology research has followed this trend, making Python one of the most popular languages among researchers. As machine learning and deep learning are more widely adopted in biological research, Python usage will keep growing. Python usage has been facilitated by the availability of packages for biological data analysis accessible through package managers such as Pip or Conda. Similarl to Python, R is the other prominent language in the field. Arguably, one of R main strengths is its wide array of tools for statistical analysis. Of particular interest is the Bioconductor repository, where many gold-standard tools for biological data analysis have been published. R usage in data science has deeply benefited from the Tidyverse packages, increasing the readability of the R syntax for both data manipulation via `dplyr`, and visualization via `ggplot2`.

Oftentimes, computational biologists require coding their own sets of instructions for processing data via scripts or programs. A script can be described as lightweight single-file program developed to tackle a narrow purpose. They are likely written in an interpreted programming language instead of a compiled one. Interpreted programming languages execute the program directly, without previous compilation, meaning each statement is run individually. They are quick to edit and can be run interactively, at expense of computational performance. In computational biology, the current most common multi-purpose scripting languages are Python and R. When working in a HPC, Shell/Bash scripting is also widely used. Perl and Matlab are also a popular language among bioinformatics and systems biology, respectively. A program, in the other hand, is a larger tool that usually combines multiple scripts and works as a “black box” to the user. It is designed to tackle computationally intensive problems, thus, a compiled language is preferred. Several tools designed for high-weight biological data processing have been written in C/C++. In recent years, however, scientists have been turning to Rust because of its speed, safety and friendly community [12]. If computational performance is not a concern, Python and R are suitable alternatives for coding programs.

Biological data processing is rarely a one-step process. To go from raw data to useful insights, several steps need to be taken in a specific order, accompanied by a plethora of decisions regarding parameters. Computational biologists have addressed this need by embracing workflow management systems to automate data analysis pipelines. A pipeline can be written as a Shell script where a handful of commands are written one after another, using Shell variables and Shell scripting syntax when needed. Although effective, this approach provides little control over the workflow, and lacks features to run isolated parts of the pipeline or track changes in input and output files. To overcome these limitations, a Bash script can be “upgraded” using the GNU Make program, which was originally designed to automate compilation and installation of software, but that it is flexible enough for building other types of workflows. Nowadays, however, several dedicated bioinformatics workflow managers have been developed. Snakemake is a workflow management system written in Python, allowing to incorporate the syntax of the tool with standard Python code. Similarly, Nextflow was build as an extension of the Groovy—a programming language for the Java virtual machine—and can execute Groovy code. These tools not provide control over any step of the pipeline, but also offer features like interacting with job schedulers, software environment managers, and cloud computing support. Alternatively, workflows can be written in the Common Workflow Language (CWL)—a declarative standard to define workflows with the goal of enabling portability and reproducibility. Workflows written in CWL can be run in any CWL-enabled engine.

Step 2: Define your project structure

After choosing your programming languages and before starting any coding, we advice you to come up with a well-thought-out project structure. This design should be intentional and tailored to the present and future needs of your project—remember to be kind to your future self. A computational biology project requires, at the very least, a folder structure that supports code, data, and documentation. Although tempting, cramming all kind of files in a unique folder is unsustainable. Instead, separate each one on different folders and subfolders if needed. As additional principles consider documentation as a requirement and raw data as immutable. To simplify this process, you can base your project structure on research templates available off-the-rack. For data science projects, the python package Cookiecutter Data Science cut down the effort to the very minimum [13]. Running the package prompts a questionnaire in the terminal where you can input the project name, authors and other basic information. Then, the program generates a folder structure to store data—raw and processed—separated from notebooks and source code, as well as pre-made files for documentation such as a “readme”, a docs folder and a license. Similarly, the Reproducible Research Project Initialization (rr-init) offers a template folder structure that can be cloned from a GitHub repository and modified by the user [14]. Although the later is simpler than the former, both follow an akin philosophy aimed at research correctness and reproducibility [15]. For workflow automation projects, we advice a similar folder structure. Snakemake recommends to store each workflow in a dedicated folder separated into workflow-related files—the Snakefile, rules and scripts—results and configuration [16]. In all cases, the folder must be initialize as a git repo for version control (See Step 4).

Beyond files and folders, the software and dependencies needed to run an analysis, workflow or program, are also part of the project structure itself. The intricacies of software installation and dependency management are not to be underestimated. Fortunately, package and virtual environment managers significantly reduce this burden. A package manager is a system that automates the installation, upgrading, configuration and removing of community-developed programs; a virtual environment manager, in the other hand, is a tool that generates isolated “environments” containing programs and dependencies that are functionally independent from other environments or the default operating system. Once a virtual environment is activated, a package manager can be used to install third-party programs.

We believe that a computational biology project must start with its own virtual environment. The main reason is reproducibility: environments save the project’s dependencies and can restore them at will so the code can be run in any other computer. There are multiple options for both package and virtual environment management—some are language-specific; others, language-agnostic. If you are working with Python, you can initialize a Python environment using virtualenv or pipenv (where different Python versions can be installed). Inside the environment, you can use the Python package manager Pip to add Python code submitted to the Python Package Index (PyPI), GitHub or locally. For the R language, R-specific environments can be created using `renv`. After initializing the environment, packages can be installed via `install.packages` function from the Comprehensive R Archive Network (CRAN) and CRAN-like repositories. R also has BiocManager to install packages from the Bioconductor repository, which contains relevant software for high-throughput genomic sequencing analysis. To fully manage a dependencies beyond installation, RStudio developed the RStudio Package Manager which works with third-party code available in CRAN, Bioconductor, GitHub or locally. A language-agnostic alternative is Conda—an increasingly popular package manager and a virtual environment manager. It supports program installation from the Anaconda repository, which contains the channel Bioconda specifically tailored to bioinformatics software. Also, if Python is installed, Python dependencies can be installed via pip. Conda is particularly helpful when working with third-part code in all sorts of languages—a common predicament for the computational biologist. Conda package and environment manager is included in both Miniconda and Anaconda distributions. The former is a minimal version of Anaconda, containing only Conda, Python, and a few useful packages.

Step 3: Choose your working set-up

With the foundation in place, the next step is to start coding. However, a more practical question needs to be answer first: where to code. The simplest tool available for this purpose are text editors. Since writing code is ultimately writing text, any tool where characters can be typed fulfills this purpose. However, coding can be streamlined by additional features as those found in code editors—text editors especially developed for writing code. Crucial features to facilitate coding include syntax highlight, indentation or autocompletion. Commonly used desktop editors include Atom, Sublime, Visual Studio Code, and Notepad++ (Windows only), all which a myriad of plugins available to enhance the coding experience. Command line text editors are also suitable options for coding, being Vim and Emacs the most powerful ones. All of these tools share the advantage of being language agnostic, allowing easy switching between languages, especially handy for the polyglot computational biologist.

In addition to text editors, integrated development environment (IDE) are also popular options for coding. As its essence, IDE are supercharged text editors, i.e. with multiple other features that make writing code easier. The main parts of an IDE are a code editor (with syntax highlight, indentation and suggestions), a debugger, a folder structure, and a way to execute your code (a compiler or interpreter). IDEs are not language agnostic, meaning that they allow to code in one language. The array of features also comes at a cost—IDEs usually use more memory and imply more visual clutter, if that is a concern of yours. For Python, Jupyter Lab, Spyder and JetBrains' PyCharm are popular options, while for R, RStudio is the gold-standard. Notably, the differences between an IDE and a code editor are somewhat blurry, especially when enough plugins have been added to a code editor.

In the latest years, notebooks have acquired relevance in computational biology research. A notebook is an interactive application that combines live code (read-print-eval loop or REPL), narrative, equations and visualizations. Common notebooks use an interpreted languages such as Python or R, and narrative follows markdown syntax. Data analysis greatly benefits from using notebooks instead of plain text editors or even IDEs: the combination of visuals and texts allows researcher to tell compelling stories about their data, and the interactivity of its code enables quick testing of different strategies. Jupyter notebook is a popular web-based interactive notebook developed originally for Python coding, but also accepts R and other programming languages upon installation of their kernels—a computing engine that executes the notebook's live code "under the hood". Jupyter notebook can also be run in the cloud using platforms such as Google CoLab and Amazon WebServices, taking advantage of the current trend of cloud computing. RStudio also allows the generation of R-based notebooks known as R Markdown, which is specially well-suited for generating reports.

Step 4: Follow good coding practices

After dealing with steps one to three, finally comes writing code. Coding, however, requires good practices to ensure correctness, sustainability and reproducibility for you, your future self, your collaborators (as we will discuss in Level 2) and the whole community (as we will discuss in Level 3). First and foremost, you need to make sure your code works correctly. In computational biology, correctness implies biological and statistical soundness. Both are big topics beyond the scope of this manuscript. To achieve the former, however, a useful approach is to design positive and negative controls in your program, analysis or workflow. In an experiment, a positive control is a control group that is expected to produce results; a negative control is expected to produce no results. The same approach can be applied to computation, using input data whose output is previously known. Biological soundness can also be tested by quickly assessing expected orders of magnitude in both, intermediate and final files.

Beyond the correct functioning of your code, you will need to pay attention to the way your code looks, also know as “coding style”. This includes a series of small and ubiquitous decisions regarding where and how to add comments; indentation and white spaces usage; variable, function and class naming; and overall code organization. It is true that, as in writing, there is a lot of your own personality in the way you code. However, sticking to existing coding style rules facilitates collaborations with your future self and others. Indeed, as we sometimes have trouble reading our own handwriting, we can also struggle reading our own code if we overlook any guidelines. At the very least, your code must display internal consistency. Even better, you can follow any of the multiple coding style rules that have been published. Although arbitrary, most of these rules have been developed with readability in mind. A good place to start, however, are style guides from software development teams. Google, for example, has published guidelines for Python, R, Shell, C++, and HTML/CSS [17]. Also, a series of guidelines for the Python programming language have been published with the name of Python Enhancement Proposal (PEP), where the most widely adopted is PEP-8 [18]. To aid flagging stylistic errors in your code, tools called “linters” are usually included with code editors and IDEs or provided as plugins.

In the matter of code styling, two topics merit additional attention: variable naming and comments. Variable names should be descriptive enough to convey an idea about the variable, function or class’ content and use. The goal is to produce “self-documented” code that reads close to plain English. To do so, use multi-words variable names if necessary. In such cases, the most common conventions include Camel Case, where the second and subsequents words are capitalized (“camelCase”); Pascal Case, where all words are capitalized (“PascalCase”); and Snake Case, where words are separated by underscores (“snake_case”). Notably, all these conventions can be used in a same coding style to differentiate variables, functions and classes. For example, PEP-8 recommends Snake Case for functions and variables, and Pascal Case for class names. In addition to master variable naming, code comments—explanatory human-readable statements not evaluated by the program—are necessary to enhance the code’s readability. No matter how beautiful and well-organized your code is, high-level code decisions will not be obvious unless stated. As a corollary, code explanations that can be deduced from the syntax itself should be omitted. Comments can span a single line or several ones, forming a block, and can be found in three strategic parts: at the top of the program file (“header comment”), which contains the author and the date of the code and what it accomplishes; above every function (“function header”), which contains the purpose and behavior of the function; and in line, next to tricky code whose behavior is not obvious or warrant a remark.

When working with a sizable code base, a good practice related is to strive for modularity—splitting your code’s functionalities into independent entities known as modules. Modularity enhances code readability and reusability—enabling your code to be used by you or others in future applications—and expedites maintenance. In Python, subdivisions are defined as follow: a module is a collection of functions and global variables, a package a collection of modules, a library a collection of packages, and a framework a collection of libraries. Modules are simply files with the `.py` extension. Packages, in the other hand, must be indicated to the Python interpreter adding a file named `__init__.py` (which could be empty or not).

Code styling rules also apply to data science notebooks. However, when writing notebooks you must also engage in “literate programming”—a programming paradigm where the code is accompanied by human-readable explanation of its logic and purpose. In other words, notebooks must tell a story about the analysis, connecting the dots between the code, the results and the figures. Human-readable language is often written in Markdown—a lightweight markup language. Little has been written about good practices for literate programming, but we advice you to explain the purpose of each chunk of code and provide some interpretation of its results.

Equally as important as writing good code is to use version control—the practice of tracking and managing changes in your code. This is a good and necessary practice even if your only collaborator is your future self. One of the main advantages of version control is keeping a change log of your files that can be utilized to go back to previous versions of your code, and remind you of previous approaches disregarded in newer versions. The most widely used version control system, Git, achieves these tasks with the command `git checkout`. Additionally, version control also allows you to safely try new functionalities using "branches"—carbon copies of the main original branch (known as "master") where you can add code independently and optionally merge it to the original one. Git creates branches using the command `git branch` and the same `git checkout` can be used to switch among them. We will discuss the utility and implementation of branches in collaborative projects in the next section (See Level 2: Collaboration). Nowadays, there are multiple code repositories that also provide version control with Git, such as GitHub, GitLab or Bitbucket. They have the additional benefit of backing up your code and code history in the cloud, keeping your work safe and shareable. You can interact with these platforms using the browser or via graphic user interfaces (GUI) such as GitHub Desktop or GitKraken.

Table 1: Steps to start any computational biology project.

Step	Use case	Common tools
Step 1: Choose your programming languages	Interacting with a Unix/Linux HPC	• Shell/Bash
	Data analysis	• Python, R
	Scripts and programs:	• Interpreted: Python, R, Perl • Compiled: C/C++, Rust
	Workflows	• Snakemake (Python), Nextflow (Groovy), CWL
Step 2: Define your project structure	Project structure	• Templates: Cookiecutter Data Science, rr-init • Workflows: Snakemake structure
	Virtual environment managers	• Language-specific: pipenv (Python), virtualenv (Python), renv (R) • Language agnostic: Conda
	Package managers	• Language-specific: pip (Python),BiocManager (R), R Studio package manager (R) • Language-agnostic: Conda
Step 3: Choosing your working set-up	Text editors	• Desktop applications: Atom, Sublime, Visual Studio Code, Notepad++ • Command line: Vim, Emacs
	IDEs	• For Python: Jupyter Lab, JetBrains/PyCharm, Spyder • For R: R Studio
	Notebooks	• Jupyter (Python, R), R Markdown (R)
Step 4: Follow good coding practices	Coding style	• Styling guides: PEP-8 (Python), Google (Python, R) • Linters
	Literate programming	• Markdown

Step	Use case	Common tools
	Version control	• Version control system: Git • Code repositories: GitHub, GitLab, Bitbucket • Git GUIs: GitHub Desktop, GitKraken

Level 2: Collaboration

Goal: How to allow your collaborators to reproduce and interact with your research/software?

Topics: - Sharing notebooks: R Markdowns and Jupyter Notebooks (Binder, Google CoLab) - Sharing apps: Shiny apps, Dashboard. - GitHub (branching, pull requests) - Workflow automation: Snakemake (NextFlow, Make, Bash script) - Sharing data and metadata

Level 3: Community

Goal: How to develop and maintain a computational biology project with community feedback over time?

Topics: - GitHub releases and semantic versioning - Git Flow, GitHub Issues - Continuous integration and unit tests - Dependencies per user: pip-tools - Sharing software as Python packages, Conda/Bioconda or containers (Docker, Singularity) - Include license (MIT) and DOIs - Documentation: read the docs.

Case studies

- Example of computational biology project 1: RNA-seq analysis (workflow)
- Example of computational biology project 2: Genome-scale metabolic model (systems biology project)
- Example of computational biology project 3: Software development (computational tool)

Conclusion

Pending

Acknowledgments

Pending

References

1. **NIH working definition of bioinformatics and computational biology**
   Huerta Michael, Downing Gregory, Haseltine Florence, Seto Belinda, Liu Yuan
   (2000-07-17)
   https://www.kennedykrieger.org/sites/default/files/library/documents/research/center-labs-cores/bioinformatics/bioinformatics-def.pdf

2. **What is bioinformatics? A proposed definition and overview of the field.**
   NM Luscombe, D Greenbaum, M Gerstein
   Methods of information in medicine (2001) https://www.ncbi.nlm.nih.gov/pubmed/11552348
   PMID: 11552348

3. **Good enough practices in scientific computing**
   Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, Tracy K. Teal
   PLOS Computational Biology (2017-06-22) https://doi.org/gbkbwp
   DOI: 10.1371/journal.pcbi.1005510 · PMID: 28640806 · PMCID: PMC5480810

4. **Software engineering for scientific big data analysis**
   Björn A Grüning, Samuel Lampa, Marc Vaudel, Daniel Blankenberg
   GigaScience (2019-05) https://doi.org/gf4f4m
   DOI: 10.1093/gigascience/giz054 · PMID: 31121028 · PMCID: PMC6532757

5. **So you want to be a computational biologist?**
   Nick Loman, Mick Watson
   Nature Biotechnology (2013-11-01) https://doi.org/p3j
   DOI: 10.1038/nbt.2740 · PMID: 24213777

6. **Ten Simple Rules for Taking Advantage of Git and GitHub**
   Yasset Perez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, … Juan Antonio Vizcaíno
   PLOS Computational Biology (2016-07-14) https://doi.org/gbrb39
   DOI: 10.1371/journal.pcbi.1004947 · PMID: 27415786 · PMCID: PMC4945047

7. **Ten Simple Rules for Reproducible Research in Jupyter Notebooks**
   Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, Peter W. Rose
   arXiv (2018-10-13) https://arxiv.org/abs/1810.08055v1

8. **Streamlining Data-Intensive Biology With Workflow Systems**
   Taylor Reiter, Phillip T. Brooks, Luiz Irber, Shannon E. K. Joslin, Charles M. Reid, Camille Scott, C. Titus Brown, N. Tessa Pierce
   Cold Spring Harbor Laboratory (2020-11-16) https://doi.org/gg353v
   DOI: 10.1101/2020.06.30.178673

9. **A Padawan Programmer's Guide to Developing Software Libraries**
   James T. Yurkovich, Benjamin J. Yurkovich, Andreas Dräger, Bernhard O. Palsson, Zachary A. King
   Cell Systems (2017-11) https://doi.org/gg8tqz
   DOI: 10.1016/j.cels.2017.08.003 · PMID: 28988801

10. **2018 Kaggle Machine Learning & Data Science Survey** https://kaggle.com/kaggle/kaggle-survey-2018

11. **Modulecounts** http://www.modulecounts.com/

12. **Why scientists are turning to Rust**
Jeffrey M. Perkel
Nature (2020-12-01) https://doi.org/ghqc7g
DOI: 10.1038/d41586-020-03382-2 · PMID: 33262490

13. **Home - Cookiecutter Data Science** https://drivendata.github.io/cookiecutter-data-science/

14. **Reproducible-Science-Curriculum/rr-init**
Reproducible Science Curriculum
(2021-02-12) https://github.com/Reproducible-Science-Curriculum/rr-init

15. **A Quick Guide to Organizing Computational Biology Projects**
William Stafford Noble
PLoS Computational Biology (2009-07-31) https://doi.org/fbbpkn
DOI: 10.1371/journal.pcbi.1000424 · PMID: 19649301 · PMCID: PMC2709440

16. **Distribution and Reproducibility — Snakemake 5.32.2 documentation**
https://snakemake.readthedocs.io/en/stable/snakefiles/deployment.html

17. **google/styleguide**
Google
(2021-02-15) https://github.com/google/styleguide

18. **PEP 8 – Style Guide for Python Code**
Python.org
https://www.python.org/dev/peps/pep-0008/